FaDO: FaaS Functions and Data Orchestrator for Multiple Serverless Edge-Cloud Clusters

Christopher Peter Smith*, Anshul Jindal*, Mohak Chadha*, Michael Gerndt*, Shajulin Benedict†,
*Chair of Computer Architecture and Parallel Systems, Technische Universität München, Germany
†Department of Computer Science and Engg., Indian Institute of Information Technology Kottayam, Kerala, India.
Email: {christopher.smith, anshul.jindal, mohak.chadha}@tum.de*, gerndt@in.tum.de*, shajulin@iiitkottayam.ac.in†

***Abstract*—Function-as-a-Service (FaaS) is an attractive cloud computing model that simplifies application development and deployment. However, current serverless compute platforms do not consider data placement when scheduling functions. With the growing demand for edge-cloud continuum, multi-cloud, and multi-serverless applications, this flaw means serverless technologies are still ill-suited to latency-sensitive operations like media streaming. This work proposes a solution by presenting a tool called FaDO: FaaS Functions and Data Orchestrator, designed to allow data-aware functions scheduling across multi-serverless compute clusters present at different locations, such as at the edge and in the cloud. FaDO works through header-based HTTP reverse proxying and uses three load-balancing algorithms: 1) The Least Connections, 2) Round Robin, and 3) Random for load balancing the invocations of the function across the suitable serverless compute clusters based on the set storage policies. FaDO further provides users with an abstraction of the serverless compute cluster's storage, allowing users to interact with data across different storage services through a unified interface. In addition, users can configure automatic and policy-aware granular data replications, causing FaDO to spread data across the clusters while respecting location constraints. Load testing results show that it is capable of load balancing high-throughput workloads, placing functions near their data without contributing any significant performance overhead.**

***Index Terms*—serverless, function-as-a-service, data-aware, multi-cloud, orchestration, edge-computing**

I. Introduction

Significant progress has been made in different domains [1]–[4] based on the idea of *serverless computing* since its launch by Amazon as AWS Lambda in November 2014 [5]. Serverless computing is a cloud computing model that abstracts server management and infrastructure decisions away from the users [6]. In this model, the allocation of resources is managed by the cloud service provider rather than by *DevOps*, thereby benefiting them from various aspects such as no infrastructure management, automatic scalability, and faster deployments [7]–[10]. Function-as-a-Service (FaaS) is a key enabler of serverless computing [6]. In FaaS, a serverless application is decomposed into simple, standalone functions that are uploaded to a serverless compute platform such as AWS Lambda [11], Google Cloud Function (GCF) [12], and Azure Functions (AF) [13] for execution. Functions can be invoked by a user's HTTP request or by another type of event created within the serverless compute platform. The FaaS platform is responsible for deploying and facilitating resources to the application functions.

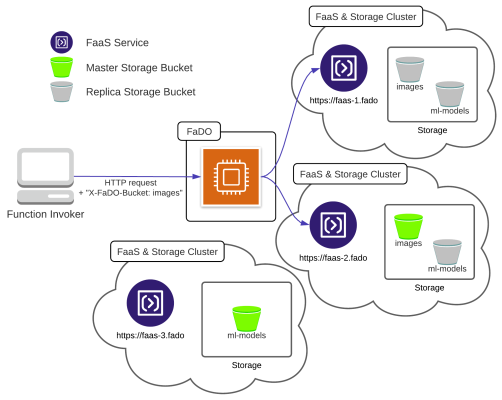

Fig. 1: An example usecase of *FaDO* to direct function's invocations accessing *images* storage bucket to relevant clusters.

The benefit of not having to manage the infrastructure also comes with some challenges, one of them being the function placement [14]–[16]. To achieve good performance, the infrastructure should be able and willing to co-locate particular code and data physically. This is often best achieved by shipping code to data, rather than the approach of pulling data to code [17]. When using FaaS, the cloud provider is responsible for scheduling the workload. However, data placement is not considered, leading to a non-optimal performance. The function's execution time varies based on the location of its accessing data [18]. A local data access will be faster than remote data access, which has an overhead of network latency. The data-access variations cause differences in performance between the identical function deployments for the same serverless compute platform. Latency-sensitive tasks, such as media streaming or complex distributed machine learning calculations, are thus not well suited to the current FaaS model.

Moreover, with the growing market and popularity of the Internet-of-Things (IoT), an increasing number of significant applications are run on heterogeneous and distributed serverless compute clusters [18]–[21]. These clusters combine computing resources in different locations (such as at the edge, or in the cloud in different regions) and with varying constraints, making the control over function placement, and also data placement (for example, to obey privacy laws like the General Data Protection Regulation (GDPR)), a desirable feature.

To this end, this work focuses on resolving the issues mentioned above of data-access performance and management in the multiple heterogeneous serverless compute clusters co-located with storage services by dynamically placing both functions and data. Towards this, we develop a tool called **FaDO: <u>Fa</u>aS Functions and <u>D</u>ata <u>O</u>rchestrator** to allow data-aware function scheduling across multiple serverless compute clusters. *FaDO* also provides a unified and central interface to the various storage services on the multi-serverless compute clusters, allowing users to upload and download data objects and organize them into storage buckets.

An example use case of *FaDO* is shown in Figure 1 where we have three serverless compute clusters co-located with storage databases, and the client invoking a function requires the storage bucket named *images*. When *FaDO* receives the invocation, it looks for the *X-FaDO-Bucket* header. It uses its value to select the appropriate subset of FaaS endpoints (serverless compute clusters endpoints) and load balances among them based on the set scheduling policy. In this case, *FaDO* load balances the client's function invocations to the FaaS services at *https://faas-1.fado* and *https://faas-2.fado*. Users can organize their data objects into storage buckets and set bucket-specific replication policies to influence how *FaDO* distributes the data across the clusters. *FaDO* monitors the different storage buckets and watches for changes to their replication policies to keep all replica sets up-to-date and consistent with their policies. This includes creating new replica buckets or deleting unneeded ones and sending new data to out-of-sync replica buckets.

