Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

A Serverless Gateway for Event-driven Machine Learning Inference in Multiple Clouds

Diana M. Naranjo | Sebastián Risco | Germán Moltó | Ignacio Blanquer

Instituto de Instrumentación para Imagen Molecular (I3M), Centro mixto CSIC - Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, España

Correspondence
Email: {dnaranjo,serisgal}@i3m.upv.es, {iblanque,gmolto}@dsic.upv.es

Summary

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

KEYWORDS:
Cloud Computing, Serverless Computing, Function as a Service, Machine Learning

1 | INTRODUCTION

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

The FaaS model is a part of serverless computing, which also includes the BaaS (Backend as a Service) category. It is considered an evolution of cloud programming models, with a higher level of abstraction, where the cloud provider dynamically manages the provisioning of resources. Serverless computing, and particularly the FaaS model, has become a paradigm for the deployment of applications in the Cloud, primarily because of the advantages it provides to developers with respect to the adoption of containers and microservices-based architectures [1]. Indeed, one of the fundamental challenges in the transition to serverless computing for a microservices-based architectures is that applications must be designed as a set of functions.

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

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

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

When using platform services from public cloud providers, there is a risk of being dependent on the services and products they offer. Indeed, FaaS APIs and formats strongly differ among providers. This dependency is often referred to as "vendor lock-in", since switching technologies and vendors may be costly. In order to mitigate this risk in the use of the resources for serverless computing from public cloud providers, several open-source frameworks have emerged, as is the case of OpenFaaS[2], Knative[3] and Apache OpenWhisk[4]. In this sense, the emergence of containers and the development of Container Orchestration Platforms (COPS), such as Kubernetes, facilitate the implementation of FaaS models in open-source platforms such as those mentioned above.

In parallel, the development of artificial intelligence and machine learning has led companies to include these types of services due to the wide range of benefits that can affect all aspects of human life, such as customer service, detection of fraud and business intelligence[5]. Artificial intelligence combined with cloud computing is seen as the next step in machine learning automation[6].

Deploying machine learning models on local servers has a certain complexity mainly due to the lack of high-end local computing power, which introduces significant delays in the inference and training processes. In fact it is a reality that regular maintenance and scaling is becoming increasingly complex[7]. In this sense, serverless computing emerges as a profitable and scalable solution that allows addressing the main challenges in terms of excessive resource provisioning and simplifies the implementation of the underlying infrastructure.

In order to address these challenges, this paper introduces a web-based serverless scientific gateway that supports the inference phase of machine learning and artificial intelligence models on dynamically scalable serverless platforms. Execution of the models can be done in public or in on-premises clouds, as specified in the web interface. This platform constitutes an extension of the work carried out by Naranjo et al.[8,9] presented in Gateways 2020. In our previous work, only the deployment in a public cloud was supported. For this contribution, we have included an analysis of execution times and economic cost of the platform and the deployment in on-premises cloud with the OSCAR framework, thus supporting multi-cloud infrastructures. The fundamental objective of this platform is to support the inference of machine learning and artificial intelligence models in multi-clouds by abstracting the configuration, management and scaling details of the underlying infrastructure by adopting serverless computing both from on-premises and public clouds.

After the introduction, the structure of the article is as follows. First, section 2 introduces the related works in the area of the execution of machine learning and artificial intelligence models on serverless platforms, both in public and on-premises clouds. Next, section 3 presents the components and the architecture of the platform. Then, section 4 introduces the models integrated in the platform. Later, section 5 presents the results of the use cases and section 6 discusses the results obtained. Finally, section 7 summarizes the main achievements and future works are presented.

2 | RELATED WORK

Previous research exposes the advantages of the serverless paradigm in scientific computing. This is the case of the work by Spillner et al.[10] which introduces the benefits of adopting the FaaS model for multiple scientific applications such as computer graphics, cryptology, mathematics, and meteorology. A study by Baldini et al.[11] analyzes existing serverless platforms by identifying key features, use cases, and describing technical challenges and open issues. The conclusions of this research indicates that the FaaS model appropriately adapts to a number of distributed applications, including event processing pipelines in compute-intensive applications.

One of the pioneers in the use of the FaaS model was Jonas et al.[12] who introduced the PyWren framework in order to perform Python-based distributed computing on AWS Lambda to support different distributed computing models efficiently.

Later, this study was expanded in [13] presenting *numpywren*, a linear algebraic system built on a serverless platform. In addition, LAmbdaPACK is presented, a domain-specific language designed to implement highly parallel linear algebra algorithms in a serverless environment.

A more recent study by Eismann et al. [14] presents a guide for the design of new serverless approaches examining 89 use cases obtained from other scientific literatures. Each case is studied by analyzing different characteristics that include general aspects, but also workloads, applications and requirements. The work carried out by Jindal et al. [15] introduces an extension of FaaS to computing clusters, to support functions across a network of distributed heterogeneous target platforms, called Function Delivery Network (FDN). As a result, the varied characteristics of the target platform, the possibility of collaborative execution between multiple target platforms, and the data localization provided by FDN are shown. The work done by Mahmoudi and Khazaei [16] introduces SimFaaS, an open-source tool written in Python that allows to simplify the validation process of a performance model developed on serverless public computing platforms. Through SimFaaS it is possible to predict various metrics related to service quality such as cold start, average response time and the probability of rejection of requests that helps to understand the limits of the system and measure the compliance with the Service Level Agreement (SLA) without the need for expensive experiments.

Serverless computing also covers other fields of computing such as the analysis of large amounts of data (Big Data). The work by Giménez-Alventosa et al. [17] presents MARLA (MapReduce on AWS Lambda) [†] a high performance open-source serverless architecture to run MapReduce jobs on AWS Lambda and Amazon S3, without the need for the user to pre-provision the computing infrastructure.

As stated in the introduction, an important element to consider in the adoption of serverless computing is the risk of vendor lock-in with the technologies and services of public cloud providers. In this scenario, it is difficult to migrate to a different provider without substantial cost due to the technical incompatibilities [18]. In order to mitigate this phenomenon, developers have focused on creating open-source solutions such as OpenFaaS [2], Knative [3], Fission [19], Nuclio [20], Apache OpenWhisk [4], and Oracle Cloud Fn [21], to name a few. These platforms support the definition and execution of functions in response to certain events. The difference between them is fundamentally in the programming language they support, the event sources and in the use of an orchestration platform such as Kubernetes.

The study conducted by Hendrickson et al. [22] presents OpenLambda, an open-source platform for running applications and web services based on a serverless architecture. In the work presented by Kaviani et al. [23] Knative compared with other serverless platforms in order to extract a minimal execution model with a common denominator that is close to a unified serverless platform. Palade et al. [24] performed an analysis of four open-source serverless frameworks: Kubeless, Apache OpenWhisk, OpenFaaS and Knative in some typical scenarios related to edge computing and IoT (Internet of Things) networks. The results of this research indicate that Kubeless surpasses the other frameworks in terms of response, time and performance.

The work carried out by Li et al. [25] presents an analysis of open-source serverless frameworks taking into account platform design problems that affect performance. They determine that simple autoscaling based on resources or workloads is not adequate to meet the needs of serverless platforms. In the work developed by Benedetti et al. [26] the suitability of a local serverless platform for IoT applications, implemented through OpenFaas, is discussed and analyzed. A performance study is presented taking into account latency and resource consumption for the cold and warm boot deployment mode.

