DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Self-Adaptive Data Management for Heterogeneous FaaS Platforms

Lucas Ramos Possani

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Self-Adaptive Data Management for Heterogeneous FaaS Platforms

Self-Adaptives Datenmanagement für Heterogene FaaS-Plattformen

Author:	Lucas Ramos Possani
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Anshul Jindal , M.Sc.
Start Date:	15.06.2020
Submission Date:	15.12.2020

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.12.2020 Lucas Ramos Possani

Acknowledgments

First and foremost, praise be to the God and Father of our Lord Jesus Christ, without whom nothing that has been done would have been possible. The source of knowledge and understanding. He is sovereign no matter the circumstances.

I would like to thank Prof. Dr. Michael Gerndt for giving me the opportunity to work with the Chair for Computer Architecture and Parallel Systems, the field I have been working on and have focused on during my master's degree. I would also like to thank M.Sc. Anshul Jindal for all the support in this research, for the insights and guidance in our weekly meetings.

I would like to express my appreciation to my beloved wife for all the support, for bearing with me, and for being present despite the distance.

Finally, I am thankful for my family, friends, and church members who somehow contributed with encouragement and prayers.

Abstract

Serverless computing has gained more traction over the years and, now, with the advancement and increasing number of IoT devices, the cloud model is being extended to the Edge. However, the practicality of not having to manage servers by the users also comes with its drawbacks: cloud providers do not take into account the data location when scheduling a function. By serverless computing, we specifically refer to the Function-as-a-Service (FaaS) model; and, by data location, we refer to the data the function is going to use as a payload. To evaluate the hypothesis that data location has a significant impact on the response time of the function request, we conducted experiments running functions in different clusters and using data from object storages and databases distributed in multiple locations. Additionally, we implemented a tool (FaaST) to automate the process of choosing the best cluster to deploy the function given the latency and to migrate the data close to the function when it is not available locally. The results confirm the importance of running a function close to the data it requires. Furthermore, when this is not possible, the data should be migrated while still serving the user with a slower response time.

Contents

1. Introduction

1.1. Motivation

Cloud providers have gone a step further in providing serverless services to their customers. By serverless, one can understand a service in which the backend servers are provided, maintained, and administered by the cloud provider. Examples of such services are compute, storage, analytics, etc. In particular, FaaS is a type of compute serverless service in which the user does not have to worry about infrastructure management, but only about the code being deployed.

Amazon Web Services (AWS) was the first cloud provider to offer a preview of such a service called as AWS lambda in 2014 [5]. The pricing is charged based on the number of requests to the functions and the duration, the time it takes for the function code to execute [4]. The latter varies according to the number of resources such as memory and CPU cores allocated to the function, and are automatically adapted to deliver the best performance.

The benefit of not having to manage the infrastructure also comes with some challenges, one of them being the function placement [12]. To achieve good performance, the infrastructure should be able and willing to physically co-locate certain code and data. This is often best achieved by shipping code to data, rather than the current FaaS approach of pulling data to code [17].

After AWS's Lambda release, many other companies also started to offer FaaS, such as Azure Functions[1], Cloud Functions[2] from Google, and IBM Cloud Functions[3]. IBM Cloud Functions is based on Apache OpenWhisk that was donated by IBM Research to Apache Software Foundation in 2016 [29]. Apache OpenWhisk is a serverless open source cloud platform. It works by executing functions (called actions) in response to events [28]. Actions run in containers and do not have a local state; the filesystem is ephemeral. Thus, to persist data, they have to use other services like databases or other types of external services.

Another concept that is finding a lot of attention is Fog Computing - an extension of the

[1]https://docs.microsoft.com/en-us/azure/azure-functions/
[2]https://cloud.google.com/functions#overview
[3]https://cloud.ibm.com/functions/

cloud towards the edge network [9]. With more and more IoT devices generating data, the computation is being pushed towards those devices. On the one hand, the data is processed close to its source decreasing the latency. On the other hand, those devices are usually limited in resources, which limits their compute power compared to the Cloud resources. The final architecture ends up being a hybrid environment connecting edge devices to the cloud.

However, when extending FaaS to heterogeneous clusters (edge-cloud continuum), challenges like communication latency, function scheduling, and data access patterns grow further. Current serverless platforms are limited to clusters of homogeneous nodes and homogeneous functions. Although the functions are stateless and, thus, state changes and look-ups require frequent access to databases, current platforms do not take the data access behavior of functions into account. This thesis focuses on resolving the issue of data-access performance in the edge-cloud continuum.

1.2. Problem

As mentioned previously, when using FaaS, the cloud provider is responsible for scheduling the workload. However, data placement is not taken into account, which leads to a non-optimal performance. When using Kubernetes based solutions, some services allow the user to set constraints on where the code is going to run, but the feature is not related to data placement specifically.

OpenFaaS, for example, allows the user to set constraints regarding the node used to run the functions as shown in the Code Snippet 1.

```
  constraints:
    - "node.platform.os == linux"
```

Listing 1: Function Constraints in OpenFaaS [35]

But, then, those constraints are used as nodeSelectors[4] during the deployment creation. nodeSelectors are high-level filters to make a node eligible for the pod to run. It does not matter what is running on the node.

Apache OpenWhisk also allows the user the select a set of nodes to run the task by providing a label, but it is limited to one label only and labels would not be practical for selecting nodes based on specific instances of storage as shown in the Code Snippet 2.

[4]https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodeselector

```scala
case class KubernetesInvokerNodeAffinity(
    enabled: Boolean,
    key: String,
    value: String)
```

Listing 2: OpenWhisk's node affinity [26]

On the other hand, Apache OpenWhisk also provides a way to customize the PodTemplate used by the function shown in the Code Snippet 3, which could be used to set affinity to nodes with specific volumes, but that configuration would have to change dynamically (per Pod instance).

```
# Pod template used as base for Action Pods created. It can be either
#  1. Reference to file `file:/path/to/template.yml`
#  2. OR yaml formatted multi line string. See multi line config support
 ↪  https://github.com/lightbend/config/blob/master/HOCON.md#multi-line-strings
#
#pod-template =
```

Listing 3: OpenWhisk - PodTemplate [25]

Both platforms mentioned above do not provide enough flexibility for the user to set the location concerning data placement.

1.3. Goal

The proposed thesis focuses on building a tool consisting of data placement and migration strategies which automatically chooses one of these strategies for achieving the best data-access performance. In this regard, either : (1) the function can be scheduled closer to the data, (2) data can be placed/stored where the function will always run, or (3) the data is migrated closer to where the function is scheduled - it can be done right before the function runs or by constantly replicating the data in both clusters, which would incur in storage overhead and network load. By data we mean here the data required by the function to process a request or a job. These three decision aspects are to be taken based upon how the data is accessed (access-pattern), where the data is located, how big the data is, and the availability of the resources.

This leads us to three different problem scenarios:

1. Data Placement (DP):
    - Data is always placed in the cloud cluster
    - Data is always placed in the edge cluster
    - In both Clusters (data replication)
2. Data Migration (DA):
    - Data is migrated from cloud to edge
    - Data is migrated from edge to cloud
3. Function Scheduling (FS):
    - Function is always scheduled in the cloud cluster
    - Function is always scheduled in the edge cluster
    - Random/round-robin
    - Wherever the data is located

1.4. Contributions

Towards reaching the goal, we make the following contributions: 1) We enrich the current research works in the field with extra data and discussions of the open questions regarding serverless computing. 2) We deploy a function in different environments with data spread in multiple places and assess the impact on its performance. 3) We present FaaST a data migration and placement strategy tool that helps users to get a better response time when deploying functions into FaaS platforms when using multiple storage object services and a hybrid edge-cloud continuum approach.

2. Background

2.1. Serverless

Serverless computing is a term coined by industry to describe a programming model and architecture where small code snippets are executed in the cloud without any control over the resources on which the code runs. It is by no means an indication that there are no servers, simply that the developer should leave most operational concerns such as resource provisioning, monitoring, maintenance, scalability, and fault-tolerance to the cloud provider. [6]

Serverless is a broader term that is not only applied to compute resources. A user can also leverage serverless storage services such as Amazon S3[1], among others. The search for such services has increased over time and has reached its peak in July of the present year as shown in Figure 2.1.

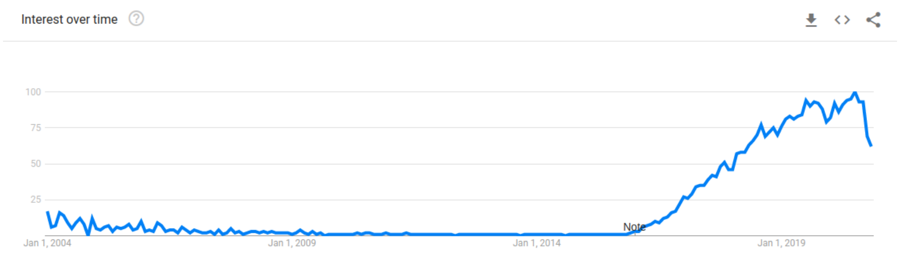

Figure 2.1.: Google Trends - Serverless [15]

2.2. FaaS Cloud Model

FaaS is just another term to describe the same service coined as serverless computing. It is a result of advancements in infrastructure technologies such as Virtual Machines (VMs) and containers - whose creation was possible due to improvements in resource isolation in Linux systems (e.g. cgroups).

[1]https://aws.amazon.com/s3/

Its general architecture is depicted in Figure 2.2. Functions, also called event-driven computing, are triggered by an event. This event can be generated by an HTTP request with its multiple methods, a new entry/modification/deletion of an entry in the database, a new file in a document store, among other cloud event sources. Then, the infrastructure will look for an existing function to fulfill the request or create a new one (in this case, it will incur in a cold start - this process involves the startup of a container that can take a few seconds). The code will be injected into the container and the message (usually a JavaScript Object Notation (JSON) format) will be processed by the function. The result will return to the caller and logs will be stored. If the function stays for a certain time idle, it will be scaled to zero. This happens to save resources on the cloud but the container stays idle for a while (warm) to diminish the impact of another cold start for a subsequent request.

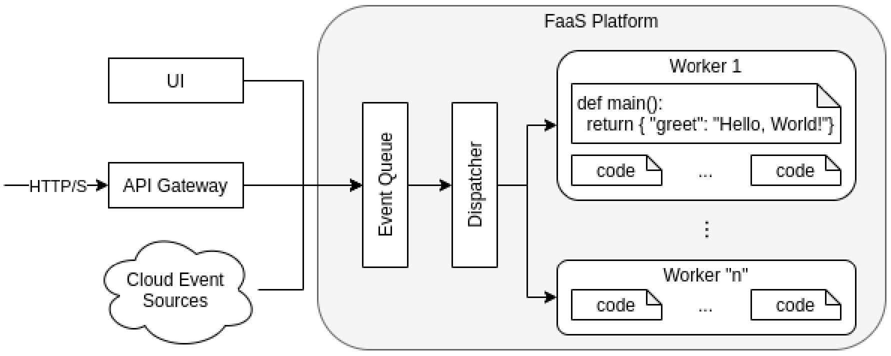

Figure 2.2.: Serverless platform architecture

FaaS also comes with limitations. It is intended for short-running tasks and the amount of memory available to each function is not unlimited either. Another characteristic of such functions is that they do not carry state. In order to consume data or share it with other functions, external communication with storage services or databases has to be done. Hellerstein et al [17] conclude that with all communication transiting through storage, there is no real way for thousands (much less millions) of cores in the cloud to work together efficiently using current FaaS platforms other than via largely uncoordinated (embarrassing) parallelism.

2.3. FaaS Platforms

There are many options available for running functions nowadays. The services offered by the main public cloud providers are: AWS Lambda (the first one launched), Azure Functions, Cloud Functions from Google, and IBM Cloud Functions.

Lee et all [24] conducted an extensive comparison among the mentioned cloud provides and their studies show that the elasticity of Amazon Lambda exceeds others regarding CPU performance, network bandwidth, and a file I/O throughput when concurrent function invocations are made for dynamic workloads.

Table 2.1 is an up to date version of their TABLE V comparing the main features of the provided services:

Table 2.1.: Feature Comparison

Item	AWS Lambda	Azure Functions	Google Functions	IBM OpenWhisk
Runtime language	Java, Go, PowerShell, Node.js, C#, Python, Ruby	C#, JavaScript, F#, Java, PowerShell, Python, TypeScript	Node.js, Python, Go, Java, .NET	JavaScript, Python, Go, Swift, Ruby, PHP, Java, .NET, Ballerina (exp), Rust (exp)
Trigger	30	12	6	n/a
Price per Memory	$0.0000166667/GB-s	$0.000016/GB-s	$0.0000025/GB-s	$0.000017/GB-s
Price per Execution	$0.20 per 1M	$0.20 per 1M	$0.40 per 1M	n/a
Free Tier	400,000 GB-s / First 1M Exec	400,000 GB-s / First 1M Exec	400,000 GB-s / First 2M Exec	400,000 GB-s / First 5M Exec
Maximum Memory	3,008 MB	1,536 MB	2,048 MB	2,048 MB
Container OS	Amazon Linux	Windows, Linux	n/a	n/a
Container CPU Info	2 CPUs	1 CPU	n/a	n/a
Temp Directory (Path)	512 MB (/tmp)	500 MB (%SYSTEM-DRIVE%\local\Temp, %SYSTEM-DRIVE%\local\AppData)	"tmpfs" volume stored in memory (/tmp)	n/a
Execution Timeout	15 minutes	10 minutes	9 minutes	10 minutes
Code Size Limit	50 / 250 MB (zipped/un-zipped)	50 MB	50 / 250 MB (com-pressed/un-compressed)	48 MB

Here are some notes about Table 2.1: (1) Azure Functions show Runtime language version

3.x. (2) *exp* stands for experimental in IBM OpenWhisk Runtime Language. (3) Azure Functions makes available 5 TB for a total of 100 Function apps resulting in about 50 MB per app for the code size limit. (4) Lambda allocates CPU power linearly in proportion to the amount of memory configured. (5) Google memory size for Tier 1 is used. (6) n/a stands for the information not available.

In the context of public cloud providers, the developer only cares about the application, and all the maintenance and infrastructure management is done by the provider as previously said. However, many are the options following the *do-it-yourself* model. Open source platforms have also gained popularity and maturity over time. FnProject[2] and IronIO[3] claim to run anywhere, meaning one can use a laptop, a server, or the cloud as infrastructure. Others are more specific aiming container-orchestrator platforms, leveraging the scalability, monitoring, and other services provided by the platform: OpenFaaS[4], Fission[5], and Kubeless[6]. Additionally, we have research projects aiming to enable the exploration of new approaches to serverless computing taking Lambda as the study case [18]. One has to take into account that by using open-source projects, one has to also take care of the infrastructure so that developers stay focused on the application code.

