Rescheduling Serverless Workloads Across the Cloud-to-Edge Continuum

Sebastián Risco, Caterina Alarcón, Sergio Langarita, Miguel Caballer, Germán Moltó

Instituto de Instrumentación para Imagen Molecular (I3M)
Centro mixto CSIC - Universitat Politècnica de València
Camino de Vera s/n, 46022, Valencia, España
Tel.: +34-963877356

Abstract

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

This paper presents two strategies to mitigate the overload that disparate data ingestion rates may cause in low-powered devices at the Edge or Fog layers. To this end, it is proposed to delegate and reschedule serverless jobs between the different layers of the cloud-to-edge continuum using an open-source platform for event-driven file processing. To demonstrate the performance of these strategies, a use case for fire detection is proposed that includes processing in the Fog via minified Kubernetes clusters located near the Edge, in the private Cloud via on-premises elastic clusters and, finally, in the public Cloud by using the AWS (Amazon Web Services) Lambda FaaS service. The results indicate that these strategies can mitigate overloads in use cases involving processing across the cloud-to-edge continuum by coordinating several layers of computing resources.

Keywords: Cloud Computing, Cloud-to-Edge Continuum, Containers, FaaS, Kubernetes, Serverless Computing

1. Introduction

The cloud-to-edge continuum (or computing continuum) [1] encompasses a wide variety of components that may include low-powered devices with limited computer resources, on-premises servers with moderate resources, expensive high-performance computers and public cloud platforms. This is in line with the definition by the OpenFog Reference Architecture for Fog Computing, stating that it is a system-level architecture that distributes computing, storage, control and networking functions closer to the users along a continuum [2].

Indeed, the SPEC-RG reference architecture for the edge continuum [3] proposes an architecture for task offloading according to five computing models: Mist computing, edge computing, multi-access edge computing, fog computing and mobile cloud computing. Mist computing is sometimes used interchangeably with fog computing, even if some authors point to subtle differences [4]. This distributed computing paradigm extends cloud computing capacities into the edge of the network to bring computation closer to the data source and the end devices such as sensors and other IoT (Internet of Things) devices [5]. In this paradigm, the edge devices collect data that is locally processed at the edge of the network to the extent that it is possible due to the computing capacity constraints of such devices. Workload is offloaded into the Cloud when additional computing power is required, thus effectively using the cloud-to-edge continuum. This approach offers several benefits:

- Reduced latency: By processing data locally, mist computing reduces the time it takes to transmit data to the cloud and receive a response. This is particularly important for real-time applications that require immediate decision-making.
- Bandwidth optimization: Sending large volumes of data to the cloud can strain network bandwidth. Mist computing filters and processes data locally, reducing the amount of data that needs to be transmitted to the cloud. Only relevant or summarized data is sent, optimizing bandwidth usage.
- Enhanced privacy and security: Some applications, such as those involving sensitive data or strict privacy requirements, can benefit from keeping data locally and reducing the need for data transfer over public networks. Mist computing allows sensitive data to be processed and analyzed closer to its source, improving privacy and security.

Email addresses: srisco@i3m.upv.es (Sebastián Risco), calarcon@i3m.upv.es (Caterina Alarcón), slangarita@i3m.upv.es (Sergio Langarita), micafer1@upv.es (Miguel Caballer), gmolto@dsic.upv.es (Germán Moltó)

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

However, the execution along the cloud-to-edge continuum involves several challenges that need to be addressed, as identified by the work of Mouradian et al. [6]. This work highlights "task scheduling" and "offloading and load redistribution" as key features for computing in scenarios related to fog computing.

In this scenario, serverless has risen in recent years as an event-driven computing paradigm involving services where the service provider manages the underlying computational infrastructure entirely. This has paved the way for the surge of open-source serverless platforms to be deployed on on-premises resources that mimic this abstraction layer for the developers. These typically involve Container Orchestration Platforms, such as Kubernetes, which provide seamless resource allocation. This is the case of KNative [7], OpenFaaS [8] and, as addressed in this paper, OSCAR [9]. These platforms provide the required abstractions to execute functions or applications, packaged as Docker images, with dynamic provisioning of resources.

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

Second, an implementation of the proposed approach is done in the OSCAR[1] open-source serverless platform, together with an assessment of the functionality on a realistic use case on wildfire detection. To the best of the authors' knowledge, this provides the first implementation of a job rescheduling system for serverless computing across the cloud-to-edge continuum, provided as a ready-to-use implementation in an existing open-source framework.

The remainder of the paper is structured as follows. First, section 2 discusses the related works. Next, section 3 introduces an architecture to support job delegation and rescheduling across event-driven serverless platforms. Later, section 4 introduces a use case on serverless fire detection along the cloud-to-edge continuum to assess the benefits of the proposed approach. Finally, section 5 summarizes the main achievements and discusses future work.

2. Related Work

Several works in the state-of-the-art focus on the scheduling of serverless workloads. For example, the work by Zhang et al. [10] introduces the cost of execution as a requirement for scheduling serverless analytics tasks. They introduce a task scheduler that minimizes execution cost while being Pareto-optimal between cost and job completion time.

Kaffes et al. [11] discuss the limitations of existing scheduling mechanisms for serverless platforms when considering the diverse requirements of applications in terms of burstiness, different execution times and statelessness. They propose a centralized and core-granular scheduler for serverless functions with a global view of the cluster resources.

The usage of serverless computing along the cloud-to-edge continuum has also increased recently. This way, Rausch et al. [12] proposed a serverless platform for building and deploying edge AI applications, thus integrating concepts from AI lifecycle management into the serverless computing model. Based on OpenWhisk composer for workflow composition, they unveiled the lack of support for ARM-based architectures for OpenWhisk.

The cloud-to-edge continuum embraces a diverse plethora of heterogeneous platforms and computer architectures. In this regard, the work by Jindal et al. [13] introduces an extension of the FaaS (Function as a Service) computing model to heterogeneous clusters and to support heterogeneous functions via a network of distributed heterogeneous platforms (Function Delivery Networks). They focus on SLO (Service Level Objective) requirements and energy efficiency, deploying functions on Edge platforms to reduce overall energy consumption. The authors use OpenWhisk, OpenFaaS and Google Cloud Functions.