The key contributions of this work are:

- We develop and present a tool, called *FaDO*, to allow data-aware functions scheduling and data placement across multi-serverless compute clusters. To the best of our knowledge, this is the first work to do so.
- We present the performance results of *FaDO* and examine the viability of its design by creating four serverless compute clusters: three based on self-hosted OpenFaaS platform having different h/w configurations and one using AWS Lambda. Each of the clusters is attached with MinIO as the storage service.
- We use three load-balancing algorithms: 1) The Least Connections, 2) Round Robin, and 3) Random for load balancing the invocations of the function across the suitable and viable serverless compute clusters based on the storage configurations.

Paper organization. In Section II, the system architecture of *FaDO* and its components are explained. In Section III, we describe the evaluation setup. In Section IV, performance evaluation results are presented. Section V describes some prior works in this domain. Finally, Section VI concludes the paper.

II. System Architecture

In this section, we present a high-level overview of the *FaDO* shown in Figure 2 along with the interaction between its components. Following the philosophy of modular systems design, we compose *FaDO* with the four subcomponents described in the following subsections.

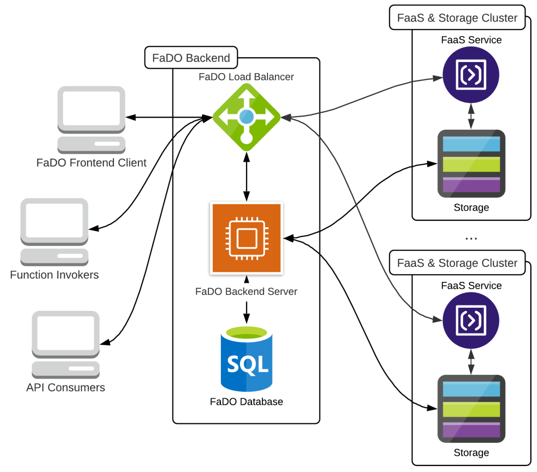

Fig. 2: FaDO's system architecture and relationships with FaaS and storage services across multiple serverless clusters.

A. The Database

The database stores details concerning the different resources in the serverless compute clusters, including FaaS endpoints, storage deployments, the different storage buckets, and their contained data objects. It also stores information on the different replication sets, detailing where the master buckets and their replicas are located. Furthermore, the database keeps current load balancing information configurations and user-defined policies. Since *FaDO* mostly contends with relational data, we selected PostgreSQL [22], a highly performant Database Management System (DBMS) as our database. Figure 3 shows the data model used by *FaDO*. While some of these entities are easily recognizable as real *things*, such as a storage deployment, others are more abstract, like a policy. The database schema uses tables and relationships to give concrete definitions to these entities and allows *FaDO* to handle and manipulate them. We briefly describe each entity in the below subsections:

1) Policies: Policies are used in the data model to define specific settings. These can be global settings or specific to a cluster or a bucket. The schema defines the `policies` table, which lists the system's existing settings, giving them a name and a default value. The `global_policies`, `clusters_policies`, and `buckets_policies` tables, allow users to define policy values that are applied globally or to a given cluster or bucket. This value then overrides the default value set on the `policies` table. These tables by default store global scheduling settings, *zone* policies to dictate where storage buckets can go.

2) Clusters: Clusters are records with an ID, and a name representing serverless compute clusters forming storage and FaaS deployment. There is a one-to-many relationship between clusters and storage and FaaS deployments. Each storage or FaaS deployment belongs to one cluster, and one cluster can have multiple storage or FaaS deployments. However, in the scope of this work, there is one-to-one relationship between clusters and storage and FaaS deployments. Furthermore, the

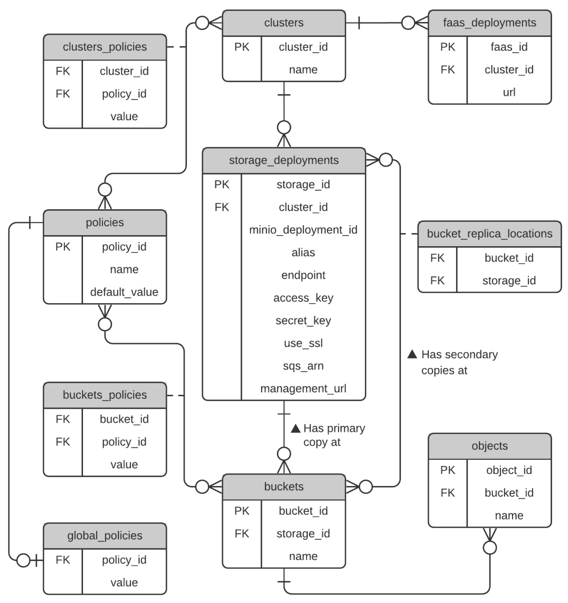

Fig. 3: *FaDO*'s data model.

`clusters_policies` links clusters and policies together and forms a many-to-many relationship, where a cluster can have many associated policies and vice-versa. Using these tables and relationships, *FaDO* users can associate co-located FaaS and storage deployments and apply cluster-wide policies. This allows the definition of zones, which are conceptual delimitations useful for determining storage bucket placement.

3) FaaS Deployments: The `faas_deployments` table defines the different FaaS endpoints running the serverless compute platforms. It is a simple table that only indicates the platforms' URLs and clusters to which they belong. This is because, at this time, *FaDO* does not interact with the serverless compute platforms beyond reverse-proxying traffic back to them.

4) Storage Deployments: The `storage_deployments` table tracks all MinIO deployments across different serverless clusters. The table indicates the deployment's serverless cluster and the connection data provided by the user that *FaDO* needs to communicate with the services and other relevant metadata that it gathers subsequently. This additional metadata includes values like the `minio_deployment_id`, a unique ID that MinIO services generate for themselves and help *FaDO* to determine the storage deployment.