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

The article conducted by Corral-Plaza et al. [27] presents an analysis of the main options for machine learning available in the cloud. The work is focused on the BigML [§] platform and Amazon Machine Learning. Another study by Ishakian et al. [28] evaluates the suitability of AWS Lambda to serve lightweight deep learning models. As a result, an analysis is made of how cold start influences processing performance and AWS Lambda storage limits restrict the implementation of larger models. In the work done by Kodandarama et al. [29] the feasibility of implementing the inference phase on a serverless platform using services provided by AWS is studied. Research results show that serverless platforms show promise for implementing the inference phase of machine learning models.

In the work by Bhattacharjee et al. [30] it is presented Barista, a local serverless platform for the implementation of machine learning models based on OpenStack to execute predictions, selecting the configuration of the virtual machine based on the

[†] MARLA - https://github.com/grycap/marla
[‡] Amazon SageMaker - https://aws.amazon.com/sagemaker/
[§] BigML - https://bigml.com/

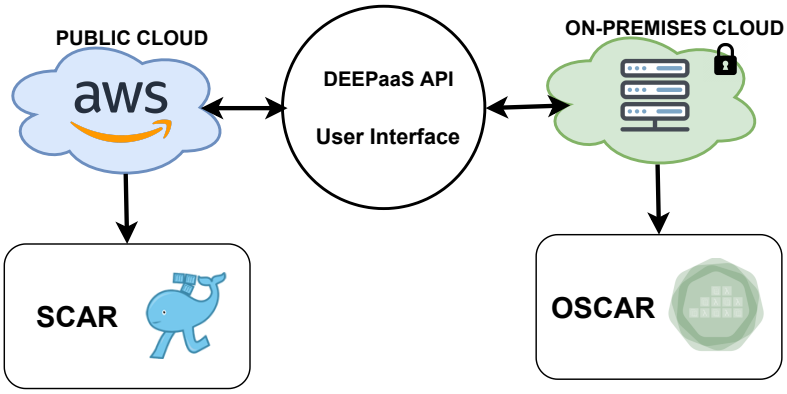

FIGURE 1 Components of the designed architecture.

objectives of service level, cost, and time of execution. For its implementation, this system requires a machine powerful enough to support the framework. The articles conducted by Christidis et al.[31 32] propose a set of optimization techniques for the implementation of machine learning models on a serverless platform, without compromising capacity or performance. The results obtained indicate the feasibility of using serverless platforms in the implementation of machine learning and artificial intelligence models. A recent work presented by Kurz[33] analyzes the feasibility of implementing double machine learning, a method based on the estimation of primary and auxiliary predictive models[34], on AWS Lambda, taking advantage of the high level of parallelism that can be achieved with serverless computing. In the case study analyzed in the research, an implementation written in Python called *DoubleML-Serverless* is presented, where its usefulness is demonstrated by analyzing the execution times and estimating the costs.

In the work by Ishakian et al.[28] a series of experiments are performed to run Amazon MXNet machine learning models on AWS Lambda. The objective in this research is to measure efficiency in terms of processing time, scalability, and memory used. The results obtained demonstrate that the use of a serverless platform is adequate to obtain the prediction of the models, as long as they are integrated into the AWS platform and that they comply with the limitations of AWS Lambda. Other research presents SerFer[29] as an inference system for machine learning applications in the AWS cloud. In this system the inference is restricted to AlexNet, a convolutional neural network (CNN)[35], and the implementation is based on a system that executes the inference phase in an EC2 instance.

The challenges in the execution of machine learning models are limited memory and compute capacity, together with long execution times. Serverless computing allows the implementation of these applications in a more cost-effective way, especially in the inference phase where large amounts of resources are required in a short execution time. In order to address the main challenges, this document presents a serverless architecture integrated with these type of applications, where the inference phase of machine learning models can be executed in a public cloud or on-premises clouds using serverless computing strategies. Access to the models is implemented through a web-based scientific gateway, which facilitates their use by users without experience in this type of technology. The models implemented in the use cases and the tools used to design the platform are open-source and publicly available in GitHub: models[36], SCAR[37], DEEPaaS[38] and web-based scientific gateway[39].

3 | COMPONENTS AND ARCHITECTURE

This section introduces the main components used to create the web-based scientific gateway to support the inference phase of machine learning models from multi-clouds based on the serverless model (both public and on-premises clouds). Figure 1 shows the components used in this development. Two fundamental deployment methods are identified in the developed platform, a public cloud and an on-premises cloud. DEEPaaS API and the user interface are common components to both deployment methods. On the one hand, SCAR allows the implementation of the FaaS model in AWS and, on the other hand, OSCAR allows supporting the FaaS model in an on-premises cloud. The following subsections provide further details on the components involved in the scientific gateway.

3.1 | DEEPaaS API

DEEPaaS API[¶] is a software developed in the European DEEP Hybrid-DataCloud[#] project. It is a REST API written in Python which provides simplified access to machine learning, deep learning, and artificial intelligence models. Through HTTP calls, the user has access to the functionalities of the implemented model. The requirements and changes to integrate the applications with the DEEPaaS API are minimal, and this allows an easier interaction with the training and validation functionalities of the models[40].

The inference and training of the models integrated with the DEEPaaS API is done through its REST API. In order to obtain the prediction of the models in the designed platform a new functionality was added to obtain the prediction of the models from the command-line interface. This functionality is a command written in Python where the user specifies certain input values in order to obtain the prediction result through the command-line. This enables support for batch execution of ML models packaged in DEEPaaS API to execute on both high-end HPC supercomputers and batch-based computing installations such as virtual clusters[9].

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

Developing command-line-based tools that connect to a REST API that implement a particular service is a common approach in distributed systems[41]. The functionality incorporated into DEEPaaS API allows extending the field of use to scenarios where both a REST API and the command line can be used.

3.2 | Web-based Scientific Gateway

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

In the web programming environment there are many frameworks and languages that facilitate the work of developers. The development of this web interface is based on the Javascript frameworks VueJS[‖] and Vuetify[**]. VueJS is a popular open-source Javascript front-end framework aimed at organizing and simplifying web development, mainly in the development of user interfaces. The use of components is one of the most powerful features of Vue. In large applications it is more efficient to divide the application into small, autonomous, and often reusable components so that development is more adaptable[42].Vuetify is a component library made for VueJS that makes web interface development easy, with each component designed to be modular, responsive, and high-performance. The web interface is compiled as a static web site that is served from an Amazon S3 *bucket* and made publicly available[††].

Authentication to the web can be done through two methods, as Figure 2 shows: Amazon Cognito and DEEP IAM. Amazon Cognito is a service offered by AWS that allows user registration, login, and access control in web and mobile applications. In this service, through the *User Pools*[43], a group of users is created where access credentials are assigned. DEEP IAM is an identity provider based on the OpenID Connect standard that allows existing users in that community to log in into our service without having to register. Both authentication methods are integrated with Amazon Cognito *Identity Pools*[44] (Federated Identities) to obtain temporary credentials from AWS that allow access to other services such as Amazon S3, AWS Lambda, among others.