In this work, we chose Apache OpenWhisk as the open-source platform to conduct our studies. Thus, we will take a closer look at its details.

2.3.1. Apache OpenWhisk

Apache OpenWhisk is an open source, distributed Serverless platform that executes functions (fx) in response to events at any scale. OpenWhisk manages the infrastructure, servers and scaling using Docker containers so you can focus on building amazing and efficient applications. [31]

Programming Model

OpenWhisk's programming model is event-driven. When an event occurs, a function (here called *action*) is invoked. This event comes from a variety of external services that can be: Datastores, Message Queues, Mobile and Web Applications, Sensors, Chatbots, etc.

Those services trigger actions when something happens and are associated by using rules. A simple example would be a block storage system that, whenever an image is uploaded,

[2]https://fnproject.io/
[3]https://open.iron.io/
[4]https://www.openfaas.com/
[5]https://fission.io/
[6]https://kubeless.io/

triggers an action to create a thumbnail. In this example, the trigger would be the uploading of an image, the action would be the creation of a thumbnail, and the rule would be the link between them. An action can also be invoked via the command line, from another function, or from the web.

Actions receive a JSON object as a parameter and also return a JSON object as result. Actions can also be chained into a sequence of functions or call other services' actions as a middleware.

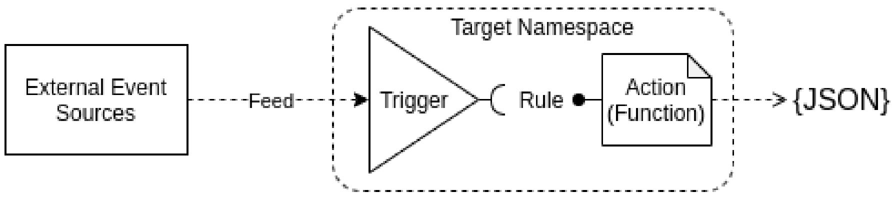

Figure 2.3.: OpenWhisk Programming Model

Architecture

OpenWhisk under the hood is "built on the shoulders of giants," and it uses some widely known and well-developed open source projects: **Nginx**[7], a high-performance web server and reverse proxy used to implement support for the HTTPS secure web protocol; **CouchDB**[8], a scalable, document-oriented NoSQL database used to store configuration and functions' results; and, **Kafka**[9], a distributed, high-performing publish/subscribe messaging system used to buffer and persist messages exchanged between controllers and invokers; [28]

It also has its custom components: **Controller**, a Scala-based implementation of the actual REST API that serves as the interface for everything a user can do, including CRUD requests and invocation of actions. It also manages authentication and authorization. The **Invoker** is the heart of OpenWhisk. The Invoker's duty is to invoke an action. It is also implemented in Scala. To execute actions in an isolated and safe way it uses Docker. [19]

[7]https://www.nginx.com/
[8]https://couchdb.apache.org/
[9]https://kafka.apache.org/

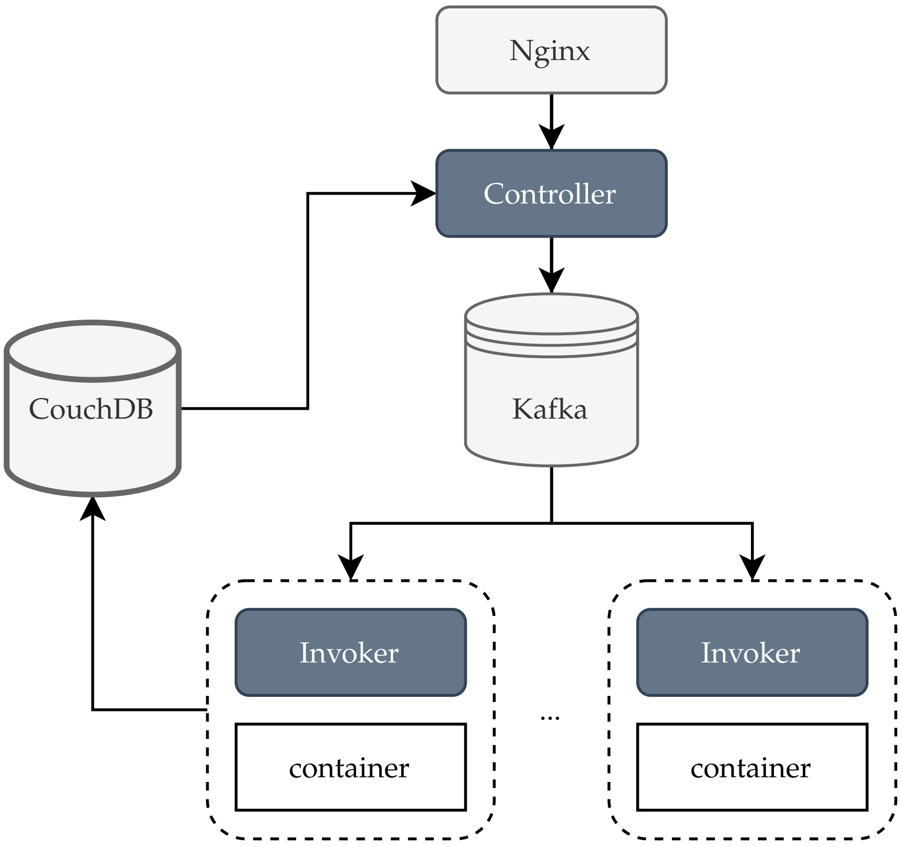

Figure 2.4.: OpenWhisk Architecture

Execution Flow

First of all, the function is invoked via the command line or triggered by an event. This will create a client request that will be handled by the reverse proxy (Nginx). Once done with the TLS, the request is forwarded to the controller that identifies the request and checks the authentication and authorization of the caller in the **subjects** database (CouchDB). If the caller has the rights to perform that action, the controller will, then, load the action from another table in the **whisks** database with the default parameters and merge them with the ones passed in the request. The controller will lookup for invokers available (acting as a loadbalancer) and put a message into the queue (Kafka). This is done asynchronously so that if it crashes, the messages are not lost.

At this point, if the user did not choose a synchronous call (*–result* flag) an *ActivationId* will be returned and could be used later to get the result of the action.

Proceeding with the invoker, it will pick up the message from the queue and run a container with the respective environment. It will copy the files to the container and let it execute the function. Once done, the result will be copied to the **activations** database (CouchDB again) together with the logs generated by the container. From this point on, the results can be retrieved with the *ActivationId* or, if synchronous, will be returned to the caller.

2.4. Cloud Automation Tools

2.4.1. Terraform

Terraform allows infrastructure to be expressed as code in a simple, human-readable language called HashiCorp Configuration Language (HCL) [20]. As a result, the code can be versioned in a Version Control System (VCS) and track any changes made to it. Breaking changes can also be easily rolled back using the previous version.

The OpenStack[10] cloud provider is used to set up the infrastructure whose resources are provided by Leibniz-Rechenzentrum (LRZ).

Modules

The provisioned infrastructure is divided into modules. A module is a container for multiple resources that are used together. Modules can be used to create lightweight abstractions, so that you can describe your infrastructure in terms of its architecture, rather than directly in terms of physical objects.[10]

In the repository layout, for example, a module *compute* can be found and it comprises not only the *openstack_compute_instance_v2* used to create an instance but also *openstack_networking_floatingip_v2* and *openstack_compute_floatingip_associate_v2* used to create a floating IP and associate it to the instance.

A module has three main components: input variables, to accept values from the calling module; output values, to return results to the calling module; and resources, to define one or more infrastructure objects that the module will manage; they are named *variables.tf*, *output.tf*, and *main.tf*, respectively, in the repository. The extension *.tf* represents an HCL file.

In the root directory, the *main.tf* file creates clusters (using the cluster module), and the cluster module uses the compute module for the master and worker nodes, as well as the network module. The default values are set in the *variables.tf* file but can be set directly in the *main.tf* file. The *output.tf* file is used to output information about the resources created, in this case: name, id, internal IP, and external IP of the instances.

[10]https://www.openstack.org/

Templates

As variables are used to configure the cluster, the values are not known beforehand. Also, dynamic values are generated such as IP addresses and, thus, also the configuration files have to be generated dynamically. Two templates are used to create files: *group_vars.tpl*, used to create configuration files with a set of variables that will be used for every host within that group; and *hosts.tpl*, used to create inventory files which contains the definition of groups and its members.

```
pod_subnet: "${pod_subnet}"
cluster_name: "${cluster_name}"
```

Listing 4: group_vars.tpl

The values for the variables between curly braces come from the *main.tf* in the root directory and have the default value set in the *variables.tf* also in the root directory.

Resources

A few resources play an important role while integrating with Ansible. While Terraform *openstack_compute_instance_v2* provides *user_data* as an option to input data such as a script, it is only used here to set up the hostname and fully qualified domain name (FQDN) of the machine, as the use case demands a more robust tool for that.

The **local_file** resource is used in combination with the *templatefile* function to create dynamic files that will be used by ansible. Given the template example in Listing 4, here is the code using it:

```
resource "local_file" "group_vars" {
  content = templatefile("./templates/group_vars.tpl",
    {
      pod_subnet   = var.pod_subnet
      cluster_name = var.cluster_name
    }
  )
  filename = "./ansible/group_vars/${var.cluster_name}"
}
```

Listing 5: local_file resource example

Another important resource is the **null_resource** that enables running provisioners that are not directly associated with a specific resource. The two provisioners that make it possible to use ansible with Terraform are *remote-exec* and *local-exec*.

The *remote-exec* provisioner is used to establish an ssh connection to the instance; thus, it awaits until the instance is ready to receive any command. The *local-exec*, then, can be used to run commands in the instance. In this case, we use it to run the Ansible playbooks.

```
resource "null_resource" "ansible_master" {
  count = var.master_count
  triggers = {
    master_instance_id = module.master.instances[count.index].id
  }

  provisioner "remote-exec" {
    inline = ["#Connected"]

    connection {
      user        = var.instance_user
      host        = module.master.instances[count.index].floating_ip
      private_key = file(var.ssh_key_file)
      agent       = "true"
    }
  }

  provisioner "local-exec" {
    command = <<EOT
    cd ansible;
    ansible-playbook -i ${var.cluster_name}_hosts.ini master.yml
    EOT
  }
}
```

Listing 6: null_resource ansible_master

Listing 6 shows the resource that is triggered every time a new master instance is created for the cluster, more specifically, a new id triggers the provisioners.

2.4.2. Ansible

Ansible is an IT automation tool. It can configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates. [1]

In this project, Ansible is mainly used to create a Kubernetes cluster via *kubeadm* - a tool built to provide best-practice "fast paths" for creating Kubernetes clusters - and deploy applications into it during setup. The master node also plays the role of an NFS server and share a directory with all the worker nodes. This is also done via Ansible.

Modules

Modules (also referred to as "task plugins" or "library plugins") are discrete units of code that can be used from the command line or in a playbook task. Ansible executes each module, usually on the remote target node, and collects return values. [22]

The following is a simple example of the module ping executed from the command line:

```
ansible all -m ping -i cloud_hosts.ini
```

Listing 7: Ansible Module (Ping)

The return values from the previous command were executed in a cluster with one master node and one worker node is the following:

```
cloud-worker-01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
cloud-master-01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Listing 8: Ansible Module (Ping) - Return Values

Inventory

Ansible works against multiple managed nodes or "hosts" in your infrastructure at the same time, using a list or group of lists known as inventory. Once your inventory is defined, you use patterns to select the hosts or groups you want Ansible to run against. [21]

The flag -i specifies the inventory host path that is dynamically created by Terraform and looks like:

```
[cloud:children]
master
worker

[master]
cloud-master-01 ansible_host=a.b.c.d

[worker]
cloud-worker-01 ansible_host=w.x.y.z
```

Listing 9: Ansible Inventory File

In this inventory file, there are four groups: *all* that is a default group and does not need to be specified, *master* and *worker* for the respective roles they play in the Kubernetes cluster, and *cloud* that group them together for common tasks to both of them.

Playbooks

Playbooks are the basis for really simple configuration management and multi-machine deployment system, they can also orchestrate steps of any manual ordered process, they can launch tasks synchronously or asynchronously. [2]

Referring to Listing 7, the same could be written as:

```
- hosts: all
  tasks:
    - name: Ping
      ping:
```

Listing 10: Ansible Playbook (Ping) - File

and run as:

```
ansible-playbook -i cloud_hosts.ini ping.yml
```

Listing 11: Ansible Playbook (Ping) - CLI

Of course, it would only make sense to create a file in case there are multiple tasks to be run. That is why *steps of any manual ordered process* are part of its description.

Roles

Roles are ways of automatically loading certain vars_files, tasks, and handlers based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users. [27]

One can also tag the role and pass extra variables to it. The structure can be seen in Listing 13 for the *k8s-master* role. The other roles have a similar structure.

```
- hosts: all
  roles:
    - ping_role
```

Listing 12: Ansible Role (Ping) - File

The *ping_role* role would have all the tasks, variables, files, etc. encapsulated. This is similar to the concept of modules in Terraform, but should not be confused with modules in Ansible. It would be executed with the same command in Listing 11.

The roles *common* and *docker* are executed in all the instances; *k8s-master*, *nfs-server*, and *openwhisk* in the master node; and *k8s-worker*, and *nfs-client* in the worker nodes.