5) Buckets: The `buckets` table lists all the storage buckets *FaDO* is aware of amongst the different storage deployments. These buckets are the containers for data objects and the grouping method used to organize data replication and scheduling. The `buckets_policies` table allows the definition of bucket-specific settings. Buckets only directly refer to the one storage deployment that keeps their master copy. However, the `replica_bucket_locations` table links the tables together to relate bucket replica locations. The schema allows *FaDO* to determine a bucket's associated clusters through these means. As function invocations specify a storage bucket, *FaDO* can choose where to forward the requests by listing the different clusters the bucket is associated with.

6) Objects: The `objects` table lists the data objects present on the different storage deployments. Every object belongs to a singular bucket, and their locations can be inferred by their bucket's master and replica locations.

B. The Load Balancer

It distributes HTTPS traffic and reverse-proxy function invocations to the appropriate serverless clusters. *FaDO* intends to provide transparent proxying of function invocations to achieve data-dependent function placement. Currently, *FaDO* only considers function invocations made through HTTP endpoint triggers. Hence, *FaDO* routes HTTP requests and expects data requirements to be passed through HTTP headers. *FaDO* routes the request as-is to the selected upstream. The load balancer is mainly responsible for load balancing function invocations to different FaaS upstreams within the serverless clusters based on information passed through HTTP headers, set scheduling policies, and storage configurations. It is also used as a general ingress for all the *FaDO*'s traffic. Many options exist that can fulfill the *FaDO*'s load-balancing requirements, such as HAProxy and NGINX. However, Caddy Server [23] was preferred for this work. It is written in Go, dynamically configurable through an administration endpoint using JSON objects, and provisions and maintains TLS certificates automatically. This makes it an ideal choice for *FaDO*, as the backend server can easily configure it through its administration API.

C. The Backend Server

FaDO's backend server written in Go is the most complex system developed for this work. Its complexity largely stems from its interactions with external systems, among which are:

- Storing data in, and retrieving it from, the database
- Creating the function invocation scheduling configurations and sending it to the load balancer
- Manipulating and monitoring the storage deployments
- Automating storage bucket replications

In the following subsections, we present briefly the backend server's various interactions with external systems, including the other *FaDO* components and MinIO storage deployments.

1) Querying the Database: The backend server uses the *pgx* and *pgxpool* packages [24] to interface with the database. These provide a Go driver and toolkit for interfacing with PostgreSQL instances and performing safe concurrent operations. The *pgxconn* package provides facilities to create database connections and transactions, which come in the form of the `pgxconn.Conn` and `pgxconn.Tx` structures.

2) Transactions: The *FaDO* database contains information about the different storage clusters and the Caddy server's load balancing configuration. However, these systems are all independent of one another, and the database's state may not always be in sync with all of them. Therefore, a cautious approach to data mutations must be taken. Database transactions contribute to this pursuit. This unit of instructions can either be *committed* or *rolled back*. All the changes resulting from its

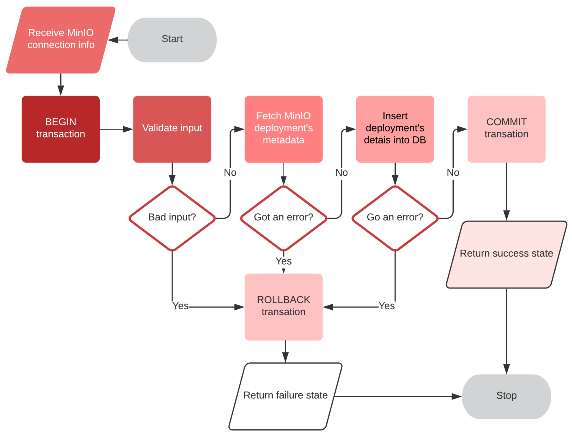

Fig. 4: Wrapping operations in a database transaction.

```
[
  {
    "handle": [
      {
        "handler": "reverse_proxy",
        "load_balancing": { "selection_policy": {  "policy": "round_robin" } },
        "upstreams": [
          { "dial": "faas.hpc-cluster.fado:31112" },
          { "dial": "faas.cloud-cluster-2.fado:31112" }
        ]
      }
    ],
    "match": [
      {  "header": { "X-FaDO-Bucket": ["images"] } }
    ]
  },
  {
    "handle": [
      {
        "handler": "reverse_proxy",
        "load_balancing": { "selection_policy": { "policy": "least_conn" } },
        "upstreams": [
          { "dial": "faas.hpc-cluster.fado:31112" },
          { "dial": "faas.cloud-cluster-1.fado:31112" }
        ]
      }
    ],
    "match":[
      { "header": { "X-FaDO-Bucket": ["ml-models"] } }
    ...
```

Listing 1: Load balancing routes for *images* and *ml-models* storage buckets with multiple upstreams and selection policies.

operations will be saved if a transaction is committed. However, if it is rolled back, all the changes will be reverted [25].

Figure 4 shows the basic sequence of events involved in tracking a new storage deployment in FaDO. At first, the backend server receives input from a client giving the new storage deployment's connection information. At this point, a database transaction is started. The data is then validated and checked against the current database state. Following this, the server fetches various metadata from the storage deployment and then inserts the deployment's details into the database. If everything goes to plan, the transaction is committed, and a successful state is returned, but if an error was encountered along the way, the transaction is rolled back, and an error state is returned instead.

3) Generating scheduling routes: Routes must be generated for each storage bucket once the load balancer is configured to receive function invocation requests. These routes specify a header match, a load balancing selection policy, and a set of upstream URLs. Listing 1 shows an example of the routes generated by *FaDO* for two storage buckets, *images* and *ml-models*. Both storage buckets are close to two FaaS endpoints, and so they each have two upstream URLs. The route for *images* specifies the *round_robin* selection policy, which will cause the load balancer to cycle through each upstream one after another. On the other hand, the route for *ml-models* uses the *least_conn* selection policy, standing for *least connections*. This policy causes the load balancer to send the next request to the upstream with the least open connections at that time.