Sicari et al. [14] build on the concept of scientific workflows using the FaaS computational paradigm to create Serverless workflow-based applications based on a customized Domain-specific Language (DSL) to federate the Cloud-Fog-Edge layers to profit from each computing tier. This is exemplified in the open-source OpenWolf platform, a serverless workflow engine for native cloud-to-edge continuum, based on OpenFaaS, for function execution and Redis to store the workflow manifests and the execution information for the workflows.

Smirnov et al. [15] introduce Apollo, an orchestration framework for serverless function compositions that can run across the cloud-to-edge continuum. The framework leverages data locality to perform cost and performance optimization. It also includes a decentralized orchestration approach where multiple instances can cooperatively orchestrate the application while balancing the workload between the spare resources.

The work by Ferry et al. [16] introduce the SERVERLEss4I0T platform to perform the deployment and maintenance of applications over the cloud-to-edge continuum, but no open-source software is provided.

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

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

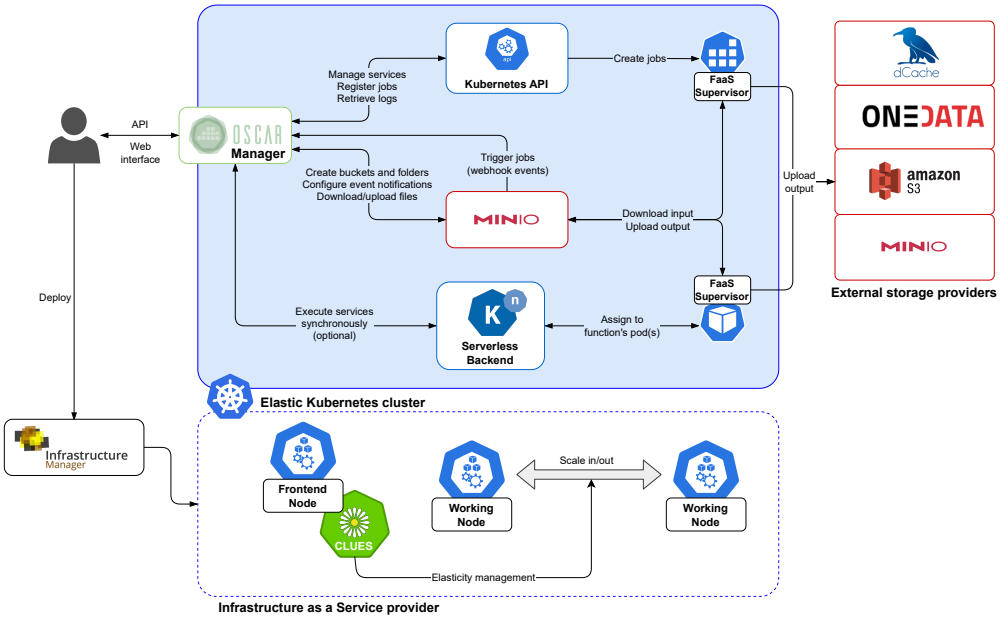

Figure 1: Overall architecture of the OSCAR serverless platform

3. Proposed Architecture

The work carried out is focused on the extension of the OSCAR [9, 17] platform, an open-source[2] framework for serverless data processing through container-based applications. OSCAR is a cloud-native framework that runs on the Kubernetes [18] container orchestration system to define serverless services for data processing. As shown in Figure 1, it allows the scheduling of Kubernetes jobs for the asynchronous processing of files uploaded to a predefined bucket of the MinIO [19] storage system. These jobs are executed as containers, created out of user-defined Docker images, that run on an elastic Kubernetes cluster that can grow and shrink in terms of the number of nodes depending on the current workload and the limits defined at deployment time, thanks to the CLUES[3] elasticity system. Output files are likewise uploaded to MinIO so users can easily retrieve them or to any supported data storage systems such as Amazon S3, Onedata or dCache.

OSCAR also supports the synchronous processing of invocations performed via HTTP requests. For this purpose, the platform is integrated with the Knative [7] Serving framework. However, this study focuses on the asynchronous feature of OSCAR, considering that it is more appropriate for compute-intensive batch tasks, such as inference processes using Artificial Intelligence / Machine Learning (AI/ML) models, as is the use case described in section 4.

OSCAR allows the definition of services via a web-based interface or through the Functions Definition Language (FDL)[4] files using the command-line interface. An OSCAR service is mainly characterized by:

- a Docker image available in a container image registry (e.g. Docker Hub or GitHub Container Registry)
- A shell script that will be executed inside the container created out of the Docker image to perform the data processing on the customized execution environment provided by the Docker image.
- A set of computing requirements for vCPUs, RAM and GPUs.
- An input storage bucket that will trigger the execution of the OSCAR service and one or more output storage backends on which the output data generated by the service will be stored.

These services can be run on an OSCAR cluster or in AWS Lambda via our development SCAR[5] [20]. AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS) to support the Functions as a Service (FaaS) computing paradigm. It allows users to run code in response to certain events (file upload, HTTP request, etc.) without provisioning or managing servers, which is the responsibility of AWS. Its highly elastic features (up to 3000 parallel invocations) and fine-grained billing model (in 1 ms blocks) turned AWS Lambda into a popular option for developing microservices-based architectures. In turn, SCAR is an open-source tool that pioneered in 2017 the deployment of container-based applications in AWS Lambda when this service still had no native container support (introduced in late 2020). SCAR facilitates the execution of general-purpose applications in AWS Lambda, and it provides an automated delegation of jobs into AWS Batch,

[2]OSCAR's GitHub repository: `https://github.com/grycap/oscar`
[3]CLUES - `https://github.com/grycap/clues`
[4]FDL - `https://docs.oscar.grycap.net/fdl/`
[5]SCAR - `http://github.com/grycap/scar`

a managed service to provide automated elastic compute clusters as a service. This allows the use of AWS Lambda to execute spiky bursts of short jobs with moderated computing requirements (AWS Lambda invocations cannot run beyond 15 minutes or use more than 10 GiB of RAM) while delegating into AWS Batch jobs that require larger memory or specialized hardware, such as GPUs.