Once authenticated, the user can interact with the different integrated models by uploading, downloading, listing, and deleting the files to be processed or those that are the result of the inference phase. In addition, from the gateway itself, it is possible to check the status of the jobs that are being processed, in case they execute for a long amount of time. All this process is possible

[¶] DEEPaaS API - https://github.com/indigo-dc/DEEPaaS
[#] DEEP Hybrid-Datacloud - https://deep-hybrid-datacloud.eu/
[‖] VueJS - https://vuejs.org/
[**] Vuetify - https://vuetifyjs.com/
[††] Web Interface - https://scar-deepaas-ui.grycap.net/

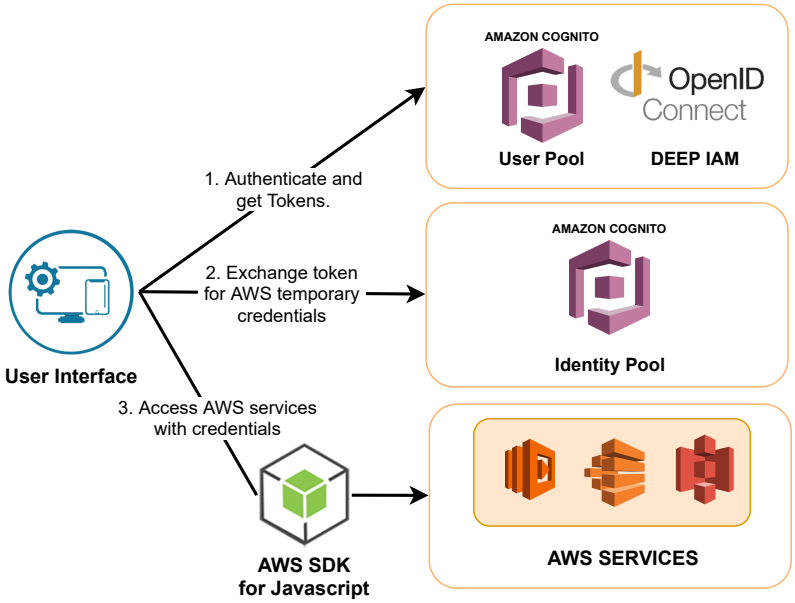

FIGURE 2 High-level authentication and authorization flow in the web interface.

through the OSCAR API and AWS services such as Amazon S3, AWS Lambda, AWS Batch and the use of *AWS SDK for JavaScript* that allows access to AWS services from a web interface.

3.3 | Serverless Frameworks

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

3.3.1 | SCAR (Serverless Container-aware ARchitectures)

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

One of the solutions would be the use of Docker containers, though Docker requires root access for its installation. To solve this limitation, a container engine able to run Docker containers in the user space is needed. In this sense, tools such as *udocker*[45], *Singularity*[46], *CharlieCloud*[47], *Shifter*[48] or *Podman*[49] play a fundamental role since they precisely allow the execution of Docker images in spaces where there are no root privileges.

SCAR[‡‡ 50] is a tool that uses *udocker* to transparently run container out of Docker images in AWS Lambda as event-driven applications, such as in response to uploading a file to a S3 *bucket* or via an HTTP call to API Gateway[§§]. SCAR started to be implemented in 2016 and, in late 2020, AWS Lambda announced native support for running applications packaged on Docker containers. However, this feature does not support images from Docker Hub, the largest repository of Docker images. This shows that as users need more customizable environments, as is the case for scientific applications, cloud providers adapt their services to the new requirements.

The functions in SCAR are created from a YAML[51] file which describes, among other features, the Docker image of the application, the script to be executed in the container, the input and output storage provider, and the execution mode. SCAR allows the implementation of several execution modes: *lambda*, *batch*, and *lambda-batch*. These execution modes determine the service to be executed based on computational requirements.

[‡‡] SCAR - https://github.com/grycap/scar
[§§] API Gateway - https://aws.amazon.com/api-gateway/

In the *lambda* mode, all executions are performed as Lambda function invocations. In addition to the AWS Lambda limitations mentioned above, it is important to add: maximum execution time of 15 minutes, 512MB of ephemeral, potentially shared, storage space, and 10GB of RAM, which affects linearly to the computing capacity. These computing requirements of AWS Lambda led to the emergence of other execution modes in SCAR, as is the case of the integration of AWS Batch into SCAR, as described in the work by Risco et al.[52].

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

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

OSCAR[¶¶] is an open-source platform that deploys and integrates several services in order to support event-driven long-running executions within an elastic Kubernetes cluster, accessed through a web interface[53], REST API or CLI. The graphical interface of OSCAR is a static web site for users to view, create, edit, and delete the functions implemented in the platform. From the web interface itself, you can access the storage system, which allows to download and view the input and output files. It is also possible to check the status of the functions and the logs generated in the execution.

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

A previous research by Naranjo et al.[54] achieved the integration of acceleration devices, such as GPUs, into the OSCAR platform. For this, the rCUDA[## 55 56] tool was used, which allows virtualizing GPU devices that represent physical GPUs in a remote machine. In addition, rCUDA allows the same GPU to be shared by multiple applications accessing them simultaneously. The use of acceleration devices on a serverless platform allows expanding the field of action and inclusion of applications that require intensive computing, such as machine learning and artificial intelligence models.

3.4 | Architecture

Figure 3 shows the proposed architecture for the web-based scientific gateway that supports the inference from machine learning models executed on serverless platforms in multi-clouds.

The integration of the models in AWS is done through SCAR, with functions that are activated once a file is uploaded into the input storage system. In the case of the on-premises cloud, the integration of the models is done through the OSCAR framework. Both deployment methods support three types of storage providers: Amazon S3, MinIO[|||], and EGI DataHub[57]. Amazon S3 is the storage system provided by AWS, MinIO is a server-side storage system compatible with the Amazon S3 API, and EGI DataHub is one of the storage systems supported by the EGI Federated Cloud[58], an IaaS-type cloud made up of on-premises and academic clouds that provide computing resources to the scientific research community.

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

[¶¶] OSCAR - https://github.com/grycap/oscar
[##] rCUDA - http://www.rcuda.net/
[|||] MinIO - https://min.io/