4) Interacting with storage services: MinIO was selected as the object storage service in this work. MinIO is an S3-compatible object storage technology that offers flexible bucket replication features [26], [27]. Its compatibility with Amazon's widely popular S3 storage solution makes it representative of current cloud storage technologies, and its bucket replication features provide building blocks to implement *FaDO*'s granular bucket replication mechanisms. This work further leverages MinIO's `mc` command line tool [28] and its Javascript and Go language SDKs [29], [30] to integrate the different MinIO deployments with *FaDO*. With these tools and methods in place, the backend server accomplishes four different operations:

Tracking a new storage deployment: To track a new MinIO service in *FaDO*, the backend server goes through several steps shown in Figure 5. It first receives necessary connection information (name, endpoint, access key, and secret key) to connect to it. Once the connection information is given, the server verifies its apparent validity. Once done, the backend server configures the *mc* command-line client, with the new connection and sets up an alias according to the provided name. It then uses the tool to configure the MinIO deployment with a new notification webhook that points back to the server. This allows to set notification configurations on storage buckets and causes MinIO to send notifications back to the server when changes occur. Then MinIO is restarted to reflect the changes.

Manipulating buckets and objects: One of the goals of *FaDO* is to provide a layer of abstraction on top of the storage deployments and allow users to interact with them in a unified fashion. To this purpose, *FaDO* allows for the creation and deletion of storage buckets as well as the adding, downloading, and deletion of storage objects. All these operations are possible using MinIO Go SDK. The backend server needs to monitor all storage buckets across various clusters. It adds itself as a notification target for each newly created bucket to receive bucket notifications from MinIO whenever a bucket is changed. The backend server's API further allows clients to manipulate data objects directly by acting as a go-between and using the MinIO SDK, essentially proxying objects between the client and the MinIO services.

Handling notifications: *FaDO* server has a dedicated API endpoint at the path `/api/notify` that is designed to handle MinIO notifications. The storage deployment that generates the notification sends a JSON object in the request body. This JSON object can be parsed to extract the affected object's *key* and the storage deployment's `deployment_id`. *FaDO* then recognizes which object, in which bucket, and on which storage deployment has been affected. Suppose the bucket

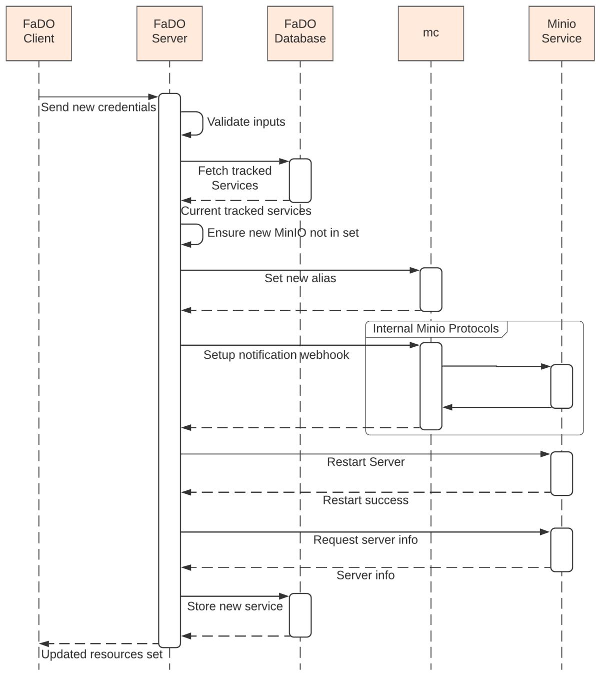

Fig. 5: Sequence of events to track a new storage deployment.

that generated the event is a master copy. In that case, *FaDO* mirrors the new contents to all the replica buckets and then updates the information in the database before reconfiguring the load balancer with the new routes.

Mirroring buckets: *FaDO* relies on the `mc mirror` command, a feature of the *mc* command-line tool that allows bucket mirroring and read replication between buckets on MinIO storage deployments.

D. The Frontend Client

Though users can interface with *FaDO* through the backend server's API, a user interface developed in ReactJS that allows users to interface with the system easily is also provided.

III. Evaluation Settings

We designed our experiments to answer the questions:

Q1. *FaDO*'s data replication and storage policies correctness: *FaDO* automatically configures functions' scheduling policies through storage configurations. Therefore, here we analyze the correctness of *FaDO*'s automatic policies generation based on storage configurations.

Q2. *FaDO*'s replication performance: *FaDO*'s replication mechanisms are built on top of MinIO's *mc* command-line tool using the `mc mirror` directive. Thus, we closely examine the tool through its performance by recording replication completion times under different scenarios.

Q3. *FaDO*'s data-aware functions scheduling performance: To confirm *FaDO*'s performance for data-aware function scheduling, measuring and recording function completion times under various scenarios is necessary. These measurements will provide insight into *FaDO*'s overall performance, overhead, and the impact of its different scheduling policies.

Cluster	Processor	H/W Spec.	Software
HPC-cluster (1 Node)	Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz	22 CPUs 754GiB mem.	OpenFaaS MinIO
Edge-cluster (3 Nodes)	ARMv8 Processor rev 1 (v8l)	4 CPUs 4GiB mem.	OpenFaaS, MinIO
Cloud-cluster (3 Nodes)	Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz	4 CPUs 8GiB mem.	OpenFaaS, MinIO
AWS-cluster -	- -	- -	AWS Lambda, MinIO

TABLE I: Specifications for the clusters for evaluating *FaDO*.

A. Testing Environment

To demonstrate the working of *FaDO*, we created four different serverless compute clusters. The hardware and software characteristics for each cluster is listed in Table I. The *Edge-cluster* consists of three embedded Nvidia Jetson Nano devices. We utilized OpenFaaS on top of k3s [31], a lightweight version of Kubernetes, to host a Kubernetes cluster on it. A high-performance computing (HPC) cluster provisioned on-premise. It is an HPC node and the most powerful cluster in this work, called *HPC-cluster*. We disabled hyper-threading and turbo boost on it. OpenFaaS on top of Kubernetes is deployed in it. The *Cloud-cluster* is composed of three virtual machines hosted on a private cloud at the Leibniz Supercomputing Center (LRZ). OpenFaaS on top of Kubernetes is also deployed in it. Additionally, we created a cloud cluster using AWS lambda called *AWS-cluster*. For this cluster, internal configuration details of the VMs or the containers in which the functions are deployed are not available.