The advantage of using a common Functions Definition Language is the ability to compose serverless workflows across the different layers of the cloud-to-edge continuum. For example, as described in our previous work by Risco et al. [21], workflows can be composed by services defined on OSCAR platforms configured on lightweight clusters (i.e. on ARM-based devices such as Raspberry Pi) located on the Edge or Fog, on OSCAR clusters in on-premises clouds or Lambda functions in the public Cloud.

The main benefit of OSCAR is the ability to provide scalable event-driven computations upon file uploads to an object storage (or an HTTP-based invocation). OSCAR can run on multiple computer architectures (*amd64* and *arm64*) and container-based platforms (Kubernetes, K3s). It is also, integrated with SCAR for highly scalable cloud bursting into AWS Lambda. Therefore, for this reasons, it can be used to support serverless event-driven computing along the continuum and it has been the selected platform on which to develop our contributions. Further information about OSCAR is available in the work by Pérez et al. [17] .

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

Indeed, replication and distribution are features required to achieve high availability in a distributed system. Applying this approach in the cloud-to-edge continuum allows the use of resources from disparate computing infrastructures, coordinated by a distributed control plane that mediates access and resource distribution. Therefore, we introduce the ability to create replicas of serverless services for this work. An OSCAR cluster has the OSCAR Manager component (shown in Figure 1), which provides the entry point to trigger the execution of an OSCAR service. The cluster can be deployed on various computing infrastructures supported, such as Raspberry Pis, IaaS Clouds and public Clouds. The dynamic deployment on multiple Clouds is achieved thanks to the Infrastructure Manager (IM)[6] [22], an open-source[7] Infrastructure as Code (IaC) tool to provision and configure virtualized computing resources from multiple cloud back-ends. The dynamic deployment support of OSCAR clusters via the Infrastructure Manager allows users to self-deploy them on their preferred Cloud, where the user-defined OSCAR services are deployed to be triggered for scalable data-driven processing.

[6]Infrastructure Manager (IM) - `https://im.egi.eu`

[7]IM's GitHub repository - `https://github.com/grycap/im`

An OSCAR service can have multiple replicas, each one potentially running on a different cluster with a similar configuration (but each service replica can specify a different number of computational resources). Each file upload to MinIO, or an asynchronous invocation to its REST API, triggers the creation of a job that is executed on the scalable Kubernetes cluster, which grows and shrinks depending on the number of jobs. In this scenario, it is important to support efficient strategies to distribute the workload among the available OSCAR service replicas to reduce the execution time.

To this end, two strategies are proposed to reschedule jobs among OSCAR service replicas: *Resource Manager*, described in section 3.1, and *Rescheduler*, described in section 3.2. Furthermore, section 3.3 defines the extension of the Functions Definition Language (FDL) used in SCAR and OSCAR to support this new functionality, as well as details the mechanism for delegating the events that trigger the execution of the jobs.

3.1. Resource Manager

Given the capabilities for resource discovery on the nodes of a Kubernetes cluster, a resource manager has been implemented in OSCAR to bypass job scheduling on a cluster that does not have available resources.

For this purpose, the Kubernetes core API is used to obtain the status and resources available of all the active working nodes. If the resources available on a working node exceed those requested by an OSCAR service execution, the incoming job can be scheduled on the node. The availability of a working node to be scheduled is checked on a regular basis according to the periodicity specified on the environment variable *RESOURCE_MANAGER_INTERVAL*, configurable by the user. As shown in Figure 2, and highlighted by a dotted box, the lifecycle of the Resource Manager consists of periodically checking through the K8s API the available resources of each working node and caching them for the job handler to query.

In turn, the job handler receives an event from a file upload on a MinIO bucket and checks the availability of resources. If there are no available resources in any of the working nodes of the cluster and the OSCAR service has a replica defined in its specification, it will delegate the event to the replica. The job handler will schedule the job in the current cluster only if resources are available.

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

3.2. Rescheduler

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

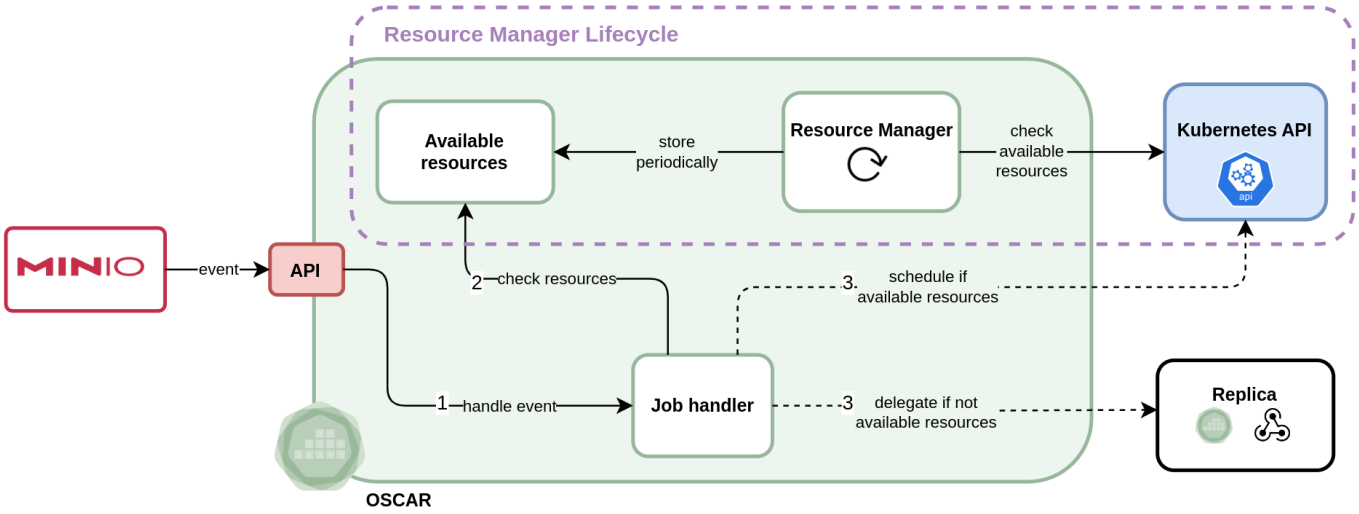

Figure 2: Simplified diagram of the Resource Manager component.