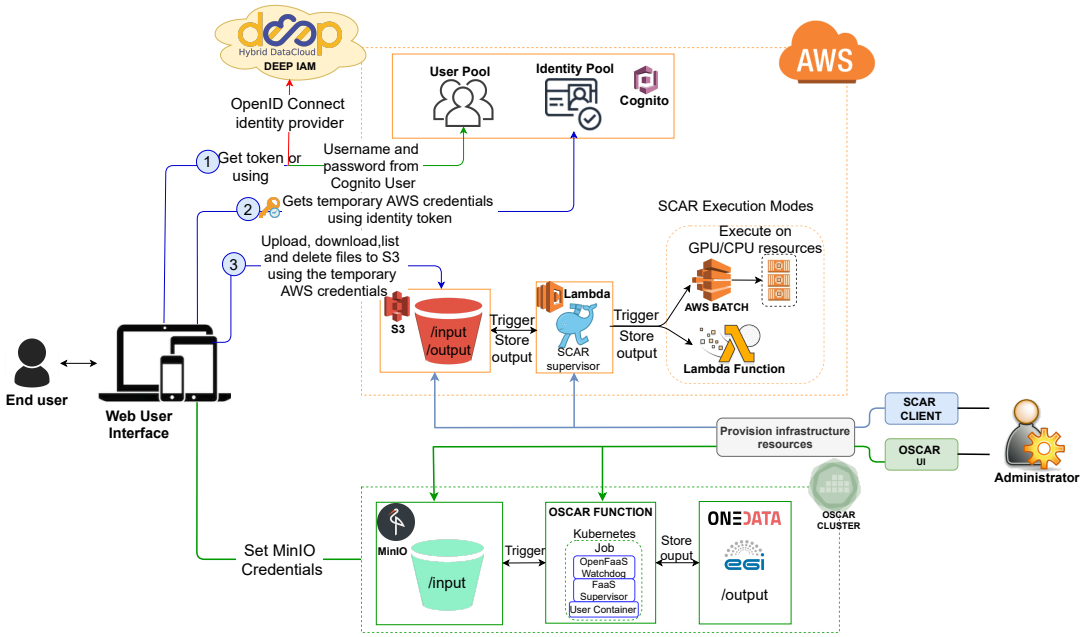

FIGURE 3 Architecture for the integration of Machine Learning models in AWS and an on-premises cloud.

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

4 | USE CASES

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

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

- Audio Classifier[59]: This model allows to perform audio classification with deep learning. It allows to classify through a model previously trained with the AudioSet[60] dataset of 527 high-level classes. To implement the prediction, the model expects as input a URL or an audio file and as output a JSON file with the top 5 predictions is returned.
- Plants species classifier[61]: This model allows to classify plant images among 10 thousand species from the iNaturalist[62] dataset. For the inference phase the model expects as input a URL or an RGB image and returns a JSON file with the top 5 predictions.
- Body Pose Detection[63]: This model allows real-time detection of body poses using deep neural networks. It can be used to estimate single or multiple poses in images or videos. In our case it is used to detect body poses in images. To obtain the prediction the model expects a URL or a RGB image and as a result returns as output the different key points of the body with the corresponding coordinates. This case study obtains an image identifying each of the key points in addition to a JSON file with the result of the classification.

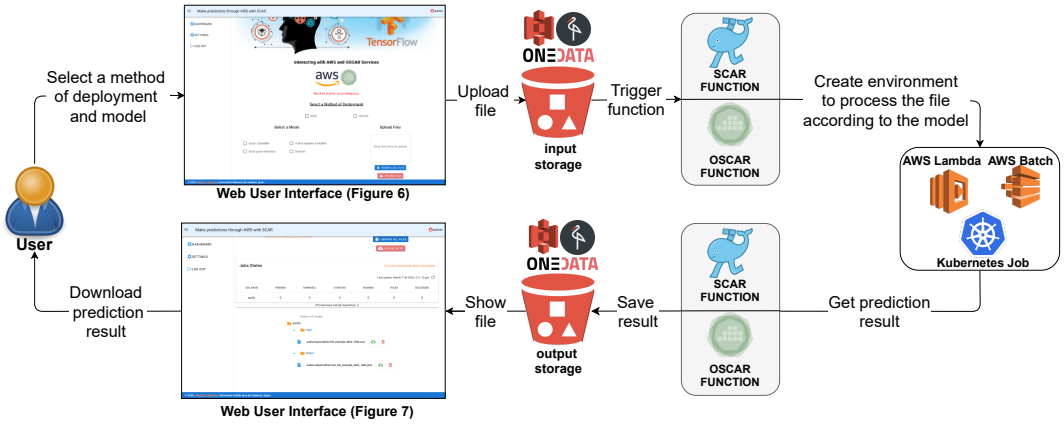

FIGURE 4 Simplified file processing workflow. Functions with different deployment methods selected, either on AWS or in a local cluster with OSCAR.

- Darknet[64]: Darknet is an open-source neural network framework written in C and CUDA that supports CPU and GPU computation. This example uses the YOLO (you only look once) library for real-time object detection, such as people, cars, animals, etc.

Figure 4 shows, in a simplified way, the processing flow of the files. In the following points a more detailed explanation of the process is made from when a file is uploaded until the prediction result is obtained.

- **Authentication:** The first step to access the classification models is to authenticate on the web. To do this, users can authenticate through their Amazon Cognito credentials or through DEEP IAM, if they have credentials from this identity provider.
- **Method of deployment:** Once the authentication process is completed, the user selects one of the available deployment methods, AWS or OSCAR. In case OSCAR is selected, it is necessary to configure the MinIO credentials in the SETTINGS tab, Figure 5.
- **Select Model**: After selecting the deployment method, the user can select one of the available models. When a model is selected, two links of interest to the user are displayed, *Input example for models* which shows an example input file and *Link to the model in the Catalog* link where more information about the model can be obtained. This information allows the user to have specific information about the selected model.
- **Upload Files**: At this point, the user can now upload files from the web interface, in order to trigger the execution of the function corresponding to the selected model, to perform the processing of the file(s). The input and output files are stored in the storage system specified in the function definition. In the storage process, a directory structure has been created where the files are stored in folders named after the selected model and the user, allowing each user to access only her information. This allows the development of a multi-tenant environment and the addition of an activation event for each model independently.
- **Job Status**: The models that are executed in AWS Batch are generally long-running. Therefore, the web interface allows the user to query the status of the jobs that are being processed and, thus, know when the result of the prediction has been obtained.
- **Download Result**: Once the inference process has been performed, the prediction result is stored in the output storage system. In the case of AWS the result is accessible from the web interface and in the case of OSCAR the result is stored either in MinIO or in the EGI DataHub user space, accessible through a link from the web interface.

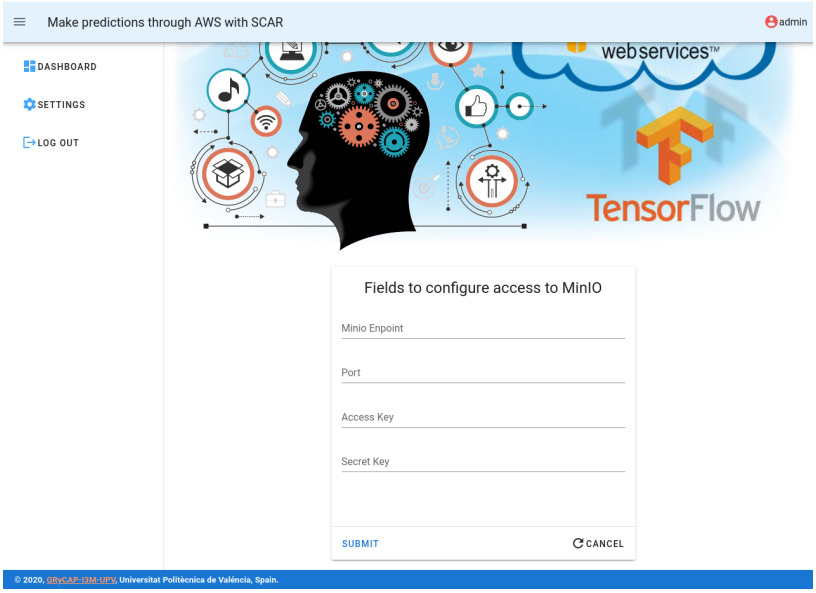

FIGURE 5 Settings tab to configure access to MinIO.