Each of the above clusters except *AWS-cluster* also runs a MinIO deployment. Since we cannot run MinIO in AWS lambda, therefore for *AWS-cluster*, we created a *t2.medium* instance VM to run the MinIO storage service. All the functions are deployed with 1024MB memory. The control plane hosting *FaDO*: the backend server, the Caddy load balancer, and the PostgreSQL database, runs on a VM provisioned in a private cloud having four vCPU cores and 8 GiB of memory.

B. Benchmarks

To investigate the performance of *FaDO*, we used two functions: 1) An image processing function at path `/function/image` that receives a bucket and object name. It fetches the data from its local MinIO service and manipulates the image before finishing. It is referred to as the ***image*** function, and 2) An information function at path `/function/nodeinfo` that does not take any input and does not access MinIO, but returns information concerning the current cluster. It is referred to as the ***nodeinfo*** function.

IV. Results

In this section, we present the results concerning our evaluation questions.

A. FaDO's data replication and storage policies correctness

Beyond letting users override storage configurations and manually define where a master bucket should be replicated and which FaaS upstreams a bucket's load balancing route should contain, *FaDO* automatically configures functions scheduling policies through storage configurations. These configurations include buckets' allowed zones and their target replica counts. A scenario was devised to confirm that *FaDO*'s replication and policy mechanisms work, wherein gradual changes are made using *FaDO*'s frontend client, and the results of the backend server's operations are recorded by fetching Caddy's configuration. The steps took place as follows:

1) The *HPC-cluster*, *Edge-cluster* and *Cloud-cluster* are used. Each belongs to the zone *Germany*, while the *HPC-cluster* and *Edge-cluster* additionally belong to zone *high-performance*. Since there are no buckets in the beginning, caddy's configuration also shows no scheduling policies (Artificial hostname *lb.fado* used instead of privately owned domain) :

```
{
  "handle": [
    { "handler": "subroute" }
  ],
  "match": [{ "host": ["lb.fado"] }],
  "terminal": true
}
```

2) A bucket named *rep-policy-demo* is created, and an object named *image.png* is added to it. The bucket is set with the allowed zone *Germany* and the target replica count 0. Caddy's configuration shows the bucket is only created on the *HPC-cluster* and therefore will direct function invocations requiring the bucket solely to the *HPC-cluster*:

```
{
  "handle":[
    {
      "handler": "reverse_proxy",
      "load_balancing": { "selection_policy": { "policy": "round_robin" }
     ↪  },
      "upstreams": [{ "dial": "faas.hpc-cluster.fado:31112" }]
    }
  ],
  "match":[
    { "header": { "X-FaDO-Bucket": ["rep-policy-demo"] } }
  ]
}
```

3) The *rep-policy-demo* bucket's target replica count is changed from 0 to 2. Caddy's configuration shows the bucket has been replicated to the other two clusters, and the scheduling route now contains the three clusters' upstreams:

```
{
  "handle":[
    {
      "handler": "reverse_proxy",
      "load_balancing": { "selection_policy": { "policy": "round_robin" }
     ↪  },
      "upstreams": [
        { "dial": "faas.hpc-cluster.fado:31112" },
        { "dial": "faas.edge-cluster.fado:31112" },
        { "dial": "faas.cloud-cluster.fado:31112" }
    ...
}
```

4) The *rep-policy-demo* bucket's allowed zones are set to no longer contain zone *Germany*, and only contain zone *high-performance*. Caddy's configuration shows the bucket has been removed from *Cloud-cluster*, and the scheduling route now only lists the upstreams for the *HPC-cluster* and *Edge-cluster*:

```
{
  "handle":[
    {
      "handler": "reverse_proxy",
      "load_balancing": { "selection_policy": { "policy": "round_robin" }
     ↪  },
      "upstreams": [
        { "dial": "faas.hpc-cluster.fado:31112" },
        { "dial": "faas.edge-cluster.fado:31112" }
    ...
}
```

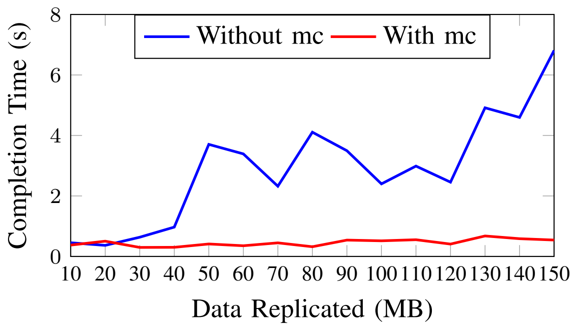

Fig. 6: Mirroring different data amounts from the *HPC-cluster* to *Cloud-cluster* using `mc` and without it.

These recordings confirm the correctness of *FaDO*'s replication and storage policy mechanisms. The backend server ensures that buckets are only replicated to allowed locations, reacting to changes to keep the replicated data up-to-date and the replica buckets consistent with user policies.

B. *FaDO's Replication Performance*

FaDO's replication performance depends entirely on MinIO's *mc* command-line tool and its `mc mirror` directive. Therefore, to assess *FaDO*'s replication performance, it was useful to isolate the command-line tool and measure its performance directly. To accomplish this, a bucket was created on the *HPC-cluster* with its replica in *Cloud-cluster* and the performance of replication of different data amounts ranging from 10MB to 150MB is analyzed under the two following scenarios:

1) Use `mc mirror` for copying the primary bucket data to the mirrored bucket every time a change is made (With `mc` in Figure 6)
2) Manually full copying the primary bucket data to the mirrored bucket every time a change is made (Without `mc` in Figure 6).