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

Each OSCAR service can have its own *threshold*, which defines the maximum amount of time (in seconds) that a Kubernetes job from an invocation of an OSCAR service with replicas can be queued before delegating it. Therefore, the scheduled jobs are filtered by a label containing this information. Also, to figure out to which OSCAR cluster each job needs to be delegated, the jobs are filtered by another label that provides the service name.

Figure 3 shows how the Rescheduler periodically checks the cluster's pending jobs that exceed the defined threshold. This interval is configurable through the `RESCHEDULER_INTERVAL` environment variable. It has a default value per cluster through the `RESCHEDULER_THRESHOLD` environment variable. However, as mentioned before, and detailed in Figure 4, it can be configured for each service via the `rescheduler_threshold` parameter in the FDL.

Jobs that exceed the defined threshold will be automatically delegated to a replica by the Rescheduler and, once scheduling is achieved on the replica, will be removed from the current cluster queue.

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

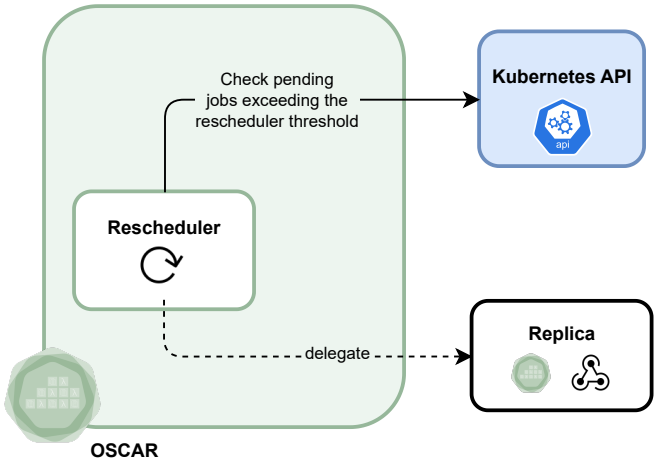

Figure 3: Simplified diagram of the Rescheduler component.

3.3. Delegation Mechanism

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

As shown in Figure 4, two different types of replicas can be specified. On the one hand, the "oscar" type of replicas are services defined in another OSCAR cluster. This requires to

indicate the cluster identifier (`cluster_id` parameter) where such service is deployed, as well as its name. The OSCAR command-line interface (CLI)[8] automatically embeds the access credentials to the clusters of the replicas in the configuration of the services so that users do not have to worry about managing them. On the other hand, the "endpoint" type of replicas support the delegation of events to HTTP endpoints, which will be sent via POST requests. Support for these endpoints makes it possible to use any FaaS service (such as AWS Lambda) where function invocation via REST APIs can be enabled. Thanks to this support, jobs can be rescheduled between OSCAR clusters, which can run on the edge, on-premises and public Clouds, and self-managed services in the public Cloud such as AWS Lambda functions, which can be exposed via HTTP APIs, using *function URLs* or via API Gateway, as done with the SCAR framework.

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

Algorithm 1 Delegation algorithm pseudocode.

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

Regarding security, all jobs delegated to other OSCAR clusters are performed using authorisation tokens obtained from the OSCAR configuration API via the *basic auth* credentials embedded in the services configuration. Moreover, different authorisation mechanisms can be provided thanks to the support of user-defined custom headers in the "endpoint" replica type. In addition, all invocations support the HTTPS protocol, so the traffic between the client and server will be encrypted.

[8]OSCAR CLI - `https://github.com/grycap/oscar-cli`

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

Figure 4: Support for replicas in the Functions Definition Language file.

Notice that this approach takes into account the peculiarities of event-driven serverless systems regarding the job delegation across replicas to avoid unnecessary data transfers and the ability to invoke remote HTTP endpoints as the entry point for public serverless services.

To assess the benefits of this approach for automated serverless workload redistribution along the cloud-to-edge continuum, we carried out the use case described in the next section.

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

Increased wildfires due to rising temperatures are one of the most alarming impacts of global warming [23]. Detecting fires in their early stages is essential to act quickly and minimise

the damage caused to forests. However, it is not easy to anticipate these events. While they often correlate with several meteorological factors, external factors can also provoke them. Surveillance data analysis is an active field of research to prevent this type of situation. Advances in image processing and artificial intelligence enable the development of models capable of detecting fires from images taken from surveillance systems.

This section proposes a use case for processing surveillance images across the cloud-to-edge continuum. For this purpose, an architecture is presented in which the data capture devices would be located at the Edge. These devices would be composed of thermal sensors capable of analysing different meteorological metrics such as temperature or relative humidity and cameras capable of obtaining images periodically. The information obtained by the thermal sensors will be used to detect the level of fire risk at a given time, thus increasing or decreasing the rate of obtaining the images to be processed. To process the images, Minified Kubernetes clusters (using the *k3s* [24] distribution) composed of Raspberry Pis located in the Fog, i.e. near the capture devices, will be used. Each cluster will be in charge of processing images from several cameras. In the experiment described in section 4.1, a cluster in the Fog will process images from three cameras. Moreover, the Amazon SNS service will notify the firefighters in case of fire detection (see Figure 5).

4.1. Case Study Design

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

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

- *Fog cluster:* composed of four Raspberry Pi 4 model B, each with 4GB of RAM and a Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz. The Kubernetes minified distribution *k3s* has been used to deploy the components, running one node as the frontend, with the remaining three Raspberry Pi set as working nodes.