```
.
├── ansible
│   ├── ansible.cfg
│   ├── common.yml
│   ├── functions.yml
│   ├── master.yml
│   ├── roles
│   │   └── k8s-master
│   │       ├── files
│   │       ├── tasks
│   │       ├── templates
│   │       └── vars
│   └── worker.yml
├── config.yaml
├── daemon
├── dafs.py
├── graphs
│   ├── graphs_influx.py
│   ├── graphs_summary.py
│   └── k6cp.sh
├── influxbkp
├── lafs.py
├── main.tf
├── modules
│   └── cluster
│       ├── main.tf
│       ├── output.tf
│       └── variables.tf
├── script.js
├── sequence_diagrams
├── templates
│   ├── group_vars.tpl
│   └── hosts.tpl
├── output.tf
├── README.md
└── variables.tf
```

Listing 13: Repository Partial Layout

2.5. Containers

As containers are the main units for executing functions, it is also important to trace its origins, features, and the reason why they are being used instead of VMs.

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. [33]

Early on, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. As a solution, virtualization was introduced. It allows you to run multiple VMs on a single physical server's CPU. *Virtualization* allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application. *Containers* are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more. [32] Nonetheless, containers offer the control and isolation of VMs with the performance of bare metal. [13]

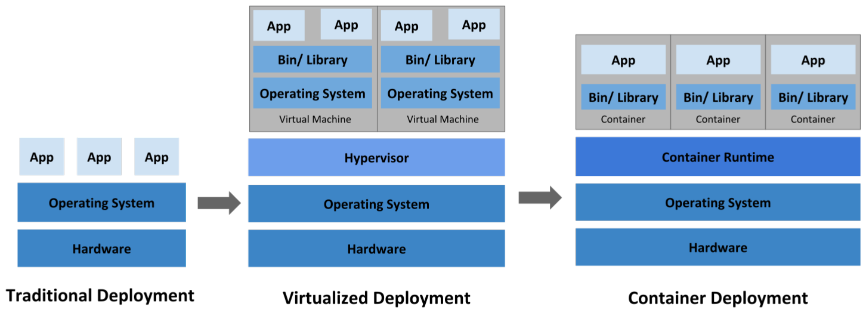

Figure 2.5.: Container Evolution [32]

As depicted in Figure 2.5, a hypervisor - also called Virtual Machine Monitor (VMM) - controls the distribution of real resources among the VMs in the **Virtualized Deployment**. There are two types of hypervisors: (1) Bare-metal hypervisor, and (2) Hosted hypervisor. The former, also called Type-I Virtualization sits on top of the hardware while the latter is loaded on top of the OS. In spite of the hypervisor's type, each VM will have its own OS (also known as Guest OS).

Similarly, for the **Container Deployment**, the *container runtime* is the software that executes containers and manages container images on a node. Containers share the same operating system kernel and isolate the application processes from the rest of the system; thus, they must be compatible with the underlying system. [34]

There is also an effort nowadays to bring minimal OSes for running containerized workloads securely and at scale. They are purpose-specific and should be considered as they come out of the box with automation and container orchestration built-in. Fedora CoreOS[11] is an example of such a system.

An important note here is that both systems (hypervisors and container runtimes) manage the underlying resources and; thus, the VMs and containers, respectively, should have their resources limited so that the issue faced in the **Traditional Deployment** is not carried forward. It means that one instance of virtualization could still take over the system and starve other instances if not managed properly.

2.5.1. Docker Architecture

Docker[12] created the industry standard for containers, so they could be portable anywhere. It also open sourced *libcontainer*[13] and partnered with a worldwide community of contributors to further its development. In June 2015, Docker donated the container image specification and runtime code now known as *runc*[14], to the Open Container Initiative (OCI) to help establish standardization as the container ecosystem grows and matures. [33]

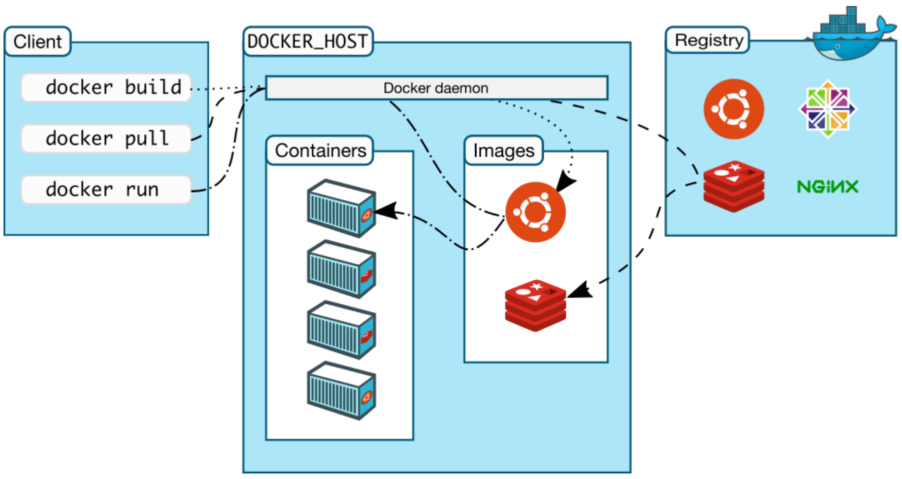

Figure 2.6.: Docker Architecture [11]

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. [11] The client can run in the same machine as the daemon.

[11]https://getfedora.org/coreos
[12]https://www.docker.com/
[13]https://github.com/opencontainers/runc/tree/master/libcontainer
[14]https://github.com/opencontainers/runc

A container can be seen as a running image. An image is a file consisting of a series of layers that contain the application, its dependencies, tools, all that is necessary for running the application; therefore, being reproducible in whatever compatible system it is running. It can be created from a *Dockerfile* that is a file with a set of instructions for building the image. They are stored in private and public registries.

In Figure 2.6 a simple flow for running a container is depicted. (1) the image is built (docker build), in the case of the Ubuntu image; or pulled from a registry to the host's internal registry (docker pull), in the case of the Redis image. (2) the container is run (docker run). Notice that if the image is stored in a public registry, when running the container, it will automatically look for the image in the internal registry and, if not found, will pull the image from the registry skipping step 1.

2.6. Kubernetes

Given the flexibility, security, isolation, and light-weight of containers, it is easy to run tens or hundreds of them in a single server. Then, it comes to the hard work of managing all of them and that is the main purpose of Kubernetes.

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. [32]

We use Kubernetes in this work to host the OpenWhisk clusters that run the functions. It is also used by the FaaST tool to host one of its components. Kubernetes is the main orchestrator regarding containers nowadays and this gives the reason to take a closer look at its components and functionalities.

2.6.1. Architecture

A Kubernetes cluster may consist of a single node running all of its components for testing purposes or a multi-node highly-available cluster with a set of machines for running the control-plane components, etcd key-value store, and worker nodes separately. In this research, a master node refers to a node consisting of the control-plane and etcd components. A node is also a reference to a server or VM that is part of the cluster.

Control Plane Components: A control plane makes global decisions about the cluster and compromises a few components: **kube-apiserver** exposes the Kubernetes API and is the front end for the Kubernetes control plane. **etcd** is a consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data. **kube-scheduler** watches for newly created Pods with no assigned node, and selects a node for them to run on. **kube-controller-**

manager runs controller processes such as Node, Replication, Endpoints, and Service Account & Token controllers. And, finally, **cloud-controller-manager** embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that just interact with your cluster. [23]

Worker Components: Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet is an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. **kube-proxy** is a network proxy that also runs on each node in the cluster. [23]

It also has a container runtime responsible for running the containers and Docker is the one used in this work.

Addons: Addons are also an important part of the cluster as they provide cluster-level features. One is actually required, that is, a Domain Name System (DNS) server. The other addon that provides an overview of the cluster and is used for management and troubleshooting is the Web UI (Dashboard).

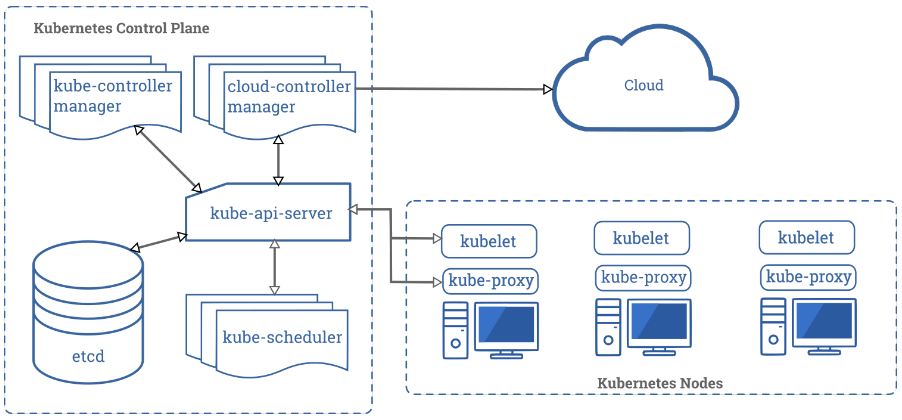

Figure 2.7.: Kubernetes Architecture [23]

Figure 2.7 shows the components of a Kubernetes cluster as well as their relations. If the cluster could be compared to a body, the *kube-api-server* would be the heart as all calls go through it. It is also the only component directly connected to *etcd*. And the *kube-controller-manager* would be the brain is it acts as a control loop that watches over the state of the cluster. If anything is different than what it should be, the *kube-controller-manager* makes API calls via the *kube-api-server* to make sure the cluster is at the desired state.

Another component that is also important in the context of this work is the *cloud-controller-manager*. As the infrastructure is being provided by LRZ and it is running Openstack to manage the resources, this is the component responsible for creating volumes using Cinder[15], the block storage service in Openstack, for example.

2.6.2. Objects

Kubernetes objects (or resources) are persistent entities in the Kubernetes system that are used to represent the state of the cluster.

It was said previously that Kubernetes is used to manage containers; actually, the smallest deployable object in the Kubernetes object model is called *Pod*. A Pod may run a single or multiple containers and within a Pod they share the volumes that are mounted and the network space. It means that they can communicate using the volumes or calling *localhost*.

[15]https://docs.openstack.org/cinder/latest/

3. Related Work

Although serverless, viewed as FaaS, is quite a new topic, researchers have already put some effort into showing its history, perspective, challenges and open questions [6, 8, 14, 17]. In the field, there is also work on scheduling functions based on budged and latency [12, 16, 30], an auction-based approach [7], and even a new way of managing the cloud resources as a big machine [3]. Furthermore, the papers can be split into cloud-only solutions [30] and hybrid architectures (including Edge and intermediary nodes) [12, 7, 16, 30]. A mix of VMs and functions for cost optimization is also presented by Gunasekaran et al [16]. Finally, Lee et al [24] did an in-depth comparison using CPU, memory, and disk-intensive functions running in the major cloud providers to show the differences between serverless computing and virtual machines for cost efficiency and resource utilization. This section describes different approaches and usage of serverless functions in the field.

Baldini et al [6] pose some challenges that come with serverless functions: *cold start*, while being a key differentiator, the ability to scale to zero and not charge users by idle time also brings the drawback of not having the code ready to run, increasing the response time; *Hybrid cloud*, with serverless popularity, integration among services is expected but unlikely that a single platform will be able to serve all use cases; *state*, often required in real applications, is still obscure regarding its management. D. Bermbach et al. [8] add to that list the lack of *Edge services*; meaning that differently from the cloud, there is currently no way to really "rent" on-demand edge capacity, leaving that responsibility to the user. Nevertheless, cloud providers have already started providing software for such devices as Amazon Greegrass.[1]. The general problem of managing an increasing number of Edge devices by the user is pointed out as a *management effort*. Hellerstein et al [17] describe FaaS as a Data-Shipping Architecture in the sense that it still ships data to code rather than shipping code to data and see it as perhaps the biggest shortcoming of FaaS platforms. The approach of Fluid Code and Data Placement, described as *stepping forward to the future*, is the suggested solution to the problem previously mentioned by which the platform would physically colocate certain code and data. That is the approach we followed.

Extensive work has been done regarding optimization for cost and latency. Elgamal et

[1]https://aws.amazon.com/greengrass/

al [12] present an algorithm that optimizes the price of serverless applications in AWS Lambda by dealing with three main factors: function fusion, function placement, and memory configuration of serverless functions. In our work, we do not consider a chain of functions and the cost of transitioning from one function to another which would incur extra costs. We also consider moving the data, an aspect that was not considered in their scope. Gunasekaran et al [16] create a scalable and elastic control system that exploits both VMs and serverless functions to reduce cost and ensure Service Level Objective (SLO) for elastic web services. Our approach, however, does not consider VMs for running the applications. Suresh and Gandhi present a function-level scheduler designed to minimize provider resource costs while meeting customer performance requirements. They do so by profiling the application and estimating the CPU shares. It is intended to be an option to existing baselines. They use a cloud-only solution; thus, not taking into account Edge nodes and data movement.

Bermbach et al [7] have a very particular auction-based approach in which application developers bid on resources fog nodes to make a local decision about which functions to offload while maximizing revenue. It requires no centralized coordination and focuses on maximizing the earnings for the infrastructure provider. On the other hand, there is no guarantee for the user that its function will be executed. Our approach was not designed with scalability in mind, it has a central coordination point and focuses on the fast response to the user. It should also always succeeds.

A vision of functions as processes and the data-center as a big computer is presented by Al-Ali et al [3] in this new abstraction called ServerlessOS. It aims to support not only event-driven compute but more general applications. Data management could be beneficial in this case, but nothing is mentioned about Edge devices.

To the extend of our knowledge, there is no present work that takes into account an edge-cloud continuum environment, in a FaaS only multi-cluster approach that involves data management for better function placement.

4. Methodology

4.1. Approach

The approach we used to assess the impact of accessing data in different locations included the creation of two Kubernetes clusters: one simulating Edge devices and another the cloud itself, both clusters were running in LRZ datacenter (the cloud infrastructure provided to the experiment by the university).

The function used in the experiments is written in Python and does manipulations on images.

For building the data migration and placement strategy we created an abstraction layer on top of FaaS providers that assess data placement based on user input data and then decides in which cluster to run the function. This takes into account latency data from monitoring (Prometheus) and data location (Object Storage).

We divided the experiment into two components that handle different - but connected - problems: latency and data migration.

4.2. Latency Aware

We will refer to this component as the Latency Aware Function Scheduler (LAFS). Part of its operation depends on a daemon that is deployed into each Kubernetes cluster and is responsible for collecting the latency times of each function. It runs periodically (every minute) updating their values. The main component is triggered when a user wants to run a function.

Before using the tool, the configuration has to be filled with the information and credentials for the clusters. Then, we a user wants to run a function, the following flow takes place: (1) the user requests a function running the tool (2) the tool calculates what is the cluster the lowest latency (2.1) it gets the execution time of every daemon running in the Kubernetes cluster (2.2) it compares the times and chooses the one with the lowest latency (2.3) it schedules the function to that specific Kubernetes cluster (3) it gives the user back the output of the function. The flow can be seen in Figure 4.3

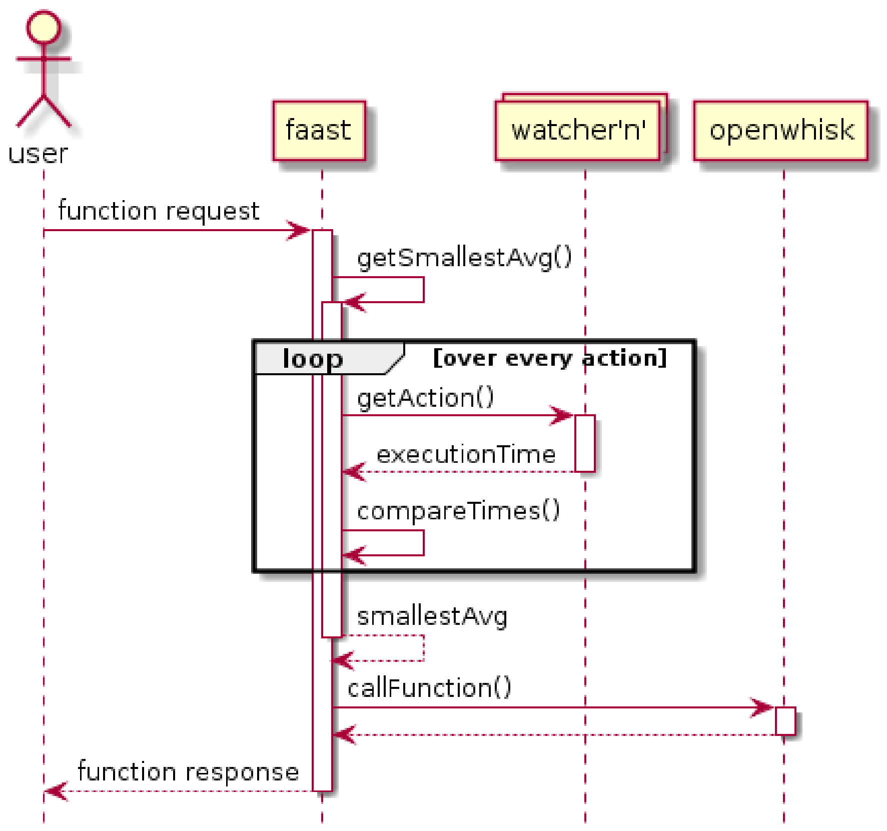

Figure 4.1.: LAFS Flow

As mentioned before, there is a daemon, also called watcher, that periodically gets all existing functions from OpenWhisk and, then, loops over them querying Prometheus for the response time timeseries of each in the last 15 minutes. It calculates the average time and updates the value that it keeps in memory (which is served to the main component). The flow of execution is depicted in Figure 4.2

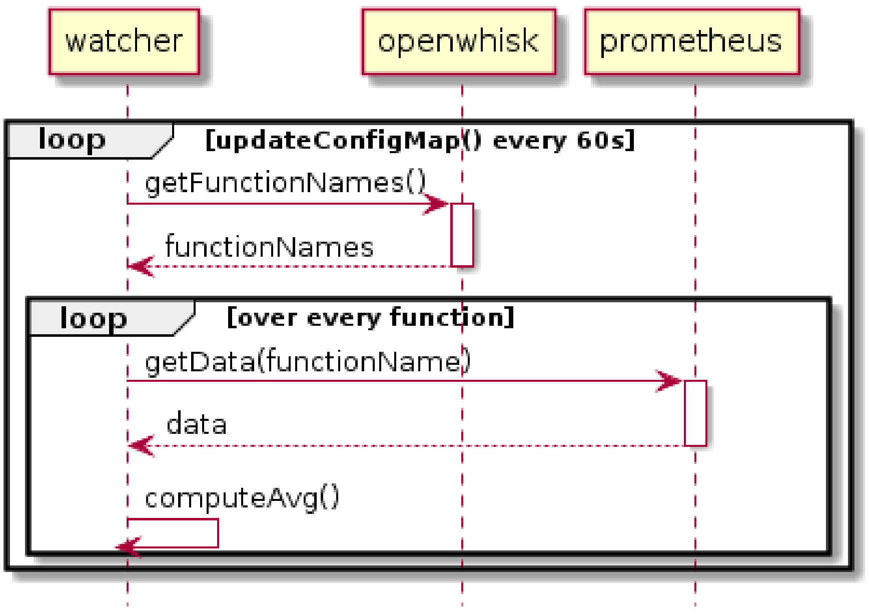

Figure 4.2.: Watcher Flow

4.3. Data Aware

The second component is referred to as the Data Aware Function Scheduler (DAFS). Different from LAFS, it is concerned with the location of the data that the function is going to use rather than the latency it takes to respond. One might think that the latency is related to the data location and it is indeed. However, this component does not take that into consideration when scheduling a function because it uses only one cluster and the decision is actually where to get the data from.

First of all, the component verifies if the payload of the function is present in the local object store - the user is responsible for determining which storage is the local one. The configuration uses *localBackend* and *remoteBackend* to differentiate the location of the cluster. If the data is present locally, it uses the *localBackend* as the location the data is going to be fetched from. If that is not the case, it schedules the function using the *remoteBackend* and triggers, in parallel, a process to copy the data used by the function to the local backend.

If, in the meantime, the user uses the tool again, it will verify that the data is still not copied and will still use the *remoteBackend*. Once it detects no missing data, it uses the *localBackend* instead.

The flow is shown in Figure 4.3. The stick arrowhead describes an asynchronous call.

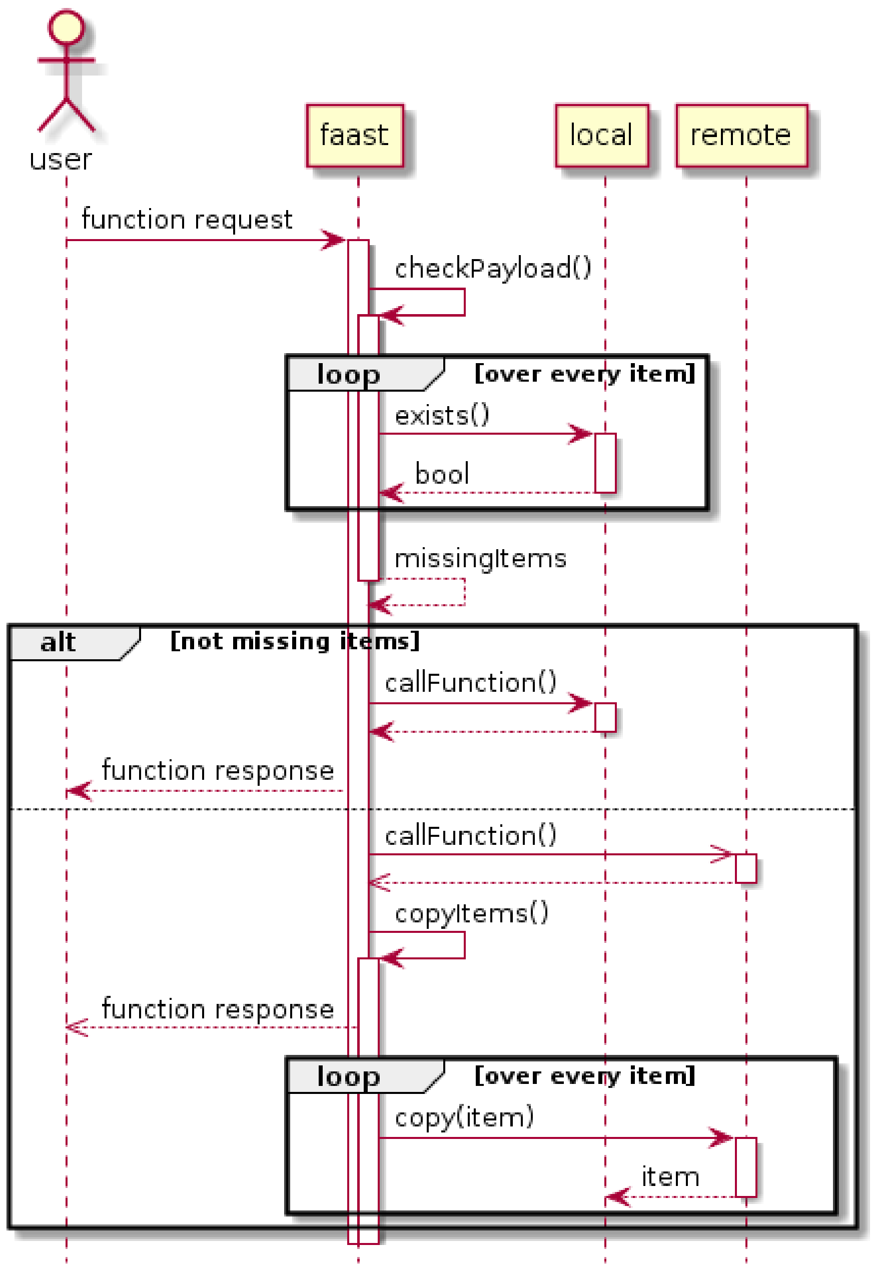

Figure 4.3.: DAFS Flow

5. FaaST

In this chapter, we are going to describe the FaaS Scheduler Tool (FaaST) in more details.

5.1. Overall Architecture

Currently, FaaST is implemented into two separate Python scripts: *lafs.py* and *dafs.py* corresponding to LAFS and DAFS, respectively, from the previous section. However, the configuration is the same one depicted in Listing 14. This means that having those three files, one could use it with any OpenWhisk cluster. Nevertheless, we not only created FaaST but also provided all the infrastructure from scratch for running it, including sample functions.

First of all, one has to clone the FaaST repository and install the required binaries: Ansible, Terraform, and Docker. Then, one has to create a private key to access the VMs, download the credentials of the cloud provider, and replace the values in the *variables.tf* accordingly. The configuration is prepared to run on OpenStack. The *main.tf* file is used to create multiple clusters, by default, only one cluster is created. The setup is automated and one can deploy it with only two commands: *terraform init*, for initializing the Terraform providers, and *terraform apply*, for deploying the infrastructure. The step-by-step can also be found in the *README.md* file in the repository.[1]

Once the clusters are up and running, the architecture should be similar to Figure 5.1. The diagram also includes an external object storage that is not deployed with the infrastructure and is used as a remote resource.

[1]https://github.com/possani/FaaST

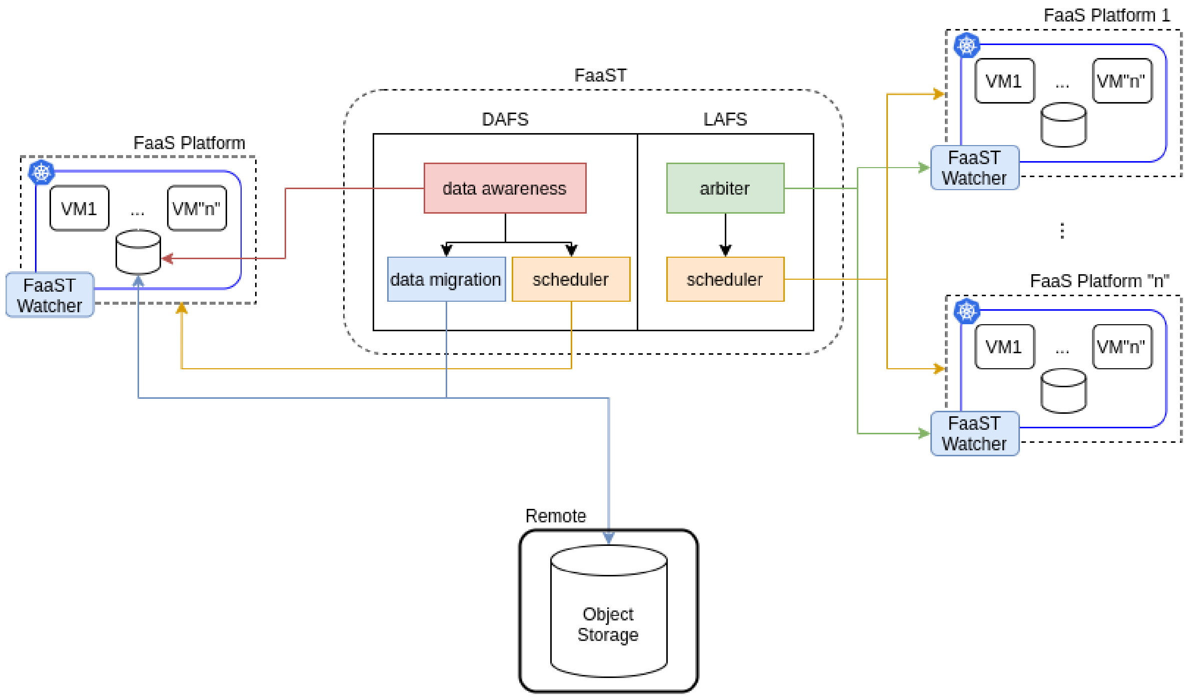

Figure 5.1.: FaaST Architecture

5.1.1. Input

The input for FaaST is a configuration file with the following components: *function*, the function name to be used; *payload*, the payload of the function; *clusters*, an array of clusters where the functions can be deployed to. Those are pieces of information used by LAFS and DAFS. The following ones are specific to the latter: *localBackend*, the address and credentials of the target cluster, the one where the data will be copied to; and, finally, *remoteBackend*, the address and credentials of the source cluster, where the data is originally from.

As three sample functions are deployed with the infrastructure, they are also included in the configuration file, but are commented out. The *minio* function does image transformations on multiple images, the *mongodb* function creates an entry in a collection called *users*, and the *pv* function creates a thumbnail out of a sample image stored in a mount volume in the host.