5 | RESULTS

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

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

Figure 6 shows the panel for selecting a deployment method, one of the available models, and the section for uploading the files to the input storage system. Loading the file generates an event that automatically triggers the function corresponding to the selected model. As mentioned before, the web interface automatically creates the directory structure taking into account the selected model and the authenticated user on the web.

From the web interface it is possible to check the status of the jobs running in AWS Batch, as shown in Figure 7. It is important to note that in AWS Batch the deployment of a compute environment can take several minutes because the EC2 instances need to be provisioned and configured; hence the importance of querying the status of the jobs running in this service. The status of the jobs is queried through a Lambda function that communicates with the AWS Batch API, as shown in Figure 8. This function is triggered every time the job status is updated from the web interface. In the case of OSCAR, the job status is queried through its API, for jobs in status: *PENDING*, *RUNNABLE*, *STARTING*, *RUNNING*, *FAILED* and *SUCCEDED*. This process allows the user to monitor the life cycle of long-running jobs.

The prediction result is stored in the output storage system specified in the function definition. Like the input files, the prediction result is stored taking into account the user and the selected model. Figure 7 also shows the section for interacting with the input and output files of the selected deployment method and model. From this section it is possible to download the files or delete them if they are no longer needed. Amazon S3 and EGI DataHub provide high availability, long-term preservation and remote accessibility from anywhere. Alternatively, since MinIO is installed inside the Kubernetes cluster, it provides certain storage capabilities limited to the lifespan of the cluster.

As an example, Figure 9 shows the prediction result for the case of the plant species classifier model. On the left (a) the original image is shown and on the right (b) the result of the prediction in JSON format. Also in (c) an example of the search result of the link indicated in red in (b) is shown.

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

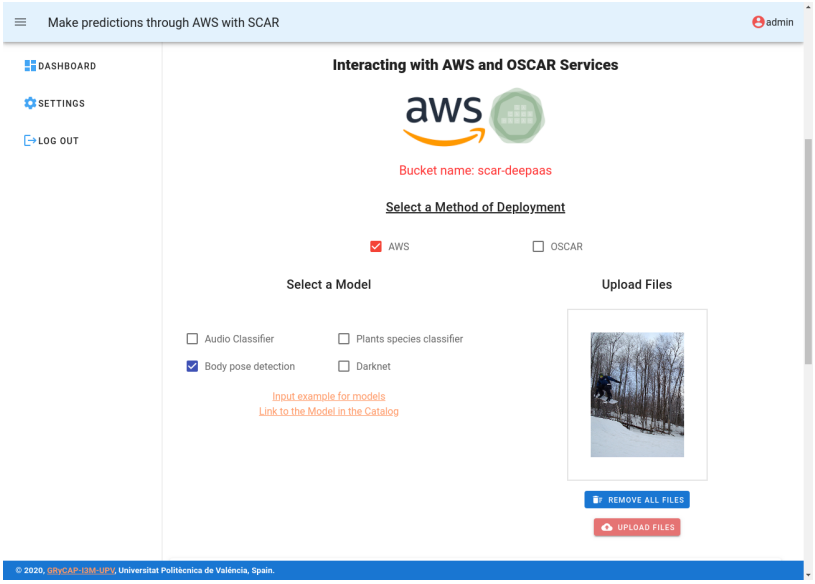

FIGURE 6 *Select Method of Deployment*, *Select Model* and *Upload Files* panels of the Web Interface.

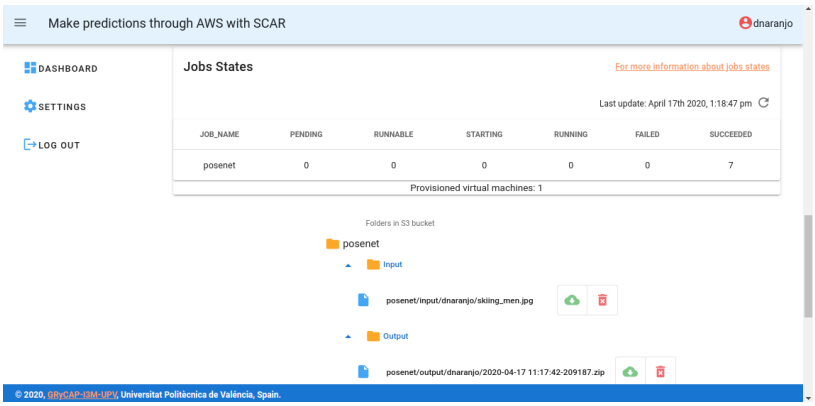

FIGURE 7 Panels to check the status of jobs and stored files.

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

Figure 10 shows the processing times obtained for the case of the Darknet model in AWS Lambda. In this case, it can be seen that the first execution is the one with the longest processing time, because the platform implements scale to zero, which refers to the fact that while the platform is not used, there is no active function, which allows to save costs. Scale to zero introduces the phenomenon of *cold start*, where in the first execution of the function, for SCAR, it is necessary to download the Docker image that contains the application code, start a new execution environment, execute the initialization code and execute the function. Once these steps have been carried out in the first execution, the following ones run faster since the unpacked Docker image may be reused from the ephemeral, potentially shared */tmp* space. It is important to note that cold start can be mitigated by keeping the function always hot at a higher cost. After this first invocation where the function is already initialized, the rest of the executions typically reuse the configuration mentioned above, which causes them to be processed in a similar time, around 13 seconds.

Figure 11 and Table 1 show the same experiment performed in AWS Lambda, but in this case the 10 executions are executed taking into account the plant species classifier model, which has to be executed in AWS Batch (*batch* execution mode). In the case of AWS Batch, the compute environment was defined with a maximum of 2 instances of 1CPU and 4GB of RAM. Jobs are sent simultaneously and queued until the scheduler detects that there are resources available and sends them to be processed. Therefore, the processing time is divided into execution time and waiting time. In the graph, the blue bars represent the processing

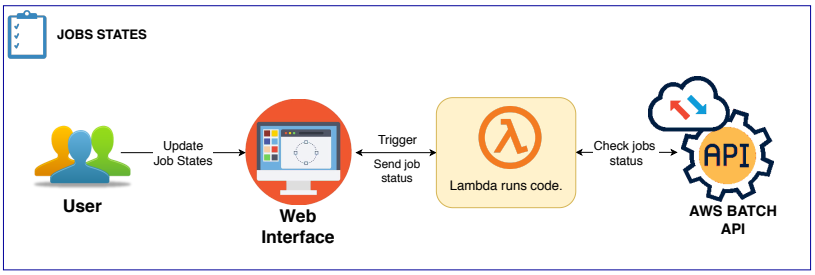

FIGURE 8 Check the status of jobs through a Lambda function.