- *On-premises cluster:* deployed on an OpenStack-based Cloud, whose underlying infrastructure is composed of 14 Intel Skylake Gold 6130 processors, with 14 cores each, 5.25 TB of RAM and 2 x 10GbE ports and 1 Infiniband port in each node. The virtualized Kubernetes-based OSCAR cluster is configured with one frontend node and one working node with eight vCPUs and 32 GB of RAM each, dynamically deployed and configured using the Infrastructure Manager (IM).

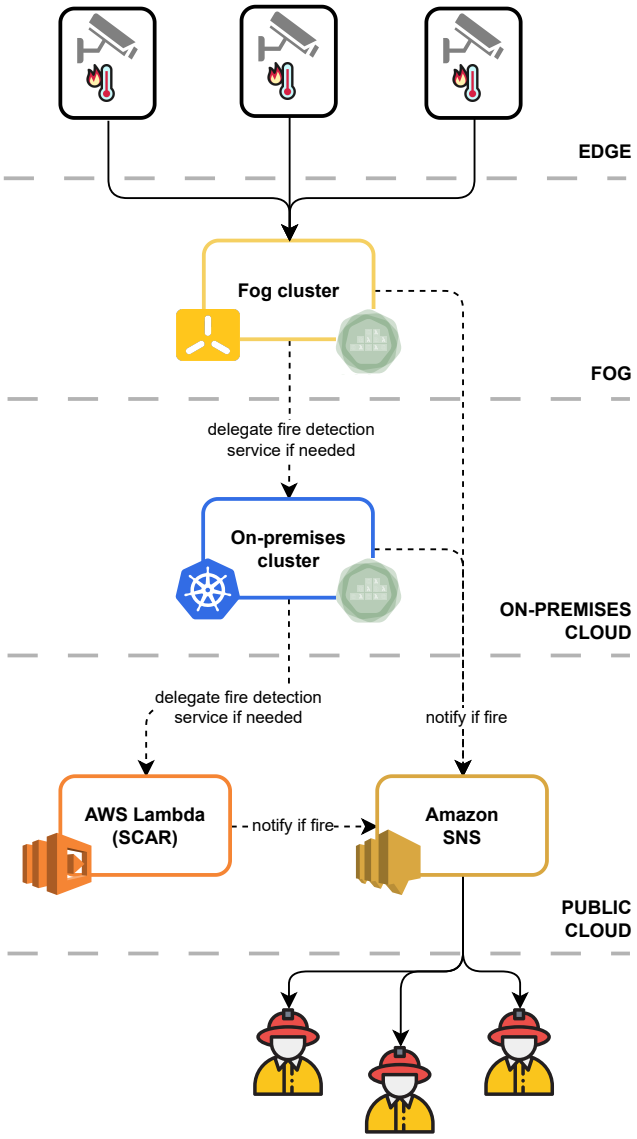

Figure 5: Use case architecture for fire detection across the cloud-to-edge continuum.

The fire detection service is based on the application[9] from the study conducted by Thompson et al. [25], in which a compact convolutional neural network model for non-temporal real-time fire detection was developed and trained. The implementation consists of a simplified ShuffleNetV2 architecture for full-frame binary fire detection and an in-frame classification using

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

superpixel segmentation. The application has been modified to provide a text file with the words "FIRE" or "NOT FIRE" as output. Meanwhile, the script employed for the service generates a compressed (*zip*) file with the text file and the image of the superpixel segmentation.

Notifications, when a fire is detected, are sent via the Amazon SNS service [26], whose SDK (Software Development Kit) client has been included in the software container built for the service. AWS credentials can be specified in the services' definition so that notifications can be sent regardless of the cluster in which they are deployed.

Figure 4 shows the definition of the OSCAR services in FDL. After profiling the application, the OSCAR services were configured to 1 CPU and 1 GB of RAM for the jobs created when the service is invoked, both in the Fog and On-premises clusters. Therefore, the number of jobs that can be executed concurrently will be 9 in the Fog cluster and 7 in the On-premises cluster, since the services involved in the OSCAR control plane also use RAM from the underlying virtual infrastructure.

Data ingestion was initially designed using Apache NiFi, a scalable tool for directed graphs of data routing, transformation, and system mediation logic, by creating a dataflow that controls the data ingestion into a MinIO bucket to trigger the OSCAR service. Since NiFi has no available processors to take pictures from the webcam, the GetWebCamera plugin was included[10]. However, we found limitations in the data capture rate of this plugin. Therefore, we decided on the use case to emulate the data ingestion through a Python script that reproduces all the data flow. It gets the image from the virtual web camera and uploads it into the MinIO bucket. The ingestion rate has two phases with a duration of 30 minutes. The first phase ingests three images every 30 seconds. The second one has an ingestion rate of three images every 5 seconds.

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

- *Scenario 1:* There is the Fog cluster, to which the images that trigger the execution of the fire detection service are uploaded, and the On-premises cluster with the service configured as a replica. When the image ingestion rate increases, the Fog cluster will be overloaded and jobs will be delegated to the On-premises cluster. This scenario has been designed to exemplify the use case using on-premises resources, except the SNS service for fire notifications, so there is no need to rely on public Cloud serverless platforms (such as AWS Lambda).
- *Scenario 2:* Same as the previous scenario but with the addition of a replica deployed as a function in AWS Lambda created through SCAR. The function has been made accessible via HTTP requests through the API Gateway service. Therefore, the FDL specifies an additional replica of type "endpoint" with 1 GB of RAM. This scenario has been developed to demonstrate how delegating jobs to higher levels of the cloud-to-edge continuum can be appropriate to profit from the scalability of managed serverless services, especially in time-constrained use cases.

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

4.2. Results and Discussions

This section presents the results obtained after conducting the previously described experiment for the two proposed scenarios. After running the experiment in both scenarios, the average processing time of the fire detection jobs on the three platforms used, i.e. Fog cluster, on-premises cluster and AWS Lambda, has been analysed. Figure 6 shows that the Fog cluster is noticeably slower than the other platforms due to the lower computational capacity of the cluster's lightweight devices (Raspberry Pis). Meanwhile, the on-premises cluster is the one that has offered the best performance, followed by AWS Lambda, in which the infrastructure is abstracted from the users, so it is not possible to know precisely the instance type used. AWS Lambda allocates computational power (e.g. CPU) proportionally to the amount of memory allocated (up to 10 GBs). For the sake of cost-effectiveness, the memory allocated to the Lambda function was only 1 GB, thus resulting in lower performance when compared to the execution in the on-premises cluster.