```yaml
function: minio
payload:
  images:
    - "sample000.png"
  service_ip: "<ip>"
  service_port: <port>
  access_key: "<key>"
  secret_key: "<secret>"

# function: mongodb
# payload:
#   connection_string: mongodb://<user>:<password>@<host>/<database>
#   name: "<username>"

# function: pv
# payload:
#   image: sample.png

clusters:
  - name: <cluster-name>
    watcher: <ip>:<port>
    openwhisk: <ip>

## DAFS MinIO
localBackend:
  service_ip: "<ip>"
  service_port: <port>
  access_key: "<key>"
  secret_key: "<secret>"
remoteBackend:
  service_ip: "<ip>"
  service_port: <port>
  access_key: "<key>"
  secret_key: "<secret>"
```

Listing 14: FaaST Configuration File

5.1.2. Watcher

As seen on Figure 4.2, the watcher updates the functions' values periodically. To be able to communicate with LAFS, we create that functionality in a separate thread so that the main programs serves that information as a web server.

The service is of type NodePort[2] and should be exposed on port 30004. One could alternatively forward the local port in a separate shell and use localhost instead. Here is the example when running a single cluster:

```
export
↪ KUBECONFIG=ansible/from_remote/local-master-01/etc/kubernetes/admin.conf
kubectl -n openwhisk port-forward deploy/watcher 8080
```

Listing 15: Kubernetes Port Forward

Now, if one *curl* localhost:8080, one should get an array with all the functions and response times.