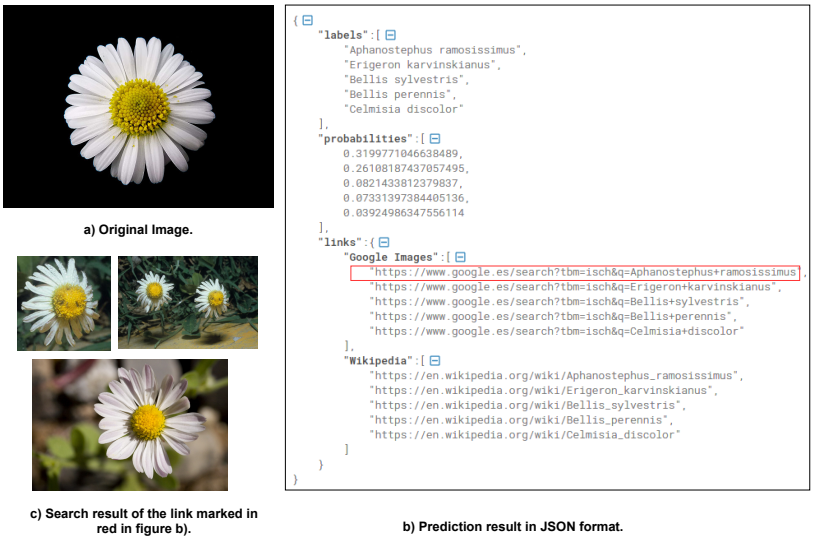

FIGURE 9 Example of the result obtained using the Plant Species Classifier Model. On the left (a) the original image, on the right (b) the result of the prediction in JSON format, and (c) an example of the search result.

time of the file without taking into account the waiting time in the job queue that is displayed in the yellow bars. The processing time is approximately equal to 14 seconds for each of the invocations.

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

Figure 12 and Table 2 show the times obtained for the same experiment performed in AWS Lambda and AWS Batch with the plant species classifier model, but in this case implemented in an elastic Kubernetes cluster with the OSCAR framework. The components used for the deployment of this cluster allow it to grow and decrease according to the number of nodes and the workload. In order to have the same environment configured in AWS Batch, a maximum of 2 nodes with 1CPU and 4GB of memory were defined. By default, only one of the nodes is active at startup, so there is one execution slot.

As in the case of AWS Batch the blue bar corresponds to the processing time and the yellow bar to the waiting time in the job queue. The processing time includes the time to download the input file, the processing of the file and the time to upload the result into the output storage system. In all executions, the processing time (blue bar) is approximately the same, since the images to be processed have the same characteristics.

The deployment of a new node in OSCAR is done through CLUES[65], an open-source modular elasticity system that allows the introduction of horizontal elasticity capabilities (increase/reduce the number of compute nodes) for cluster-based computing. Once CLUES detects that the workload increases, the new node is deployed (it takes 5 minutes approximately to configure the node), to have more resources available for processing the jobs. Once the workload decreases (around 3 minutes later with no

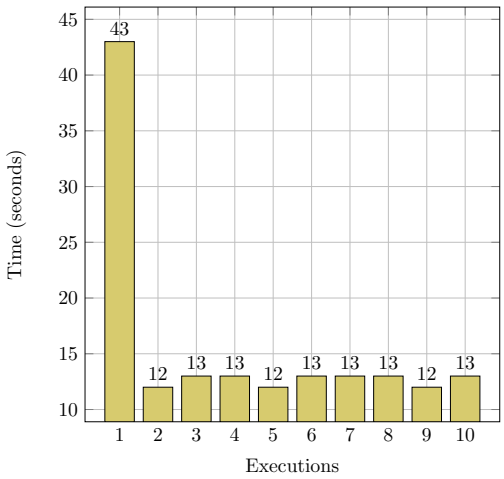

FIGURE 10 Execution times for 10 images with the Darknet model in AWS Lambda.

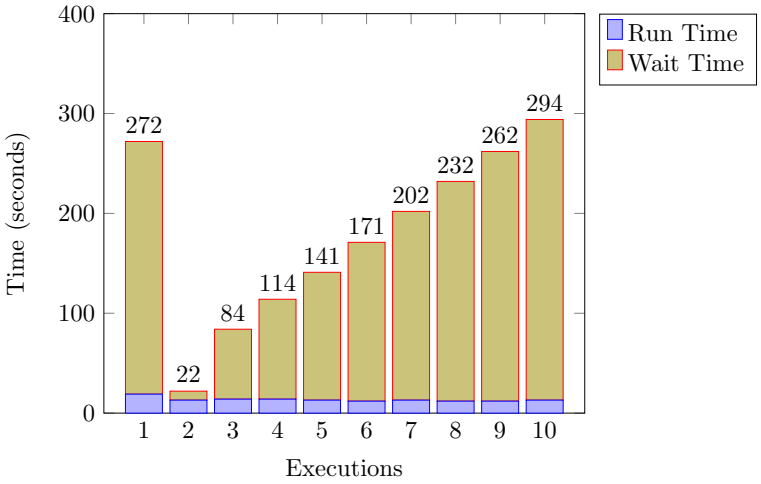

FIGURE 11 Execution times for 10 images with the plant species classifier model in AWS Batch.

workload), the system itself takes care of shutting down the node that is no longer needed, thus saving electricity. In a cluster with OSCAR by default there is always a node ready.

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

The results obtained in each of the environments are different. In the case of AWS Lambda, the effect of the cold start can be seen in the first execution and, from this moment on, the processing time is approximately the same in each of the invocations. In the case of AWS Batch, the first invocation takes longer because there is no compute environment created and it needs to be configured. From this point on the jobs are queued until compute resources are available. The processing time (not including the

TABLE 1 Execution times in AWS Batch, for 10 images using the plant species classifier model.

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

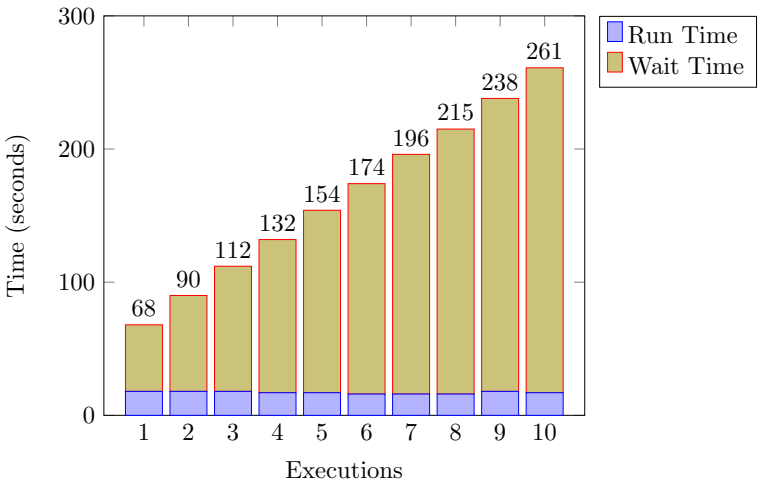

FIGURE 12 Execution times for 10 images with the plant species classifier model in an OSCAR cluster.

time spent waiting in the queue) is approximately the same for each of the executions but AWS Batch provides the ability to run jobs on GPUs. Deploying in an on-premises cloud with OSCAR, saves budget by obtaining execution times similar to public cloud platforms such as AWS and without the restrictions of certain environments such as Lambda. However, the parallelism depends on the underlying computing capacity of said on-premises cloud.