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

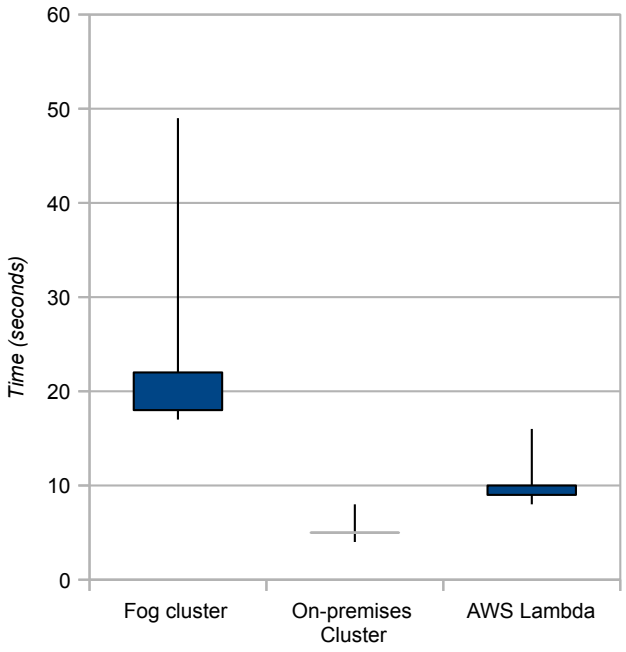

Figure 6: Average execution time of the fire detection service on the three platforms employed.

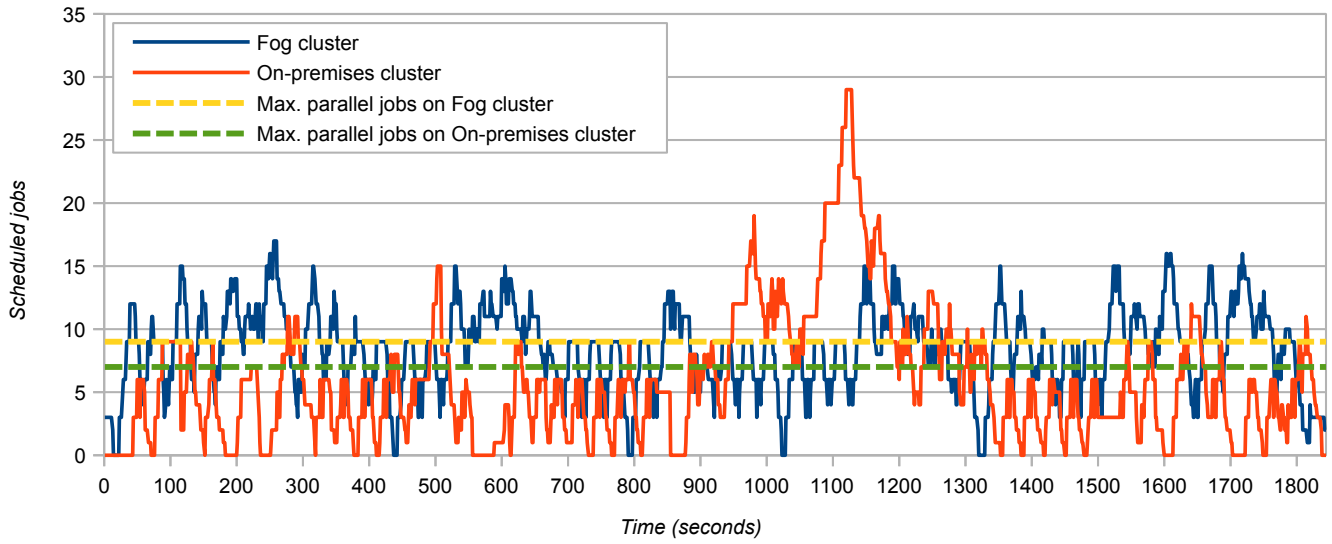

Figure 7: Number of scheduled jobs on the fog and the on-premises cluster, and the maximum number of jobs each cluster can execute simultaneously.

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

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

- In the scenario 1, a total of 477 jobs have been delegated from the Fog cluster to the on-premises cluster, 455 of them delegated via the Resource Manager and 22 via the Rescheduler. Figure 7 details the job scheduling of the second image ingestion phase for the first scenario. As can be seen, load peaks appear when the clusters become saturated. These spikes displayed above the lines of maximum parallel jobs for each cluster mean that the jobs cannot be processed and are kept in the queue until free resources are available. The peak that occurs at approximately the 1090th second in the on-premises cluster is worth mentioning, in which the cluster is fully saturated as many jobs are scheduled.
- In the scenario 2, 538 jobs have been delegated from the Fog cluster to the on-premises cluster, 510 delegated by the Resource Manager and 28 by the Rescheduler. Likewise, the on-premises cluster has delegated 85 jobs to AWS Lambda, 76 by the Resource Manager and 9 by the Rescheduler. As seen in Figure 8, thanks to the delegation from the on-premises cluster to the public Cloud, the saturation of the on-premises cluster has almost disappeared. Unlike the previous scenario, most load peaks appear only in the Fog cluster. After analysing these results, it can be concluded that reducing the Resource Manager update interval could have further mitigated these workload spikes in the Fog cluster.