```
[{"name": "pv", "avg": 75.00000000000001}, {"name": "mongodb", "avg":
↪ 30058.999999999996}, {"name": "minio", "avg": 168.99999999441206}]
```

Listing 16: Watcher Response

One could also specify the name of the function (action in OpenWhisk vocabulary) using a query parameter: curl localhost:8080?action=minio. In this case, one would only get the specific object.

5.1.3. Sub-components

The sub-components shown on Figure 5.1 with different colors are an abstraction given their single responsibilities, but are actually implemented using separate functions.

For example, the *data awareness* sub-component is implemented in the *checkLocal* function using the minio library by stating each file of the payload checking its existence in the object storage.

[2]https://v1-16.docs.kubernetes.io/docs/concepts/services-networking/service/#nodeport

The *missingStr* is, then, used by the *data migration* component, as a regular expression, that spawns a docker container for copying the images. We use a docker container here because the python library does not support multiple hosts for the same client and, thus, does not have *copy* or *mirror* operations for files between hosts. If we were two use only the python library, we would have to create multiple clients and add more overhead by copying the files locally before copying them to the remote location.

The next time DAFS is called, it will check again for missing images. If there is still any, it will use the remote location again, but, now, when checking for the existence of a container, it will not spawn a new one; rather, it will wait for the current one running to finish copying the images.

Once the images are present locally, it will run using the *localBackend*. The *scheduler* was sub-component responsible for deploying the function using the remote and local resources.

The *arbiter* sub-component (in DAFS) is the one that communicates with the *watcher* to get the avg response time of the functions and check what is the smallest one. The *watcher* initializes every single function with an avg of 0.0; this way, the *arbiter* will give them all a chance to run in a round-robin fashion. The *arbiter* will return a target cluster that will be used by the *scheduler* to deploy the function.

5.1.4. Visualizer

As many graphs would be generated during this work, another two scripts were created to automate the process. However, these ones were written in Python (not in BASH as the benchmark ones).

The *graphs_summary.py* function, parses the JSON files generated by K6, creates files with the data extracted and commands to generated a histogram using *gnuplot*[3]. If multiple runs were done, one can choose what type of aggregation to use with the files: max, min, or average. Multiple durations and cases can be selected generating a clustered graph. However, only a single function is supported at the moment. The results can be displayed in milliseconds or seconds and the same script is also used to clean up the automated files.

```
usage: graphs_summary.py [-h] [-f [FUNCTION]] [-m [METRIC]] [-sm [SUBMETRIC]]
 ↪  [--durations [DURATIONS]] [-c [CASE]] [--delete-in [DELETE_IN]] [-D] [-s]
 ↪  [-t [TESTS]]
```

The other script works in a different way, it gets the timestamp of the files and queries InfluxDB for the results in the specified duration. It also creates files with the data and

[3]http://www.gnuplot.info/

gnuplot commands to generate PNG files as outputs. With this script, one also has the option to display the 90th percentile of the HTTP request duration. Another difference is that this script does not use a histogram, but lines to show the values. Furthermore, the user can define the number of seconds to aggregate the results, making the curve smoother or sharper.

```
usage: graphs_influx.py [-h] [--duration [DURATION]] [--filter [FILTER]]
 ↪  [--group-by [GROUP_BY]] [--case [CASE]] [--p90]
```

6. Experimental Configuration

6.1. Benchmark Functions

We used three different functions for the benchmarks. We wanted to measure the performance of them in different scenarios and, based on that, better implement our tool (FaaST).

6.1.1. MinIO

As one of the functions to test the difference between accessing a file whose backend is running in the local cluster or a remote cluster, a simple function to create a thumbnail out of an image stored in a block storage service (MinIO) was created. It had all values hardcoded, including a single image payload.

To be more flexible regarding the payload and backend options, as well as to simulate real case scenarios, we made all those components configurable and added more functionalities to the function. Instead of only creating the thumbnail, it would also flip and rotate the image; thus, creating more processing time.

Additionally, due to the limitations of the Python runtime available on OpenWhisk as well as the code size limit, we created a custom runtime using Docker to run the function. Pillow[1], an imaging library used for the image transformations, was the main reason for creating our own runtime.

A key feature of the new function is that it accepts multiple images as input. Furthermore, the credentials were removed from the code providing flexibility for multiple backends and more security.

The function returns a message with the number of images that were processed.

6.1.2. MongoDB

We also created a MongoDB function that receives a connection string, for flexibility when choosing the backend; and, a username that will be used to create an entry in the users collection. This function inserts the user in the collection and removes it. It returns the

[1]https://pypi.org/project/Pillow/

username inserted in the collection, the *id* that was generated, and the *delete count*. With that information, one can check if the function succeeded.

6.1.3. PV

A similar function to the MinIO one mentioned above was also created. It has actually the same functionality but uses a Persistent Volume (PV) instead of object storage. The goal was to see how fast the images could be retrieved compared to the function using an external service to store the same files.

6.2. Infrastructure Settings

In order to measure the response time of the functions, each Kubernetes cluster comprised: an OpenWhisk cluster, for running the function; a MinIO object storage, for its data; a MongoDB instance, used by one of the function types; and an InfluxDB[2] database, for the data collected with the K6[3] (load testing) tool.

Additionally to the two clusters, we were also using two external object storages, one in the same cloud environment, and another in a different cloud provider which we named *remote*.

After creating the functions and deploying them on OpenWhisk, K6 called the functions using the endpoint exposed by OpenWhisk with different time intervals. At the end of each benchmark, a summary was generated and we used it to compare the latency in each environment.

6.2.1. Kubernetes

For doing the experiments and benchmarks two Kubernetes clusters were created from scratch using kubeadm[4], Ansible, and Terraform as described before. The idea was to have two clusters, one simulating the Edge with smaller machines, but the disk space required to run the Linux images available was 20 GB. This corresponds to an *lrz.medium* flavor and, in the end, we had two clusters running one master and one worker node with the following configuration:

For running the *pv* function, we had to create a shared file system for the Kubernetes clusters since we had a master and a worker node and the function could run in any of the two machines. So, we made the master node a Network File System (NFS), and the worker

[2]https://www.influxdata.com/
[3]https://k6.io/
[4]https://kubernetes.io/docs/reference/setup-tools/kubeadm/

Table 6.1.: Kubernetes Cluster Nodes

OS	Ubuntu 18.04 LTS Bionic
vCPUs	2
Disk	20 GB
RAM	9 GB
Nodes	2 (master and worker)

node an NFS client mounting a volume that connected to the server. This is all set up during the creating of the cluster with the Ansible roles *nfs-server* and *nfs-client*.

Additionally, to use the shared volume as a PV in the function, we installed an NFS Storage Class (nfs-client-provisioner[5]) using Helm[6]. This application uses an approach different from Hostpath in the sense that the Pod is not bound to a specific node. We also set this as the default storage due to the limitation of 4 volumes only when using Cinder. However, we still configured it in our cluster as an option.

Now, with the new Storage Class deployed, we could create Persistent Volume Claims (PVCs) in our deployments and the PVs would be created dynamically. However, OpenWhisk does not have an option to specify a PVC for a single function. What we can do is to modify the default Pod definition in the *whiskconfig.conf* file and include the declaration of the PVC there, and that is exactly what we did.

[5]https://github.com/helm/charts/tree/master/stable/nfs-client-provisioner
[6]https://helm.sh/

```
  kubernetes {
    pod-template = """---
apiVersion: "v1"
kind: "Pod"
metadata:
  annotations:
    test : "true"
spec:
  volumes:
    - name: function-pv
      persistentVolumeClaim:
        claimName: function-pvc
  containers:
    - name: "user-action"
      volumeMounts:
        - mountPath: "/files"
          name: function-pv
    """
  }
```

Listing 17: Custom Pod Template

The only issue with this approach is that all functions regardless of the necessity of using a PVC would have it mounted.

6.2.2. OpenWhisk

We had to tweak the default cluster settings that were initially too restrictive for our tests. Starting with the ingress, the annotation for the *proxy-body-size* was increased to accept Content-Length bigger than 50 MB in the client request. Proceeding to the limits, a number 10x bigger than the default configuration was in place to allow any desired benchmarks.

For the *invoker.containerFactory* option, *kubernetes* is set so that the actions would execute as Docker containers within Pods orchestrated by Kubernetes instead of being orchestrated by OpenWhisk using the Docker API.

A couple of metrics were also enabled to leverage the Prometheus exporter and the user metrics giving a better overview of the cluster state as well as information about action performance. Those metrics could be seen on Grafana that was also deployed with OpenWhisk.

```yaml
whisk:
  ingress:
    type: NodePort
    apiHostName: {{ hostvars[inventory_hostname].ansible_host }}
    apiHostPort: {{ node_port }}
    annotations:
      nginx.ingress.kubernetes.io/proxy-body-size: "200m"
  limits:
    actionsInvokesPerminute: 600
    actionsInvokesConcurrent: 300
    triggersFiresPerminute: 600
    actionsSequenceMaxlength: 500

nginx:
  httpsNodePort: {{ node_port }}

invoker:
  containerFactory:
    impl: "kubernetes"