It is to be noted that the new data is cumulatively added to existing data. Performance variation for the two scenarios can be seen in Figure 6. In the first scenario, the secondary bucket contains some of the primary's content and only receives the new data at each step. This is because MinIO's mirroring tool works similarly to Andrew Tridgell and Paul Mackerras' *rsync* utility [32], [33], where the tool calculates the difference between the two buckets and only transfers the data necessary to make their content match. While in the second scenario, the whole bucket data was copied and took a much longer time. This is a significant advantage to *FaDO*, as it dramatically reduces the amount of data *FaDO* must transfer for replication tasks, which is helpful in case the master bucket resides on an edge cluster and the secondary bucket in the cloud. It also protects *FaDO* from many large replication jobs that mutex lock the `mc` tool for long periods.

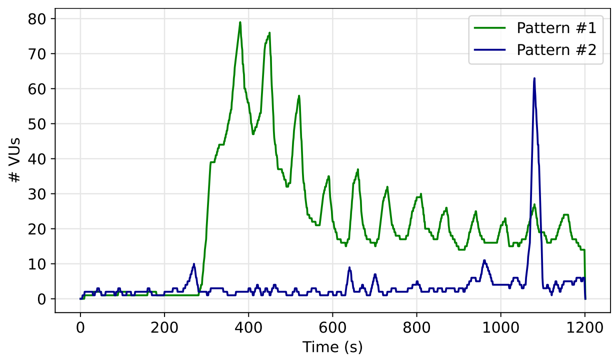

Fig. 7: The two load patterns used with *k6* visualized as the number of virtual users over time.

C. Load Balancing

k6 [34], a load testing tool, was used to test *FaDO*'s load balancing capabilities. To run tests with *k6*, the tool must be provided with a test script, which is a file written in JavaScript that contains the necessary instruction to apply load to the target application, such as sending HTTP requests. The utility will then simulate users with *virtual users* (VUs), where each VU is a process that loops through the script. The two load patterns used for testing *FaDO* are based on Wikipedia's traffic dataset offered by Kaggle [35] and are shown in Figure 7. All tests were conducted using both patterns. However, as part of this paper, we only present results on the first. The tests can be further split into three categories:

- **Direct connection tests**: these tests send function invocations directly to a single cluster. These determine a cluster's expected performance.
- **FaDO connection tests**: these tests send function invocations to a single cluster through *FaDO*'s load balancer. These determine a cluster's performance through *FaDO*.
- **Policy tests**: these tests leverage *FaDO*'s scheduling policies to distribute function invocation between the clusters.

While the *FaDO* connection test can also be considered a scheduling policy, where only one upstream is selected, the policy tests refer more specifically to the following upstream selection policies:

- ***Least Connections***: the load balancer sends the incoming request to the upstream with the least open connections.
- ***Round Robin***: the load balancer cycles through the list of upstreams.
- ***Random***: the load balancer picks a random upstream for each incoming request.

Figure 8 shows the cumulative distribution functions (CDFs) for each cluster's recorded response times for the first load balancing pattern applied to all the cluster's endpoint in both the direct connection and the *FaDO* connection tests. The X-axes relate the recorded response times, while the Y-axes indicate their probabilities. The results show that the load balancer does not significantly impact request throughput, and thus has minimal impact on cluster's performance. The direct connection and *FaDO* connection tests yield virtually the same curve in all four cases. So, not only do these two tests yield very similar numbers of requests over time, but they also have virtually the same response time distribution. This substantial evidence indicates that *FaDO*'s load balancing does not add significant overhead.

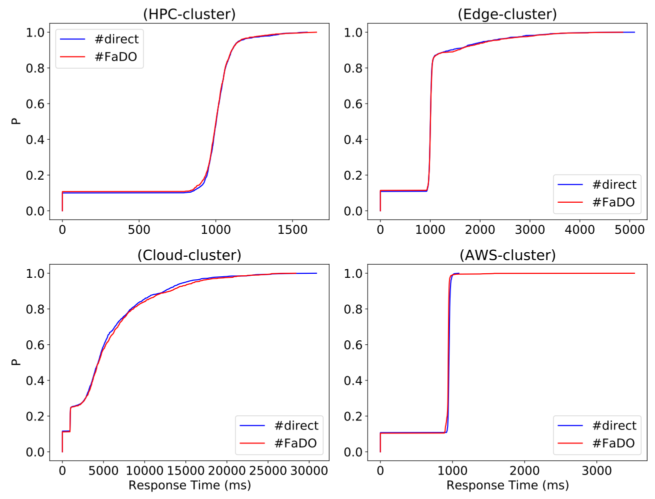

Fig. 8: The CDF for the response times recorded when connecting to the different clusters either directly or through *FaDO*. (Load Pattern 1)

Figure 8's charts further provide insight into the clusters' performances. It seems immediately apparent that *Cloud-cluster* is not as performant as the others, with a broader distribution of response times due to the network latency. On the other hand, the *HPC-cluster* and *AWS-cluster* appear to be the most performant, with a much tighter response time distribution and virtually all requests complete in about 1000 milliseconds. Finally, the *Edge-cluster* is also very good, with about 85% of its requests completed in around 1000 milliseconds. *HPC-cluster* and *AWS-cluster* are faster than the *Edge-cluster* because they have more resources and network latency is not a bottleneck.

Figure 9 gives another view of the cluster performances, relating to the cumulative number of requests sent over time. Having established that *FaDO*'s overhead is not significant, the chart now only shows measurements obtained through *FaDO*. Unsurprisingly, it also shows that the *AWS-cluster* is the most performant, followed closely by the *HPC-cluster* and then the *Edge-cluster*. The *Cloud-cluster* is, however, far outperformed.