Along with these experiments, an analysis of the costs generated in each of the environments: AWS Lambda, AWS Batch and OSCAR is shown in Table 3. The table refers to all services used in AWS and the on-premises cloud. Amazon CloudWatch is used for monitoring, providing the storage for log files. The prices indicated for Amazon S3 and Amazon CloudWatch are general as they depend largely on the size of the files to be processed and the size of the logs generated. For example, to process 1000 images of an average size of 150KB, which are the ones used in this case study, the cost of Amazon S3 would be $0.0034 per month.

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

Costs in an on-premises cloud are highly dependent on the volume and capacity of the platform. This analysis takes into account the costs generated by electricity and the personnel required for the maintenance of the platform. The infrastructure contemplates two nodes where the storage required for incoming and outgoing files is highly dependent on the expected usage of the platform. Using the Azure Total Cost of Ownership (TCO) Calculator[66] it is possible to determine the cost of an infrastructure

TABLE 2 Execution times in OSCAR, for 10 images using the plant species classifier model.

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

TABLE 3 Cost of the platform in a public (AWS) and an on-premises cloud.

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

deployed in the Azure cloud or on an on-premises platform. In this case we use the values for an on-premises platform, and for a computational environment with the same characteristics as those referenced above and taking as reference the 386 seconds of the maximum execution of the 10 invocations (Figure 12) we obtain a cost of $0.048 per execution. As shown in Table 3 the costs per execution in AWS Lambda are lower, but the limitations introduced by this environment have already been discussed. In the case of AWS Batch there are no limitations as in the case of AWS Lambda, and it also allows the use of acceleration devices like GPUs. In the local cloud we obtain the highest execution costs. However, we can avoid using a commercial public cloud platform if we have access to computational resources such as those provided by EGI Federated Cloud for scientific computing.

6 | DISCUSSION

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

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

The implementation of machine learning and artificial intelligence models on a serverless on-premises platform, with OSCAR, bring some benefits. On the one hand, it avoids vendor lock-in from the use of resources and technologies from public Cloud

providers and, on the other hand, no hardware costs are incurred. The configuration in OSCAR allows infrastructures that grow and shrink elastically, in addition to scaling the worker nodes to zero when not in use, which translates into energy efficiency. Another interesting aspect of OSCAR is the integration with EGI Federated Cloud that provides computational services to researchers under open standards.

The designed platform has included models that are not previously integrated into AWS. Even though they do not comply with the limitations of AWS Lambda, they have been integrated with event-driven scalable services that provide access to computing resources such as GPUs thanks to the *batch* execution mode. The use of the models is done in a simple way from a web interface, so that users can obtain the result of the prediction without requiring previous skills in the use of AWS or machine learning and artificial intelligence models. One of the main advantages of the proposed platform is the scaling to zero that allows you to pay for services only when they are in use, in addition to automatic scaling when demand increases. The solution proposed in this research facilitates the inference of previously trained machine learning models in the public and on-premises clouds at a reduced cost.

7 | CONCLUSIONS

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

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

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

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

8 | ACKNOWLEDGEMENTS

Grant PID2020-113126RB-I00 funded by MCIN/AEI/10.13039/501100011033. This work has also been supported by the project AI-SPRINT "Artificial Intelligence in Secure PRIvacy-preserving computing coNTinuum" that has received funding from the European Union Horizon 2020 research and innovation programme under Grant Agreement No. 101016577. This work was also previously supported by the project DEEP-Hybrid-DataCloud "Designing and Enabling E-infrastructures for intensive Processing in a Hybrid DataCloud" that received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 777435.

References

1. Fox GC, Ishakian V, Muthusamy V, Slominski A. Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research. *arXiv* 2017; abs/1708.08028.

2. Ellis A. OpenFaaS. https://www.openfaas.com/; 2020.

3. Google . Knative. https://github.com/knative/; 2020.

4. Apache . OpenWhisk - Open Source Serverless Cloud Platform. https://openwhisk.apache.org/; 2020.

5. West DM. *The future of work: Robots, AI, and automation*. Brookings Institution Press . 2018.

6. Shafiei H, Khonsari A, Mousavi P. Serverless Computing: A Survey of Opportunities, Challenges and Applications. *arXiv* 2019.

7. Carreira J, Fonseca P, Tumanov A, Zhang A, Katz R. A case for serverless machine learning. In: Workshop on Systems for ML and Open Source Software at NeurIPS. Semantic Scholar; 2018.

8. Naranjo Diana M., Risco Sebastián, Moltó Germán, Blanquer Ignacio . A Serverless Gateway for the Execution of Open Machine Learning Models on AWS. In: Gateways 2020. OSF; 2020.

9. Naranjo Delgado DM. *Serverless Computing Strategies on Cloud Platforms*. PhD thesis. Universitat Politècnica de València, Valencia, Spain; 2021.

10. Spillner J, Mateos C, Monge DA. Faaster, better, cheaper: the prospect of serverless scientific computing and HPC. In: Communications in Computer and Information Science. Springer, Cham; 2018: 154-168

11. Baldini I, Castro P, Chang K, et al. Serverless Computing: Current Trends and Open Problems. *Research Advances in Cloud Computing* 2017: 1–20.

12. Jonas E, Pu Q, Venkataraman S, Stoica I, Recht B. Occupy the cloud: distributed computing for the 99%. In: 2017 Symposium on Cloud Computing - SoCC '17. ACM Press; 2017; New York, New York, USA: 445–451

13. Shankar V, Krauth K, Pu Q, et al. Numpywren: Serverless Linear Algebra. *arXiv* 2018.

14. Eismann S, Scheuner J, Eyk vE, et al. A Review of Serverless Use Cases and their Characteristics. *arXiv* 2020.

15. Jindal A, Gerndt M, Chadha M, Podolskiy V, Chen P. Function delivery network: Extending serverless computing for heterogeneous platforms. *Software: Practice and Experience* 2021: spe.2966. doi: 10.1002/spe.2966

16. Mahmoudi N, Khazaei H. SimFaaS: A Performance Simulator for Serverless Computing Platforms. https://arXiv.org/abs/2102.08904; 2021.

17. Giménez-Alventosa V, Moltó G, Caballer M. A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Generation Computer Systems* 2019. doi: 10.1016/j.future.2019.02.057

18. Opara-Martins J, Sahandi R, Tian F. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing* 2016; 5(1): 4. doi: 10.1186/s13677-016-0054-z

19. Fission. https://fission.io/; 2020.

20. Nuclio. https://nuclio.io/; 2020.

21. Oracle . Fn Project. https://fnproject.io/; 2020.

22. Hendrickson S, Sturdevant S, Harter T, Venkataramani V, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Serverless Computation with openLambda. In: 8th USENIX Conference on Hot Topics in Cloud Computing. USENIX Association; 2016: 33–39

23. Kaviani N, Kalinin D, Maximilien M. Towards serverless as commodity: A case of Knative. In: WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019. Association for Computing Machinery, Inc; 2019: 13–18

24. Palade A, Kazmi A, Clarke S. An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge. In: . 2642-939X. 2019 IEEE World Congress on Services (SERVICES). ; 2019: 206-211

25. Li J, Kulkarni SG, Ramakrishnan KK, Li D. Understanding open source serverless platforms: Design considerations and performance. In: WOSC 2019 - Proceedings of the 2019 5th International Workshop on Serverless Computing, Part of Middleware 2019. Association for Computing Machinery, Inc; 2019: 37–42

26. Benedetti P, Femminella M, Reali G, Steenhaut K. Experimental Analysis of the Application of Serverless Computing to IoT Platforms. *Sensors* 2021; 21(3). doi: 10.3390/s21030928

27. Corral-Plaza D, Boubeta-Puig J, Resinas M. Un Recorrido por los Principales Proveedores de Servicios de Machine Learning y Predicción en la Nube. In: Actas de las XIV Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios (JCIS 2018). ; 2018: 1–10.