metrics:
  prometheusEnabled: true
  userMetricsEnabled: true
```

Listing 18: Cluster Configuration

Another piece of configuration that is not part of the main configuration file is the timeout of the function that we set in the moment of its creation when running the respective roles. We are currently using 300 seconds, the maximum limit.

In order to make the functions compatible with OpenWhisk, they have to receive and return a dictionary (in the case of Python). Additionally, as it uses external packages (minio and PIL) in the case of MinIo, they also have to be packed into the same file used to create the function. The approach to solve this issue was to create a virtual environment, install the dependencies, and zip it all into a single file.

As this file can get bigger than 50 MB (the default value for the ingress annotation proxy-body-size), we had to change the settings previously mentioned. Otherwise, one might incur the error 413 *Request Entity Too Large*.

6.2.3. Test Initiator System

Table 6.2 shows the configuration of the computer that was used to run most of the benchmarks. Later, we had to use another machine, but it also had similar specifications.

Table 6.2.: Local Computer

OS	Ubuntu 20.04.1 LTS (Focal Fossa)
CPUs	8
Disk	256 GB
RAM	8 GB
Model	ThinkPad E485

6.3. Evaluation Scenarios

Figure 7.1 shows the first benchmark which led to the decision of using only one cluster. Apart from having similar results (due to the fact they were running in the same network), we also wanted to use the Object Storage external to the Kubernetes cluster, and this extra VM would exceed the limit of 4 VMs per namespace.

The next step would be to decide the scenarios we would test: (1) we should vary the duration of the benchmarks (2) we should vary the number of images processed (3) we should use different image sizes (4) we should test different approaches when copying the files locally. (3) and (4) do not apply to the mongodb function. Then, we decided that all tests would run for 1, 5, 15, and 30 minutes. The other points will be described below for the different functions.

6.3.1. MongoDB

The mongodb function ran against MongoDB instances deployed in different environments: locally, an instance was deployed on the Kubernetes cluster as an application; lrz, an instance was deployed using a container in a separate VM in the same cloud provider; and, remote, another instance was deployed in a VM running in a different cloud provider.

6.3.2. MinIO

The scenarios we tested for the minio function were four, with *variances*: **local**, the function would use the image stored locally; **remote**, the function would use the image stored remotely; **local copy**, the function would wait for the image to be copied from remote to local and then would run using it locally; and, **local copy in parallel**, the function would start running using the image located in the remote location and copy it in parallel, once done, the function would switch to run using the local images.

The *variances* were related to the number of images that would have to be copied: single image, intuitive as the name says; multiple images, we selected 10 images to be used; partial, we would copy 100 images, but only use 10; partial1k, we would copy 1000 images, but only use 10.

Table 6.3.: Test Scenarios

			remote				
		local					30
	local-cp					30	
local-cp-parallel					30		
Type\Duration (min)	1	5	15	30			
single							
multiple							
partial							
partial1k							

6.3.3. PV

The *pv* function only ran the local case, even though, using a PV deployed using NFS means that the files would be stored in the server. In our case, we only used one machine when testing it. So, the files were always locally present. Additionally, we ran with two variances, which we also called *cases*, *single* and *multiple* images. We could also test the *local-cp* and *local-cp-parallel*, but either we would have to copy the images using Secure Copy Protocol (SCP) from the user's machine, what would not mimic a real case; or we would have to copy from the *remote* location, which would end up being the same as the MinIO function.

6.4. Load Generation

For running the benchmarks a bash script was created for each case. These scripts would prepare the environment by exporting environment variables, copying images, and selecting the specific scenarios. Then, it would call K6 for load testing the different cases. And, finally, tear down what was previously created.

6.4.1. Scripts

Although there are 5 scripts for running the different functions and scenarios, they all follow a similar approach. They are configured to run using different durations and backends (Object Storages and Databases).

Before running any of the scripts, there is a file containing all the environment variables required. One has to first fill that file out with the respective values and, then, source it to be able to run the tests.

Mongodb function: The simpler script is *k6mdb.sh*, the one used to benchmark mongodb function. It starts by looping over the different database instances. We used an instance running locally and another on LRZ. Then, it loops over the different durations (1m, 5m, 15m, and 30m), export the specific variables for that scenario, and calls K6. After the test runs, it deletes the Pods created on OpenWhisk so that it has a cold start for all the durations.

Minio function: Additionally, to the elements mentioned for the mongodb function, this one also has different cases in which files are copied from *remote* to *local*.

Let us start with the simple cases, in which files are not copied, **local** and **remote**. It starts by deleting the content of the bucket using a docker container with MinIO client in it. Then, it copies the images to the bucket according to the type (single, multiple, etc) also using the same minio client container image. Afterward, it runs K6 that outputs the results as a JSON file. Finally, after deleting the Pods created by OpenWhisk, it empties the buckets again.

For the **local-cp** case, we also count the time it took for copying the files from *local* to *remote*. However, in the previous cases, it copied the images direct to either *local* or *remote*. Here, on the other hand, it copies the images to remote and, then, copy from *remote* to *local* counting that time together with the execution time.

Figure 6.1 depicts the flow to better understand all the steps. *Local* and *remote* refer to object storages. *OpenWhisk* and *InfluxDB* are running in the *Kubernetes* cluster.

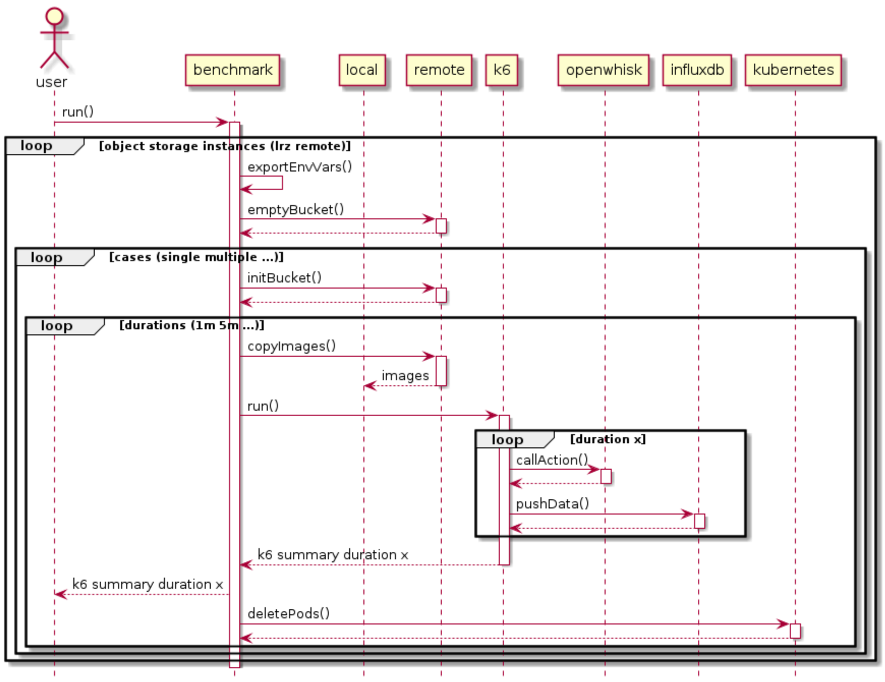

Figure 6.1.: K6 Benchmark Flow

Going one step forward, in the **local-cp-parallel** case, we start copying the images from *remote* in the background and, in parallel as the name implies, we start running the function using the resources available in the remote location. Then, a loop watches for the termination of the Pod. Once that happens, our script ends the K6 process that is running using *remote* resources, change the object storage location, calculates the remaining time left, and starts a new run with K6, now, using the local resources.

Finally, the **pv** function is simpler than the MinIO ones. It only uses the local resources and, thus, does not need to interact with other applications. The files are transferred using SCP and comprise only two cases: single and multiple. Once done, the files are also removed from the machine.

6.4.2. K6

K6 is an open-source performance tool for load testing. One has to create a JavaScript file that will contain the code and properties for the test and will be used as input to the tool.