Figure 9 also puts the load balancing policy tests into context. These measurements result from testing *FaDO*'s various scheduling policies. The load balancing tests had the load balancer send requests to all the clusters. The *least connections* policy that sends requests to the upstream with the smallest number of open connections is extremely performant. Its performance sits between that of the *HPC-cluster* and *Edge-cluster* and far outperforms the *Cloud-cluster*. Evidently, its policy favors performant clusters at the *Cloud-cluster*'s expense. The *RR* and *random* policies, are virtually identical and notably

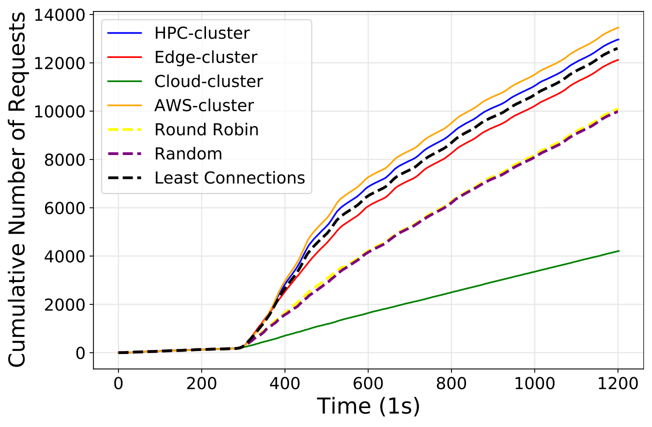

Fig. 9: The cumulative sum of completed requests recorded when connecting through *FaDO* directly to the different clusters or load balancing across them. (Load Pattern 1)

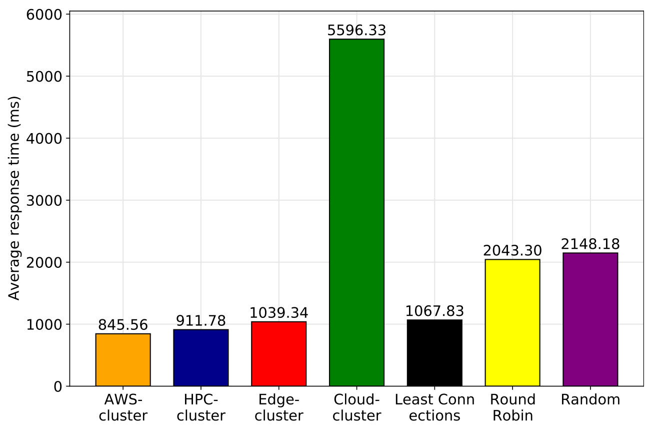

Fig. 10: The average response times obtained when connecting through *FaDO* to the different clusters or load balancing the requests across clusters. (Load Pattern 1)

worse. Though they do still outpace the *Cloud-cluster*.

Figure 10 gives a summary of the average response times observed for each policy, be it forwarding requests to a single upstream or using a selection policy to choose from a set of upstreams. The results show that the best throughput is obtained when requests are sent directly to the most powerful cluster. However, the *least connections* policy achieves comparable performance while distributing function invocations to other clusters. The *round robin* and *random* policies, while slower than the *least connections* policy, effectively mitigate the impact of a slow upstream like *Cloud-cluster*.

V. Related Work

We present prior work from two aspects:

A. Policy-aware scheduling in IaaS or PaaS

Brogi et al. present a software system called SeaClouds that tries to simplify distribution, monitoring, and migration of PaaS software across multiple heterogeneous multi-cloud platform [36]. The SeaClouds system takes the approach of deploying the different modules to the optimal platform, i.e., the platform that fulfills the requirements of the specific module. Additionally, it monitors the platform to ensure that it meets the requirements in the future. *FaDO* is purely concerned with applications built on the FaaS model. Furthermore, at this time, *FaDO* is significantly more hands-off in its approach, as it does not take responsibility for function deployment. *FaDO* tries to be as transparent as possible when it comes to function placement and implements a load balancer to proxy interactions between the user and the serverless compute clusters. Another distinguishing factor lies in *FaDO*'s data placement and replication features, as it intends to add a layer of abstraction between the user and multiple storage services. This functionality is absent from SeaCloud's definition.

Apache Brooklyn is software to control deployment, monitoring, and management of applications in multi-cloud environments [37]. Unfortunately, it has no support for FaaS.

B. Multi-Cloud Serverless Clusters

Baarzi et al. introduce the concept of a virtual serverless provider (VSP) to allow customers to deploy serverless applications to different cloud providers through a consistent interface, hiding the differences from the users and helping them escape provider lock-in [38]. They also mention the issue of data locality and the importance of placing functions as close as possible to the data since this data can be costly to move or even illegal due to regulatory restrictions. As such, there is a certain kinship between the VSPs and *FaDO*, and there is much to gain in designing *FaDO* for multi-serverless.

In our previous research, we present the concept of a Function Delivery Network (FDN) consisting of a network of multiple heterogeneous target platforms orchestrated by a control plane capable of placing functions into several FaaS platforms [18], [19]. Doing so allows reducing the overall energy consumption and provide better response times [39], [40]. In this work, we present **FaDO** which is the first implementation of FDN with data-aware function scheduling.

Google Cloud Platform (GCP) has introduced load balancing of user requests to a serverless network endpoint group (NEG) that consists of a Cloud Run, App Engine, or Cloud Functions service [41]. The load balancer serves as the frontend and proxies traffic to the specified serverless endpoint in this service. If the backend service contains multiple serverless NEGs, the load balancer balances traffic between these NEGs, thus minimizing request latency. However, serverless NEGs can point only to Cloud Functions residing in the same region where the NEG is created, and it is only restricted to their infrastructure. This is not the case with our implementation; *FaDO* can work with multiple regions, and serverless clusters.

VI. Conclusion

This work presents the **FaDO: FaaS Functions and Data Orchestrator** that achieves data-aware function placement and automates granular data replication across heterogeneous serverless compute clusters. *FaDO* leverages the cloud's *storage bucket* paradigm, using buckets as the units of replication and the metric for selecting function invocation destinations. Users of *FaDO* can thus organize their data into different storage buckets and ensure functions are scheduled close to their data by defining a special HTTP header specifying the required bucket on the function invocation requests. Load testing results further indicate that *FaDO* is capable of high-performance data-aware function scheduling through HTTP header matching.

In the future, we plan to extend *FaDO* with other public serverless compute platforms. Extending *FaDO* to use storage usage of different clusters to implement storage quota policies or hardware details to limit demanding functions to capable machines is another future perspective.

VII. Acknowledgement

This work was supported by the funding of the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program. We also thank the reviewers for their constructive and helpful suggestions.