28. Ishakian V, Muthusamy V, Slominski A. Serving Deep Learning Models in a Serverless Platform. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). ; 2018: 257-262

29. Rao Divate Kodandarama M, Danish Shaikh M, Patnaik S. SerFer: Serverless Inference of Machine Learning Models. tech. rep., University of Wisconsin; Madison: 2019.

30. Bhattacharjee A, Chhokra AD, Kang Z, Sun H, Gokhale A, Karsai G. BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services. In: 2019 IEEE International Conference on Cloud Engineering (IC2E). ; 2019: 23-33.

31. Christidis A, Davies R, Moschoyiannis S. Serving machine learning workloads in resource constrained environments: A serverless deployment example. In: 2019 IEEE 12th Conference on Service-Oriented Computing and Applications, SOCA 2019. Institute of Electrical and Electronics Engineers Inc.; 2019: 55–63

32. Christidis A, Moschoyiannis S, Hsu CH, Davies R. Enabling Serverless Deployment of Large-Scale AI Workloads. *IEEE Access* 2020; 8: 70150–70161. doi: 10.1109/ACCESS.2020.2985282

33. Kurz MS. Distributed Double Machine Learning with a Serverless Architecture. *arXiv* 2021.

34. Chernozhukov V, Chetverikov D, Demirer M, et al. Double/Debiased Machine Learning for Treatment and Causal Parameters. *arXiv* 2016.

35. Krizhevsky A, Sutskever I, Hinton GE. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 2017; 60(6): 84–90. doi: 10.1145/3065386

36. DEEP Open Catalog. https://marketplace.deep-hybrid-datacloud.eu/; 2020.

37. SCAR: Serverless Container-aware ARchitectures (e.g. Docker in AWS Lambda). https://github.com/grycap/scar; 2020.

38. DEEPaaS: A REST API to serve machine learning and deep learning models. https://github.com/indigo-dc/DEEPaaS; 2020.

39. Web Interface. https://github.com/grycap/scar-deepaas-ui; 2020.

40. López García Á. DEEPaaS API: a REST API for Machine Learning and Deep Learning models. *Journal of Open Source Software* 2019; 4(42): 1517. doi: 10.21105/joss.01517

41. Sarzyniec L, Buchert T, Jeanvoine E, Nussbaum L. Design and Evaluation of a Virtual Experimental Environment for Distributed Systems. In: 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. ; 2013: 172-179

42. Kyoreva K. State of the Art JavaScript Application Development with Vue. js. In: International Conference on Application of Information and Communication. ; 2017: 567–572.

43. AWS . Amazon Cognito User Pools. https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html; 2020.

44. AWS . Amazon Cognito Idenity Pools. https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-identity.html; 2020.

45. Gomes J, Bagnaschi E, Campos I, et al. Enabling rootless Linux Containers in multi-user environments: The udocker tool. *Computer Physics Communications* 2018; 232: 84 - 97. doi: https://doi.org/10.1016/j.cpc.2018.05.021

46. Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 2017; 12(5): e0177459. doi: 10.1371/journal.pone.0177459

47. Priedhorsky R, Randles T. Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC. In: SC '17. International Conference for High Performance Computing, Networking, Storage and Analysis. Association for Computing Machinery; 2017; New York, NY, USA

48. Jacobsen DM, Canon RS. Contain This, Unleashing Docker for HPC. tech. rep., NERSC; United States: 2015.

49. Gantikow H, Walter S, Reich C. Rootless Containers with Podman for HPC. In: Jagode H, Anzt H, Juckeland G, Ltaief H., eds. *High Performance Computing*High Performance Computing. Springer International Publishing; 2020; Cham: 343–354.

50. Pérez A, Moltó G, Caballer M, Calatrava A. Serverless computing for container-based architectures. *Future Generation Computer Systems* 2018; 83: 50–59. doi: 10.1016/j.future.2018.01.022

51. YAML. https://yaml.org/; 2020.

52. Risco S, Moltó G. GPU-Enabled Serverless Workflows for Efficient Multimedia Processing. *Applied Sciences* 2021; 11(4). doi: 10.3390/app11041438

53. Pérez A, Risco S, Naranjo DM, Caballer M, Moltó G. Serverless Computing for Event-Driven Data Processing Applications. In: 2019 IEEE International Conference on Cloud Computing (CLOUD 2019). ; 2019.

54. Naranjo DM, Risco S, de Alfonso C, Pérez A, Blanquer I, Moltó G. Accelerated serverless computing based on GPU virtualization. *Journal of Parallel and Distributed Computing* 2020; 139: 32-42. doi: https://doi.org/10.1016/j.jpdc.2020.01.004

55. Reaño C, Silla F, Shainer G, Schultz S. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In: 16th International Middleware Conference on ZZZ - Middleware Industry '15. ACM Press; 2015; New York, New York, USA: 1–7

56. Reano C, Silla F. A Performance Comparison of CUDA Remote GPU Virtualization Frameworks. In: 2015 IEEE International Conference on Cluster Computing. IEEE; 2015: 488–489

57. EGI DataHub. https://www.egi.eu/services/datahub/; 2020.

58. Integration with the EGI Federated Cloud - oscar documentation. https://o-scar.readthedocs.io/en/latest/egi-integration.html; 2020.

59. DEEP Open Catalog - Model "Train an audio classifier". https://marketplace.deep-hybrid-datacloud.eu/modules/deep-oc-audio-classification-tf.html; 2020.

60. Audioset. https://research.google.com/audioset/; 2020.

61. DEEP Open Catalog - Model "Plants species classifier". https://marketplace.deep-hybrid-datacloud.eu/modules/deep-oc-plants-classification-tf.html; 2020.

62. A Community for Naturalists · iNaturalist. https://www.inaturalist.org/; 2020.

63. DEEP Open Catalog - Model "Body Pose Detection". https://marketplace.deep-hybrid-datacloud.eu/modules/deep-oc-posenet-tf.html; 2020.

64. Darknet. https://pjreddie.com/darknet/yolo/; 2020.

65. CLUES - cluster energy saving (for hpc and cloud computing). https://www.grycap.upv.es/clues/es/index.ph; 2020.

66. Total Cost of Ownership (TCO) Calculator | Microsoft Azure. https://azure.microsoft.com/en-us/pricing/tco/calculator/; 2020.