Initially, we are using 10 Virtual Users (VUs), but due to the many timeouts from InfluxDB, we decided to use only 1. The number of requests was also similar in both cases, but the latter did not result in timeouts. We also used *insecureSkipTLSVerify* set to true as part of the options.

As for the url, we used the one exposed by OpenWhisk when creating a function. We let the server IP, function name, and payload as environment variables for reuse and flexibility.

We also set the *timeout* parameter of the HTTP request to 90 seconds (the default is 60 seconds). Additionally, we created an assert function, called *check*, to verify the number of successful requests. We did that by checking if the status code was 200.

7. Results

We started by comparing the response time of the functions when running in different clusters and using different backends.

In Figure 7.1, we can see the result of an experiment in which we ran K6 two times for 60 minutes using 10 VUs and averaged the results. We ran the experiment with both clusters (Edge and Cloud). As one can see, the results were very similar because both of them were running on the same cloud environment (LRZ). The idea was to simulate an Edge cluster running further away from the Cloud but we did not use a real Edge cluster. Given the limitations, we decided to use only one cluster from this point on.

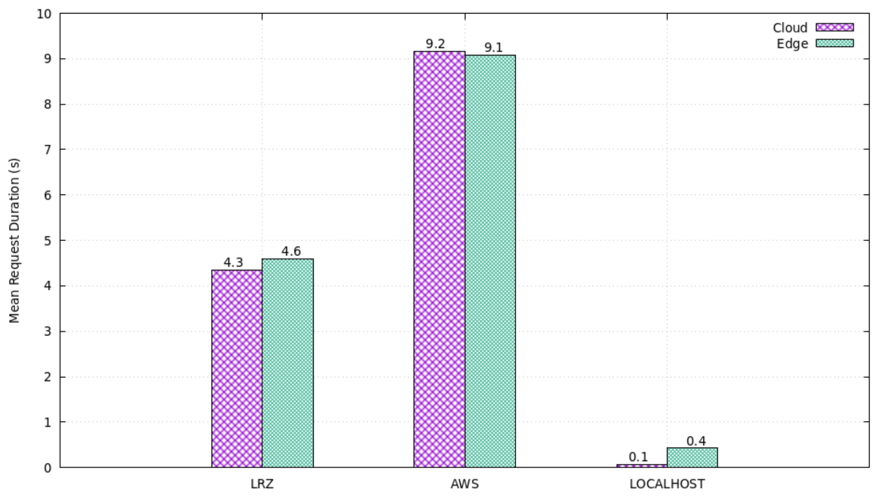

Figure 7.1.: Request Duration - MinIO

Nevertheless, more important than the time between the two clusters was the response time depending on the backend used. It is clear that the closer was the resources to the cluster where the function was running the better were the results. Localhost was the fastest, using the resources within the cluster, followed by the other instance also running on LRZ but in a different cluster and, finally, the object storage on AWS.

7.1. Image Migration

We used a set of images ranging from 4 to 55 KB and another one ranging from 23.6 to 403.8 KB to measure the time taken to copy them from *remote* to *local* as this is an important factor in our test cases.

We created a script that ran 5 times and copied 1, 5, 10, 50, and 100 images each time. We used the minio client container to copy the images with the *–json* flag to return a parseable output. The JSON files contain the *transferred* amount of bytes and the *speed* (bytes per second). Thus, we could calculate the average time of the runs.

Figure 7.2 shows that using smaller or bigger files, the curve is of the type $ax + b = y$. It means that even using images of different sizes would still lead us to the same conclusions.

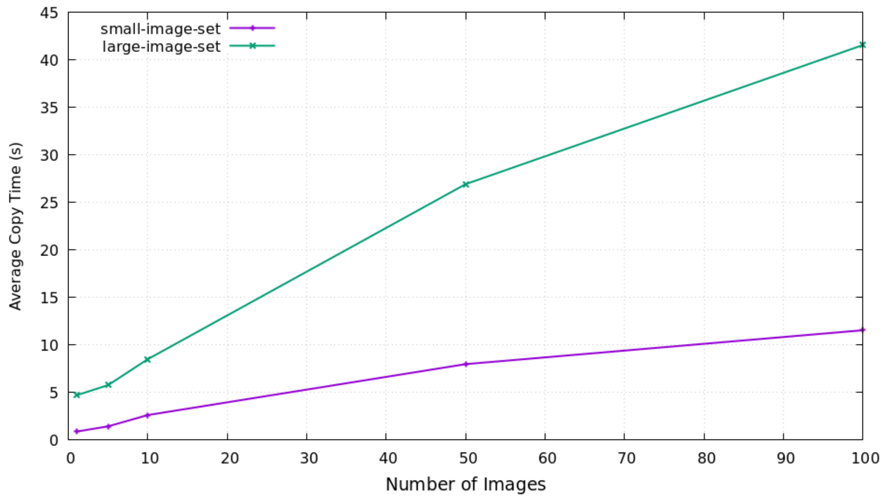

Figure 7.2.: MinIO Copy Time

7.2. Cold Start

An important piece of information about all the execution times is that it includes the cold start time. Additionally, in the case of *local-cp*, it also involves the startup time of the container and the copy time. *local-cp-parallel* also has the same elements; however, it hides those times while executing using the remote files. To better exemplify what was just said, Figure 7.3 highlights each of those phases with a different color.

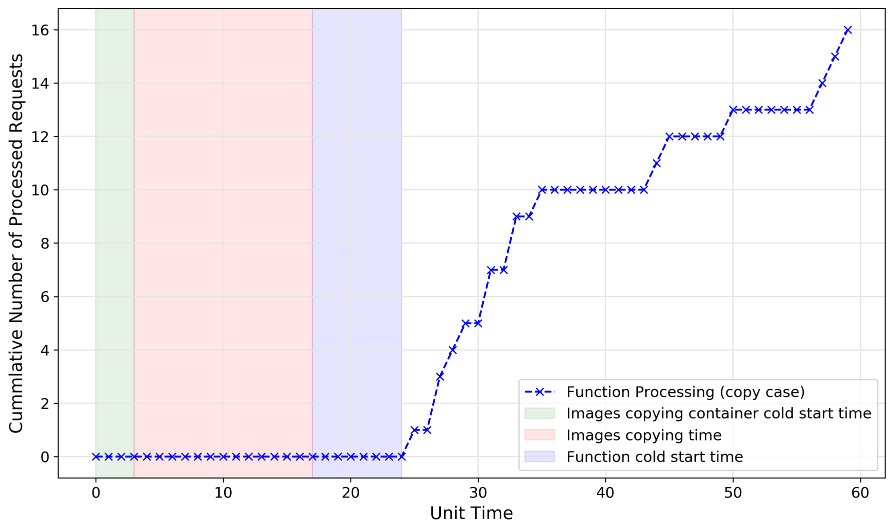

Figure 7.3.: Cold Start

The spawning of a new container in the local machine with the configuration described in Table 6.2 to copy the images takes around 3 seconds. It is very fast compared to spawning VMs but still takes a considerable time if compared to the execution time of a function. The cold start time of the container in the OpenWhisk cluster is not present for warm containers that are usually kept up for some time to avoid this problem. However, we did not consider warm containers when our experiments.

7.3. Initial Base Performance

7.3.1. Image Retrieval

The pv function was another function we tested during our experiments. Given the same functionality, regarding image transformation, as the minio function, a good comparison would be in terms of request duration time. The pv function, differently from the minio one, does not talk to another application to get the images, it gets them right from the persistent volume. This slight difference should make a significant impact on the response time.

As one can see in Figure 7.4, pv was compared to minio cases where there was no copying of images. *minio-local* and *minio-lrz* had very similar results because the services were hosted in the same cloud provider. For the *remote* case, however, the response time was way higher. pv, on the other hand, had the best response time, it happened because there was no overhead of communication with other applications.

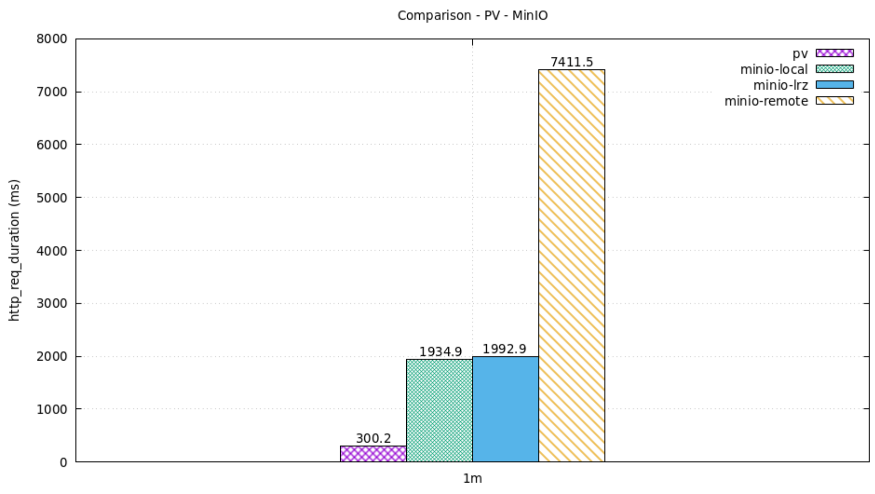

Figure 7.4.: PV MinIO Comparison

7.3.2. Database Records Insertion and Deletion

The mongodb function results comparing *local*, *lrz*, and *remote* are displayed in Figure 7.5. One can see that as the duration time of the tests increases, the response times tend to stabilize. The tests running for 15 and 30 minutes have very close results. Additionally, one can also observe that the response time for the local instance of MongoDB is slightly smaller, thus faster than the other one also running on LRZ, but in a different VM.

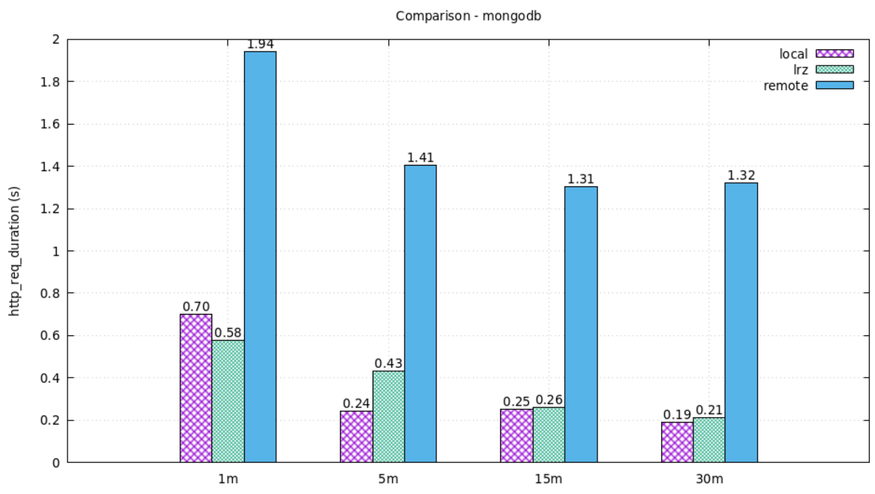

Figure 7.5.: MongoDB - Request Duration

Although the local instance has only a slight difference in time, the longer the test runs the bigger is the difference in the number of requests. Figure 7.6 shows the impact a long test can have when the function is running close to the database.

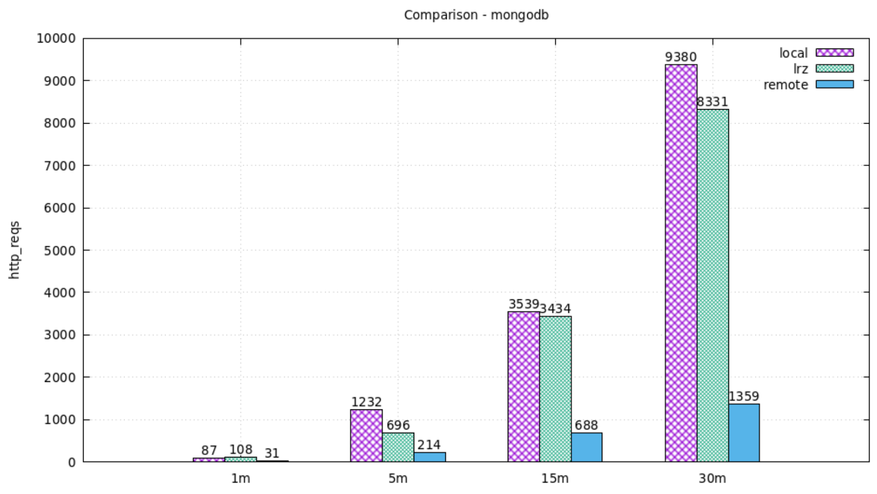

Figure 7.6.: MongoDB Number of Requests

We can also see how the requests accumulate over time for the 5 minutes test in Figure 7.7

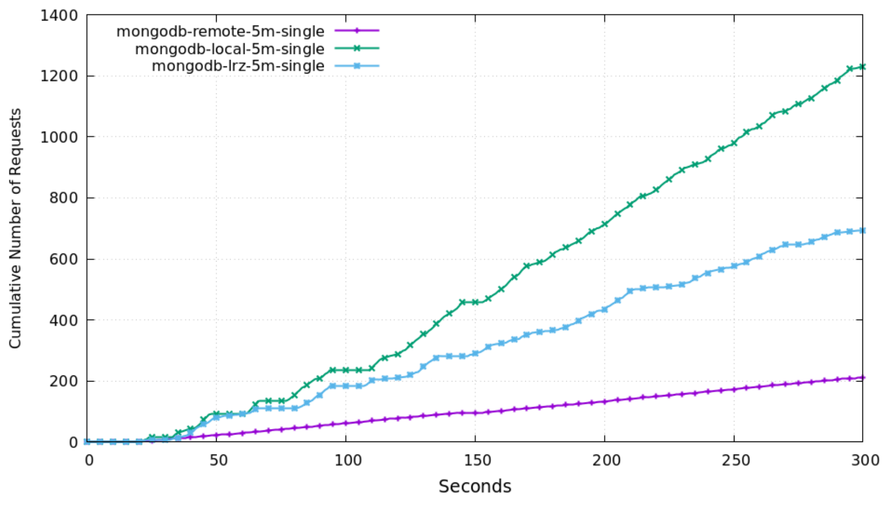

Figure 7.7.: MongoDB Cumulative - 5 min

7.4. Data Migration Performance

At this point, after all the tests have been performed, the data is saved on InfluxDB and can be queried to generate the graphs. We also have the summary files from K6 and JSON output files from the minio client.

The only cases that were not possible to get results were *local-cp-parallel* and *local-cp* of partial1k because the copy time takes over 2 minutes to run.

Figure 7.8 displays the cumulative number of requests in the period of 1 minute. Given the short time of the experiment, it is easy to see on the x-axis how long the different cases took to start. *local* and *remote* should start first as there is no copy process involved for the request and that is what it shows. However, the slope of the curve is different due to the latency of the remote backend. Thus, the number of requests in the local scenario is higher. Nearly at the same time, *local-cp-parallel* should start as it does not wait for the copying process to finish before starting. Finally, *local-cp* should start because, in this case, copying of the file is taking place. As this scenario only involves one image, the difference in the starting time should not be significant.

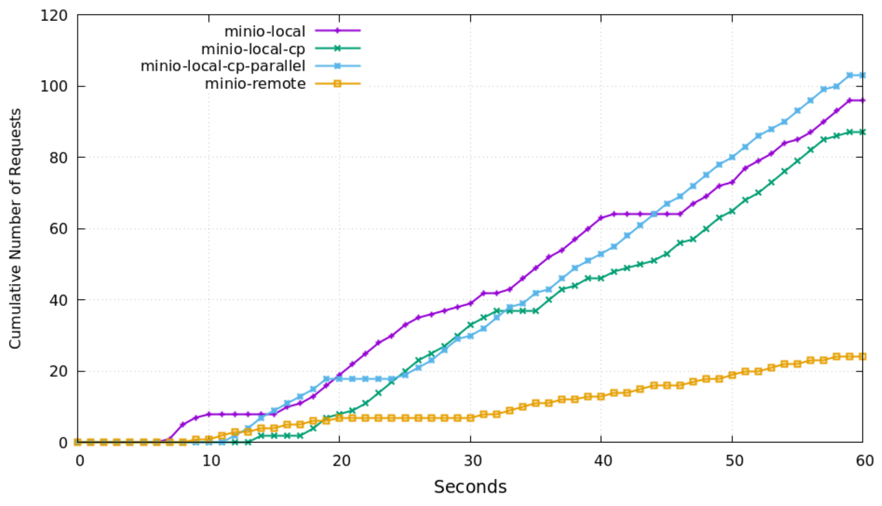

Figure 7.8.: Request Duration - MinIO - 1 min

As can be observed in Figure 7.8, the time it took to copy the file was less than 15 seconds. If we were to use the same case with any other duration, it would be very hard to see the results in the graph due to the bigger scale in the x-axis. For that reason, Figure 7.9 shows an experiment that ran for 5 minutes but with a thousand files being copied. As one would expect, the cases that do not wait for the copy to take place should start nearly at the same time and that is very clear in the graph. Additionally, due to the number of images, *local-cp* took more than 2 minutes (138 seconds) to start. Once again, we can see that the slope of the remote case is lower compared to the local ones. What is interesting about this particular graph is the *local-cp-parallel* case. During the copy phase, it is using the images in the remote location; once they are copied (after 107 seconds), it starts using them locally and the difference is remarkable.