References

[1] A. Grafberger, M. Chadha, A. Jindal, J. Gu, and M. Gerndt, "Fedless: Secure and scalable federated learning using serverless computing," in *IEEE 9th International Conference on Big Data (IEEE BigData)*, 2021.

[2] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[3] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," *arXiv preprint arXiv:1810.09679*, 2018.

[4] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*. IEEE, 2017, pp. 445–451.

[5] (2020) Aws lambda releases. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html

[6] C. S. WG, "Cncf wg-serverless whitepaper v1. 0," https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf, March 2018, [Online; Accessed: 15-July-2020].

[7] CloudFlare, "Why use serverless computing?" https://www.cloudflare.com/learning/serverless/why-use-serverless/, accessed: 2020/12/16.

[8] M. Roberts, "Serverless architectures," https://martinfowler.com/articles/serverless.html, 2018, accessed: 2020-04-17.

[9] C. Fan., A. Jindal., and M. Gerndt., "Microservices vs serverless: A performance comparison on a cloud-native web application," in *Proceedings of the 10th International Conference on Cloud Computing and Services Science - CLOSER,*, INSTICC. SciTePress, 2020, pp. 204–215.

[10] A. Jindal and M. Gerndt, "From devops to noops: Is it worth it?" in *Cloud Computing and Services Science*, D. Ferguson, C. Pahl, and M. Helfert, Eds. Cham: Springer International Publishing, 2021, pp. 178–202.

[11] Aws lambda. [Online]. Available: https://aws.amazon.com/lambda/

[12] "Cloud functions overview," https://cloud.google.com/functions/docs/concepts/overview, (Accessed on 08/22/2020).

[13] An introduction to azure functions. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview

[14] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, "Costless: Optimizing cost of serverless computing through function fusion and placement," *CoRR*, vol. abs/1811.09721, 2018. [Online]. Available: http://arxiv.org/abs/1811.09721

[15] M. Chadha, A. Jindal, and M. Gerndt, "Architecture-specific performance optimization of compute-intensive faas functions," in *IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 478–483.

[16] D. Bermbach, S. Maghsudi, J. Hasenburg, and T. Pfandzelter, "Towards auction-based function placement in serverless fog platforms," in *2020 IEEE International Conference on Fog Computing (ICFC)*, 2020, pp. 25–31.

[17] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *CoRR*, vol. abs/1812.03651, 2018. [Online]. Available: http://arxiv.org/abs/1812.03651

[18] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," *Software: Practice and Experience*, vol. n/a, no. n/a. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2966

[19] A. Jindal, J. Frielinghaus, M. Chadha, and M. Gerndt, "Courier: Delivering serverless functions within heterogeneous faas deployments," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3468737.3494097

[20] A. Jindal, M. Chadha, M. Gerndt, J. Frielinghaus, V. Podolskiy, and P. Chen, "Poster: Function delivery network: Extending serverless to heterogeneous computing," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, July 2021, pp. 1128–1129.

[21] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad, "On merits and viability of multi-cloud serverless," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 600–608. [Online]. Available: https://doi.org/10.1145/3472883.3487002

[22] The PostgreSQL Global Development Group. Postgresql - documentation. [Online]. Available: https://www.postgresql.org/docs/

[23] Stack Holdings. Welcome to caddy. [Online]. Available: https://caddyserver.com/docs/

[24] J. Christensen. pgx - postgresql driver and toolkit. [Online]. Available: https://github.com/jackc/pgx

[25] The PostgreSQL Global Development Group. Postgresql: Documentation: 13: 3.4. transactions. [Online]. Available: https://www.postgresql.org/docs/13/tutorial-transactions.html

[26] MinIO, Inc. Minio. [Online]. Available: https://min.io

[27] ——. Bucket replication. [Online]. Available: https://docs.min.io/minio/baremetal/replication/replication-overview.html

[28] ——. Minio client (mc). [Online]. Available: https://docs.min.io/minio/baremetal/reference/minio-cli/minio-mc.html

[29] ——. Minio - javascript client api reference. [Online]. Available: https://docs.min.io/docs/javascript-client-api-reference.html

[30] ——. Minio - go client api reference. [Online]. Available: https://docs.min.io/docs/golang-client-api-reference.html

[31] "rancher/k3s: Lightweight kubernetes," https://github.com/rancher/k3s, (Accessed on 07/28/2020).

[32] MinIO, Inc. mc mirror. [Online]. Available: https://docs.min.io/minio/baremetal/reference/minio-cli/minio-mc/mc-mirror.html#command-mc-mirror

[33] A. Tridgell and P. Mackerras. rsync(1) man page. [Online]. Available: https://download.samba.org/pub/rsync/rsync.1

[34] K6, "What is k6?" https://k6.io/docs/#what-is-k6, accessed: 2021-05-27.

[35] "Wikipedia traffic data exploration," https://www.kaggle.com/muonneutrino/wikipedia-traffic-data-exploration/, 2021, accessed: 2021-10-05.

[36] A. Brogi, A. Ibrahim, J. Soldani, J. Carrasco, J. Cubo, E. Pimentel, and F. D'Andria, "SeaClouds," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–4, 2 2014.

[37] A. Brooklyn, "The theory behind brooklyn," https://brooklyn.apache.org/learnmore/theory.html, accessed: 2020-01-10.

[38] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad, *On Merits and Viability of Multi-Cloud Serverless*. New York, NY, USA: Association for Computing Machinery, 2021, p. 600–608. [Online]. Available: https://doi.org/10.1145/3472883.3487002

[39] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt, "Estimating the capacities of function-as-a-service functions," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '21 Companion. New York, NY, USA: Association for Computing Machinery, 2021.

[40] M. Steinbach, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict, "Tppfaas: Modeling serverless functions invocations via temporal point processes," *IEEE Access*, vol. 10, pp. 9059–9084, 2022.

[41] G. C. Platform, "Serverless network endpoint groups overview," https://cloud.google.com/load-balancing/docs/negs/serverless-neg-concepts, accessed: 2021-05-27.