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

To summarise, Figure 9 shows the average time jobs have queued in the two scenarios. As it can be appreciated, in sce-

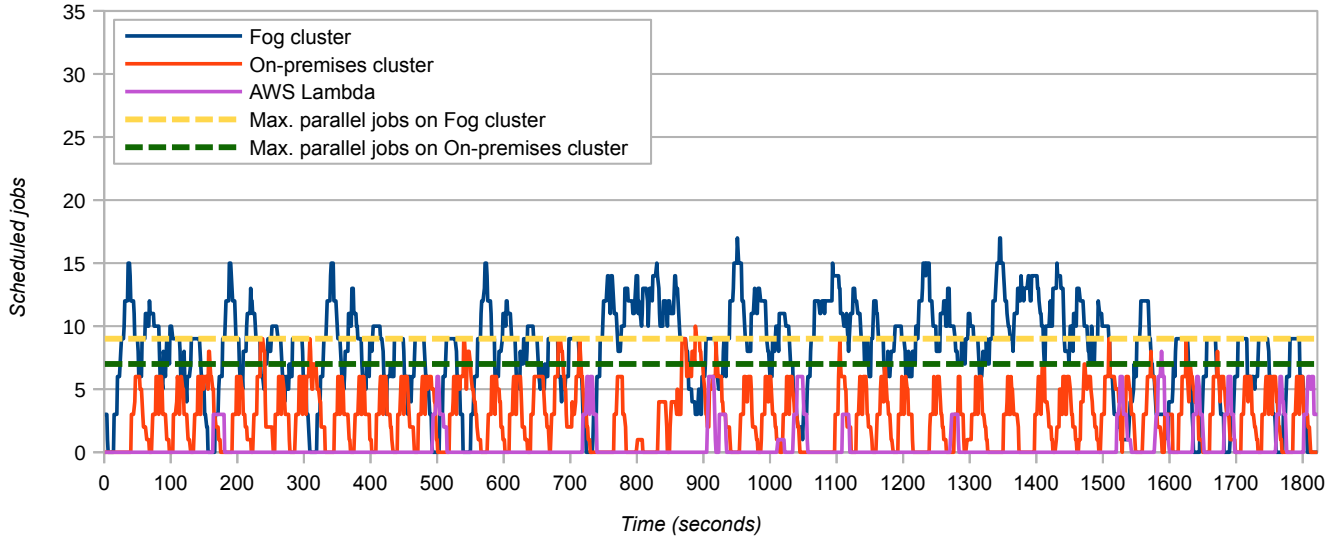

Figure 8: Number of scheduled jobs on the fog, the on-premises cluster and AWS Lambda, and the maximum number of jobs each cluster can execute simultaneously.

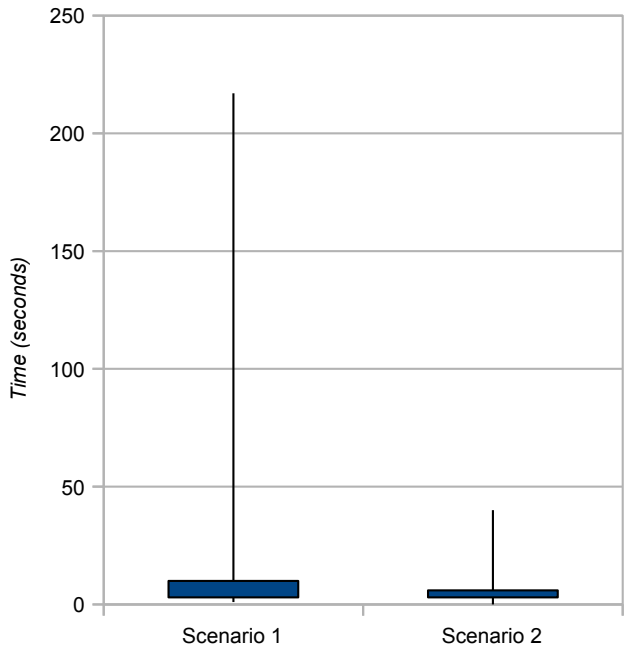

Figure 9: Average time that jobs have been queued for each scenario.

nario 2, this time has decreased notably. Indeed, AWS Lambda was introduced as an additional computing layer to offload workload executions from the on-premises cluster seamlessly. This significantly reduced the number of scheduled jobs in the on-premises cluster, as shown in Figure 8, thus alleviating its workload.

This proves that combining serverless computing with strategies to delegate jobs to replicas along the different layers of the computing continuum can considerably benefit several use cases of near real-time processing where the workload may vary in a non-predictable way. This functionality has been implemented in the open-source OSCAR framework for the sake reproducibility and to facilitate user adoption when supporting cloud-to-edge computing scenarios based on serverless computing.

The cost of delegating the execution of the 85 jobs to AWS Lambda was subsumed in the free tier, which includes one million free requests per month and 400,000 GB-seconds of compute time per month. Without considering the free tier, the cost is estimated by the AWS Pricing Calculator to be 0,12 $ in the North Virginia region.

Notice that both scenarios included a delegation approach so that each OSCAR service could offload workload to a single replica located in an upper layer of the cloud-to-edge continuum (edge, fog and cloud). However, the implemented mechanism supports a set of replicas, thus being able to exploit additional potentially distributed resources from a layer before offloading into another layer. This facilitates the definition of more complex scenarios in which OSCAR service replicas can be simultaneously employed within layers of the cloud-to-edge continuum.

5. Conclusions and Future Work

This paper has presented different strategies for delegating jobs on the OSCAR open-source serverless data-processing platform that runs on top of Kubernetes. To exemplify the operation of the two delegation mechanisms implemented, a use case was developed based on a pre-existing fire detection AI model and then adapted to the OSCAR platform. The experimentation carried out has allowed, in addition to testing the

operation of the rescheduler and the resource manager, the benefits of delegating Serverless jobs to a different on-premises cluster, but also to FaaS services on public cloud providers, thus making use of the different layers of the cloud-to-edge continuum. The results indicate that such approach can be beneficial for several use cases where the workload is unpredictable, and relying only on edge processing devices can significantly limit the ability to handle information quickly.

Future work involves fine-tuning the implementation of the Rescheduler component to minimize the execution of duplicate jobs. Also, adapting the Resource Manager mechanism to support additional workload scheduling systems on top of Kubernetes, such as Apache Yunikorn, is currently being used to limit the number of resources per service within an OSCAR cluster. In addition, we want to assess the effectiveness of the proposed strategies when including multiple replicas across the different layers of the edge-to-cloud continuum, including latency-aware algorithms to decide the delegated OSCAR service replica. Finally, we plan to introduce support for dynamically changing the replicas of an OSCAR service to reflect changes in the underlying infrastructure with the dynamic addition and removal of virtualized computing resources.

Acknowledgements

Grant PID2020-113126RB-I00 funded by MCIN/AEI/10.13039/501100011033. Project PDC2021-120844-I00 funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/PRTR. This work was supported by the project AI-SPRINT "AI in Secure Privacy-Preserving Computing Continuum" that has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant 101016577. This work was also supported by the project AI4EOSC "Artificial Intelligence for the European Open Science Cloud" that has received funding from the European Union's Horizon Europe Research and Innovation Programme under Grant 101058593.

References

[1] P. Beckman, J. Dongarra, N. Ferrier, G. Fox, T. Moore, D. Reed, M. Beck, Harnessing the computing continuum for programming our world, Fog Computing (2020) 215–230doi:10.1002/9781119551713.ch7.

[2] A. OpenFog Consortium Architecture Working Group, et al., Openfog reference architecture for fog computing, OPFRA001 20817 (2017) 162.

[3] M. Jansen, A. Al-Dulaimy, A. V. Papadopoulos, A. Trivedi, A. Iosup, The spec-rg reference architecture for the edge continuum (7 2022).
URL http://arxiv.org/abs/2207.04159

[4] S. Ketu, P. K. Mishra, Cloud, fog and mist computing in iot: an indication of emerging opportunities, IETE Technical Review 39 (3) (2022) 713–724.

[5] A. Ghasempour, Internet of things in smart grid: Architecture, applications, services, key technologies, and challenges, Inventions 4 (1) (2019) 22.

[6] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, P. A. Polakos, A comprehensive survey on fog computing: State-of-the-art and research challenges, IEEE Communications Surveys & Tutorials 20 (1) (2018) 416–464. doi:10.1109/COMST.2017.2771153.

[7] Google, Knative.
URL https://github.com/knative/

[8] A. Ellis, Openfaas.
URL https://www.openfaas.com/

[9] GRyCAP, OSCAR: Open Source Serverless Computing for Data-Processing Applications.
URL https://oscar.grycap.net

[10] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, I. Stoica, Caerus:{NIMBLE} task scheduling for serverless analytics, in: 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), 2021, pp. 653–669.

[11] K. Kaffes, N. J. Yadwadkar, C. Kozyrakis, Centralized core-granular scheduling for serverless functions, in: Proceedings of the ACM symposium on cloud computing, 2019, pp. 158–164.

[12] T. Rausch, W. Hummer, V. Muthusamy, A. Rashed, S. Dustdar, Towards a serverless platform for edge {AI}, in: 2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19), 2019.

[13] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, P. Chen, Function delivery network: Extending serverless computing for heterogeneous platforms, Software: Practice and Experience 51 (2021) 1936–1963. doi:10.1002/SPE.2966.
URL https://onlinelibrary.wiley.com/doi/full/10.1002/spe.2966https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2966https://onlinelibrary.wiley.com/doi/10.1002/spe.2966

[14] C. Sicari, L. Carnevale, A. Galletta, M. Villari, Openwolf: A serverless workflow engine for native cloud-edge continuum, in: 2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), IEEE, 2022, pp. 1–8.

[15] F. Smirnov, C. Engelhardt, J. Mittelberger, B. Pourmohseni, T. Fahringer, Apollo: towards an efficient distributed orchestration of serverless function compositions in the cloud-edge continuum, dl.acm.org (12 2021). doi:10.1145/3468737.3494103.
URL https://dl.acm.org/doi/abs/10.1145/3468737.3494103

[16] N. Ferry, R. Dautov, H. Song, Towards a model-based serverless platform for the cloud-edge-iot continuum, Proceedings - 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022 (2022) 851–858doi:10.1109/CCGRID54584.2022.00101.

[17] A. Pérez, S. Risco, D. M. Naranjo, M. Caballer, G. Moltó, On-Premises Serverless Computing for Event-Driven Data Processing Applications, in: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2019, pp. 414–421, iSSN: 2159-6182. doi:10.1109/CLOUD.2019.00073.

[18] Kubernetes, Kubernetes.
URL https://kubernetes.io/

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

[20] A. Pérez, G. Moltó, M. Caballer, A. Calatrava, Serverless computing for container-based architectures, Future Generation Computer Systems 83 (2018) 50–59. doi:10.1016/j.future.2018.01.022.
URL https://linkinghub.elsevier.com/retrieve/pii/S0167739X17316485

[21] S. Risco, G. Moltó, D. M. Naranjo, I. Blanquer, Serverless Workflows for Containerised Applications in the Cloud Continuum, Journal of Grid Computing 19 (3) (2021) 30. doi:10.1007/s10723-021-09570-2.
URL https://doi.org/10.1007/s10723-021-09570-2

[22] M. Caballer, I. Blanquer, G. Moltó, C. de Alfonso, Dynamic management of virtual infrastructures, Journal of Grid Computing 13 (1) (2015) 53–70. doi:10.1007/s10723-014-9296-5.
URL https://doi.org/10.1007/s10723-014-9296-5

[23] M. A. Moritz, Wildfires ignite debate on global warming, Nature 487 (7407) (2012) 273–273.

[24] Cloud Native Computing Foundation, K3s.
URL https://k3s.io/

[25] W. Thompson, N. Bhowmik, T. Breckon, Efficient and compact convolutional neural network architectures for non-temporal real-time fire detection, in: Proc. Int. Conf. Machine Learning Applications, IEEE, 2020, pp. 136–141. doi:10.1109/ICMLA51294.2020.00030.
URL http://breckon.org/toby/publications/papers/thompson20fire.pdf

[26] Amazon Web Services, Push Notification Service - Amazon Simple No-

tification Service (SNS).
URL `https://aws.amazon.com/sns/`