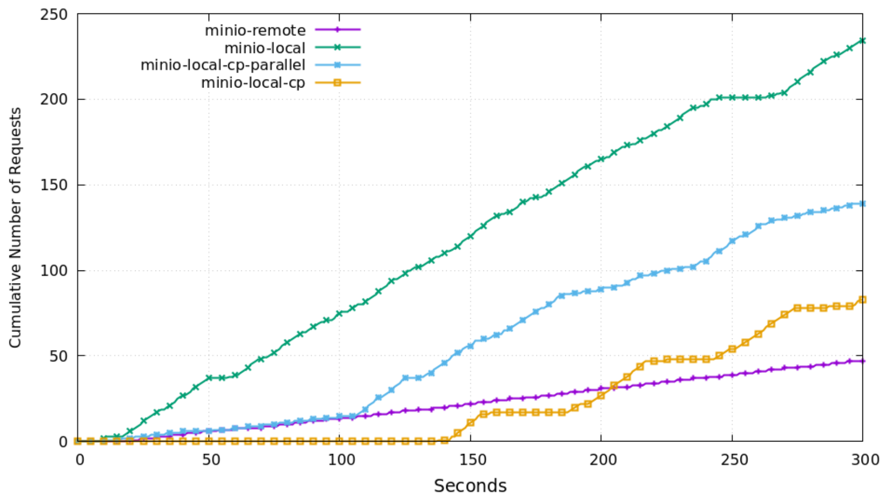

Figure 7.9.: Request Duration - MinIO - 5 min

Figure 7.10 corroborates that using the files locally is the best option. As this case uses and copies 10 files, there is no difference in the slope of the *local-cp-parallel*. Nevertheless, it shows that even in a long-running process, this option is slightly better than the local copy. Thus, we decided to use it for the DAFS component.

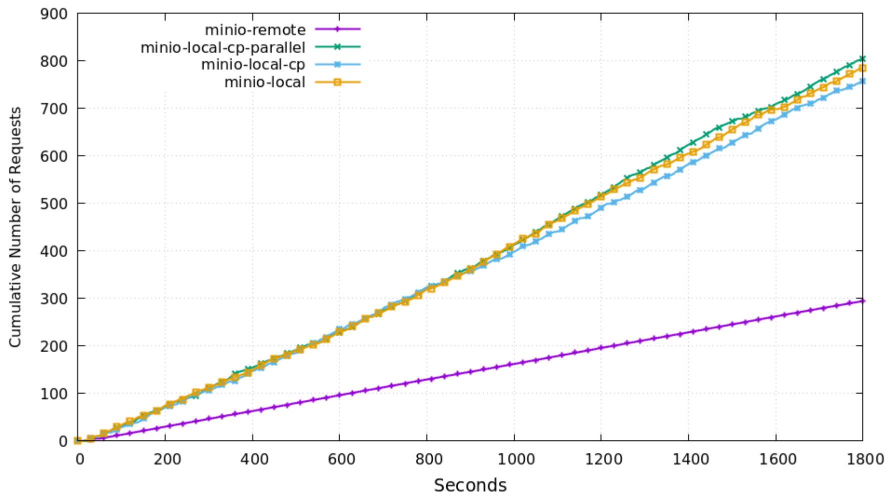

Figure 7.10.: Request Duration - MinIO - 30 min

All the test runs for the minio function comparing *remote*, *local*, *local-cp*, and *local-cp-parallel* test cases with durations 1, 5, 15, and 30 minutes can be found in Appendix A.

8. Discussion

During our experiments, we realized that there was no purpose in using two different clusters to measure the time when both of them were running in the same region. If the goal was to measure the latency in an Edge cluster, it would have to be in fact there. That is why we conduct the later experiments only using one of them. Additionally, we were using the clusters to also deploy the object store application when, in real case scenarios, those services are provided by the cloud provider. That is why we spawned an instance of MinIO and MongoDB in a separate VM to circumvent that problem.

Our tests endorsed the already known fact that using local files gives the best execution time for the functions. Furthermore, when there is no intermediary application for serving the files, as was the case for the pv function, the times are even better. One could clearly see the impact of another jump to connect and receive the images from MinIO, be it in the same cluster or not.

However, when files need to be copied from a remote location because they are not available locally, *local-cp-parallel* was the best choice compared to *local-cp*. While both should have the same copy time, *local-cp-parallel* hides it while running the function using remote files. Of course, the response time will not be the same as when the files are present locally - and for that reason, we switch to the local mode once the copying process is done, but in the meantime, the users will still get a response. In fact, the bigger the amount and size of the files, the better *local-cp-parallel* is compared to *local-cp*.

Regarding the mongodb function, we evaluated its performance using MongoDB instances in physically different places. We initially planned to do experiments similar to copying files, but one could query a single collection or multiple collections. To keep the data in sync, it would be better to copy the whole database and, it should not be done right before the request. Ideally, there should be another component just doing the synchronization and, in this case, the "copy time" would not be included in the response time anyway.

For the pv function, we could also have used a minio client to fetch the data before running it. However, the copy time would be the same as the one for the minio function *local-cp* scenario and, the fact that it was using a persistent volume in this case, it would be better to have a different copy mechanism. We could copy the files from another cluster, but a secure connection should link both of them, a similar approach to Rancher Submariner[1] where the resources are reachable within the network. However, this approach would also limit the test to inter-cluster communications.

8.1. FaaST

Serverless computing has received more attention from the scientific community which has provided alternative scheduling options to the ones currently available by the cloud providers. FaaST is another alternative to give users better results when making use of such services.

One could start using the DAFS tool to copy the files to the local cluster and, once the disk was full due to the Edge device limitations, one could switch to the LAFS tool to get the best latency when using the same remote object storage.

However, this approach would still be using the "ship data to code" style. If we are using applications whose data is in the Edge such as the Internet of Things (IoT) applications, better use of our tool would be to start with LAFS instead. It would schedule the functions to the Edge (close to the data) and, when the device does not have enough resources, we would use DAFS to copy the data over to the cloud to continue processing over there.

Additionally, one also has to consider that when using a database or object storage in the same cluster, those applications will be consuming resources that could be used by other functions otherwise. In this case, a lower number of requests is expected or multiple numbers of nodes (more resources) would have to be used to compensate for the usage.

[1]https://submariner.io/

9. Conclusion and Future Work

9.1. Conclusion

This research aimed to identify the data-access performance in an edge-cloud continuum when using different scheduling patterns for functions varying the location of the data. Based on the function's response time and the data migration time, it can be concluded that functions using local files have faster response and, thus, they should be scheduled close to the data whenever possible. The results also indicate that the best approach to circumvent the remote file location is to copy them where the function is going to run, but to serve the user's requests by using the remote location in the meantime; thus, hiding the copy time and increasing the response time to the user.

It can also be seen that data access is faster when accessing files using persistent volumes over NFS than communicating with other applications for retrieval even when they are deployed in the same node. Additionally, the same behavior can be observed with the mongodb function that uses a database instead of object storage. The results are similar when comparing MongoDB instances running in the same region and cloud provider, but have a significant difference when running in a remote location.

9.2. Future Work

There are still open issues that are not addressed in our research and a few points to be improved in the FaaST. (1) The integration between the DAFS and LAFS could be done automatically instead of being triggered by the user. (2) the credentials for the backends could be stored in a more secure place than exposed in the configuration file. (3) The tool could also be FaaS Platform agnostic but was implemented using OpenWhisk as a parameter.

A. Appendix - MinIO Test Runs

A.1. 1 Minute

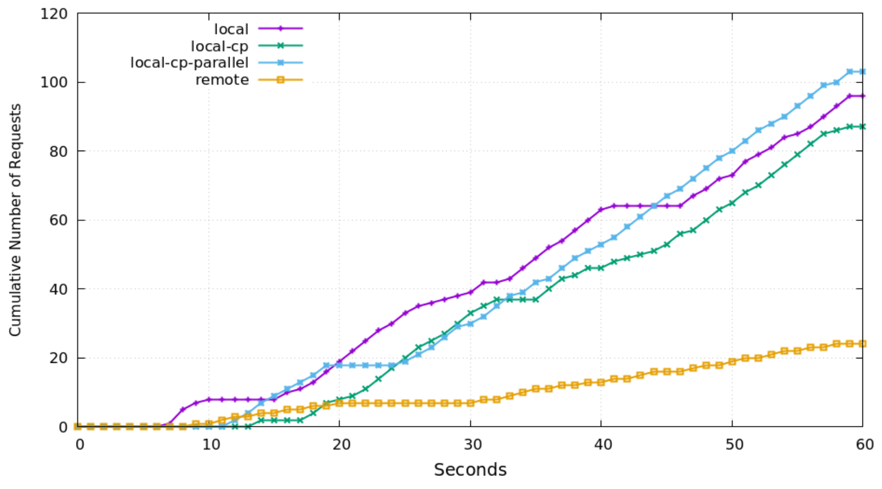

Figure A.1.: Single

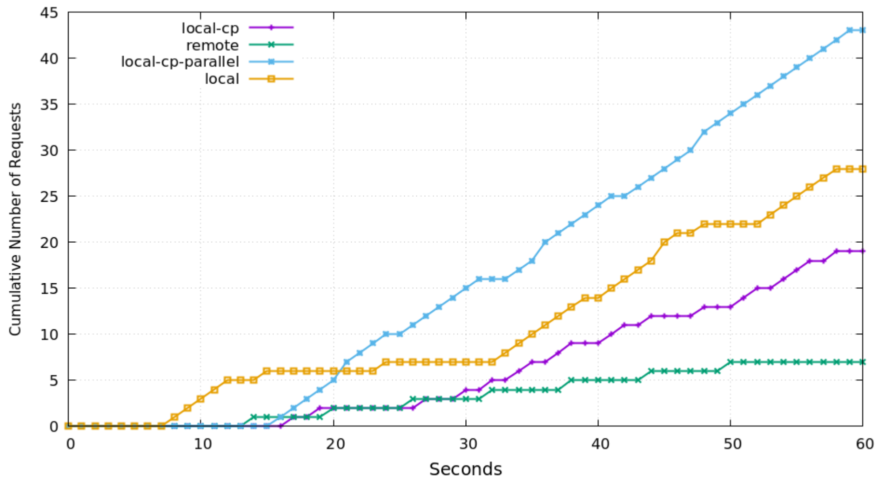

Figure A.2.: Multiple

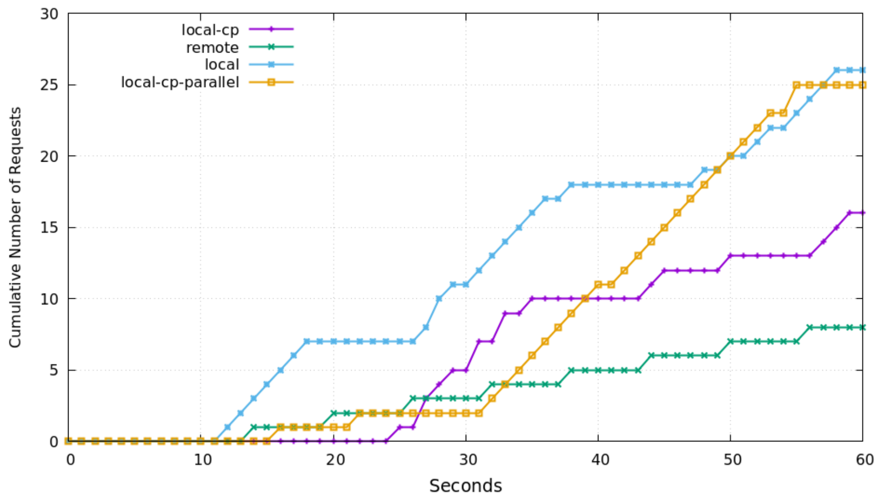

Figure A.3.: Partial

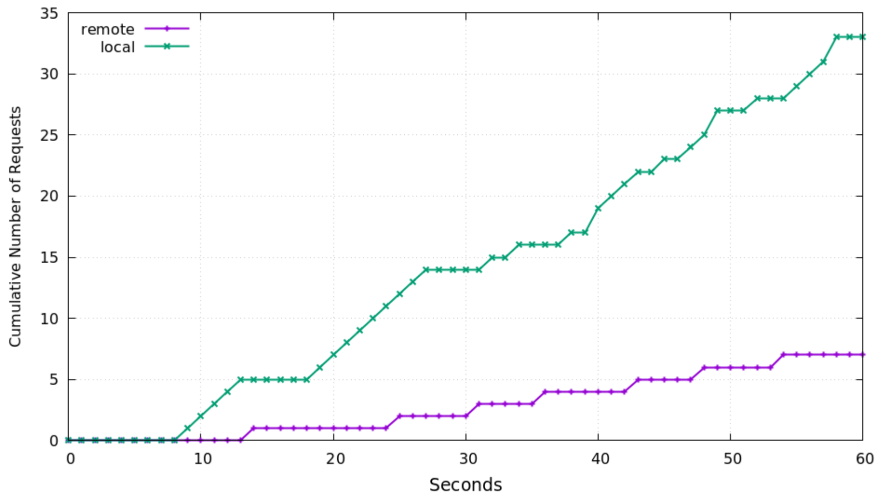

Figure A.4.: Partial1k

A.2. 5 Minutes

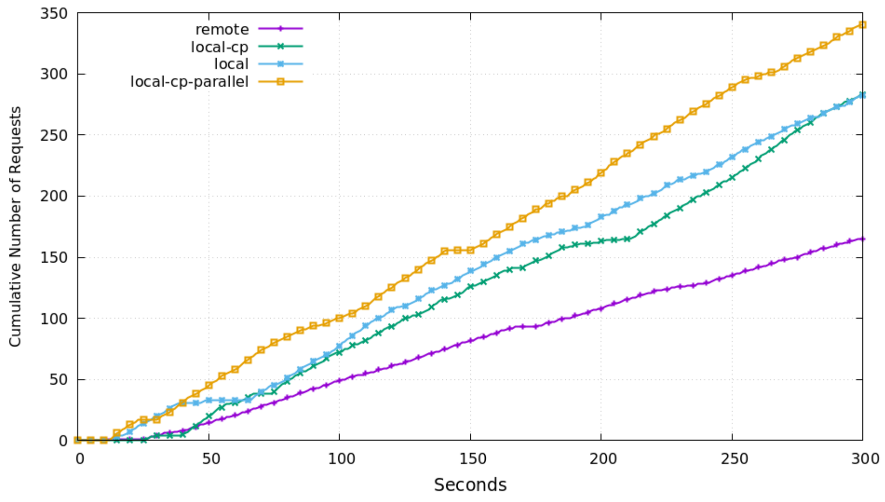

Figure A.5.: Single

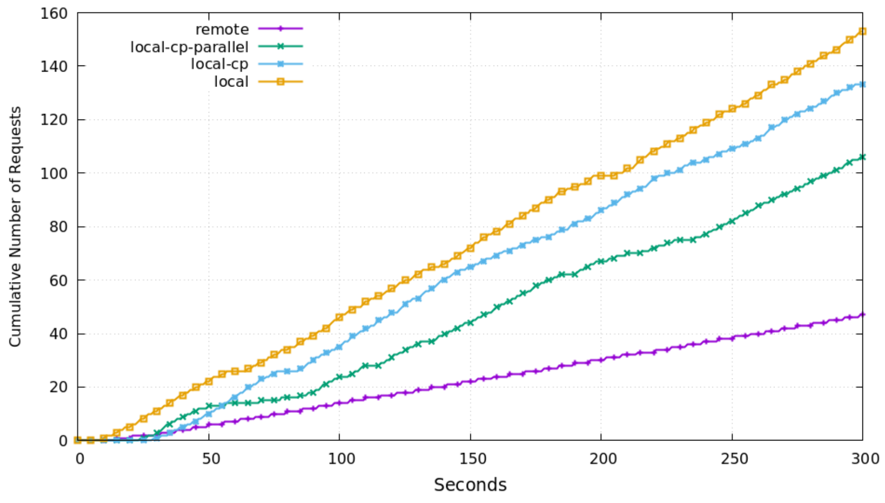

Figure A.6.: Multiple

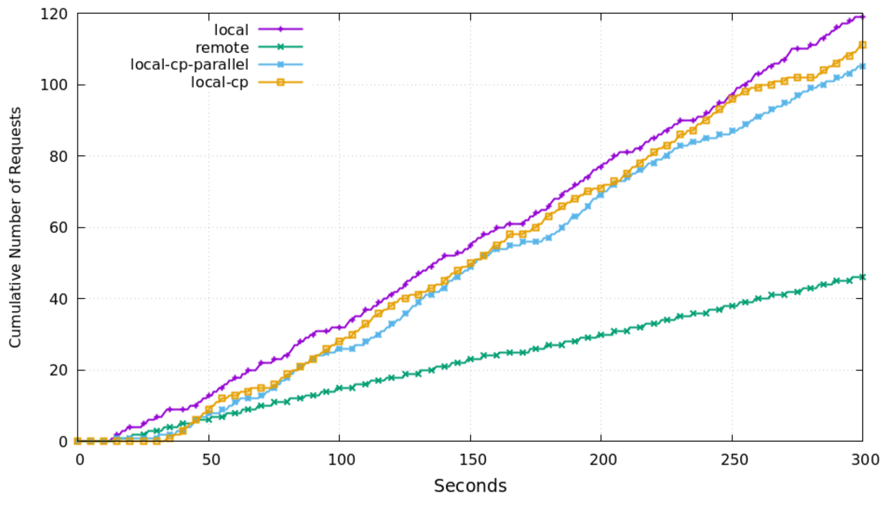

Figure A.7.: Partial

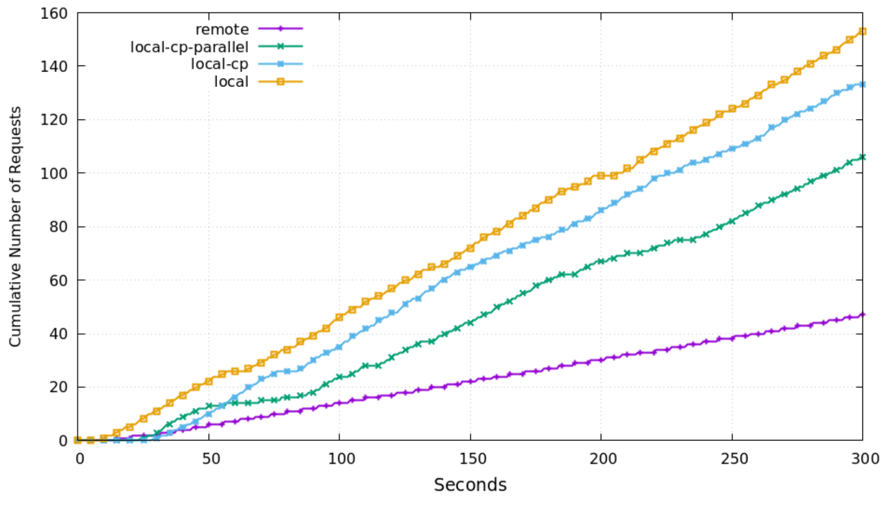

Figure A.8.: Partial1k

A.3. 15 Minutes

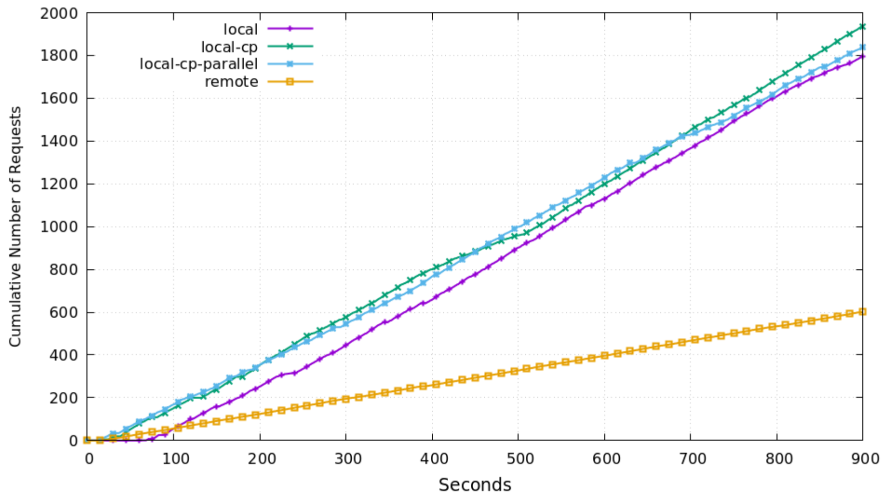

Figure A.9.: Single

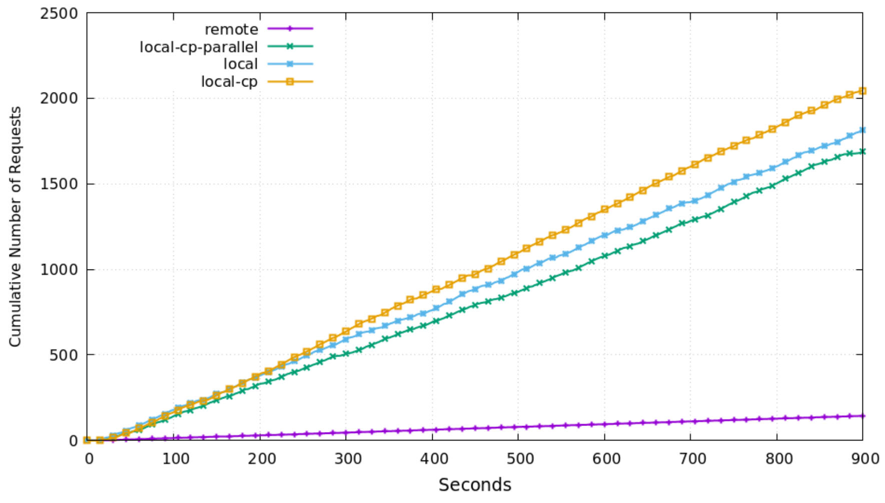

Figure A.10.: Multiple

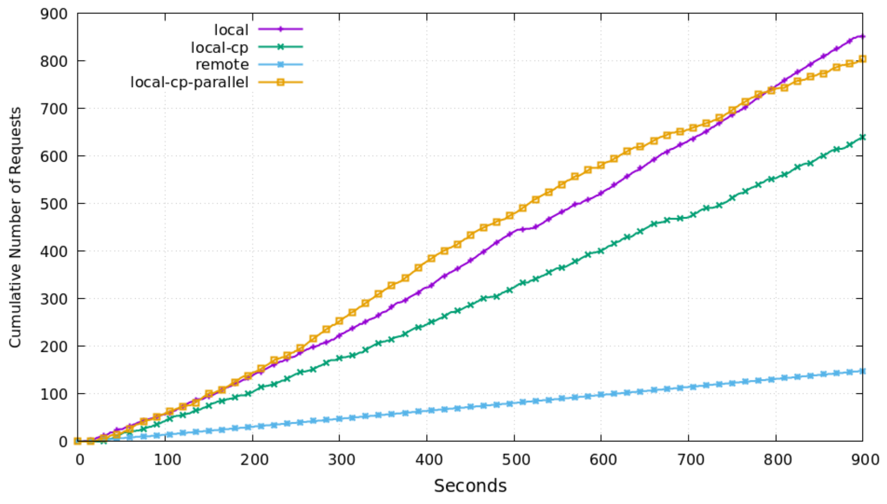

Figure A.11.: Partial

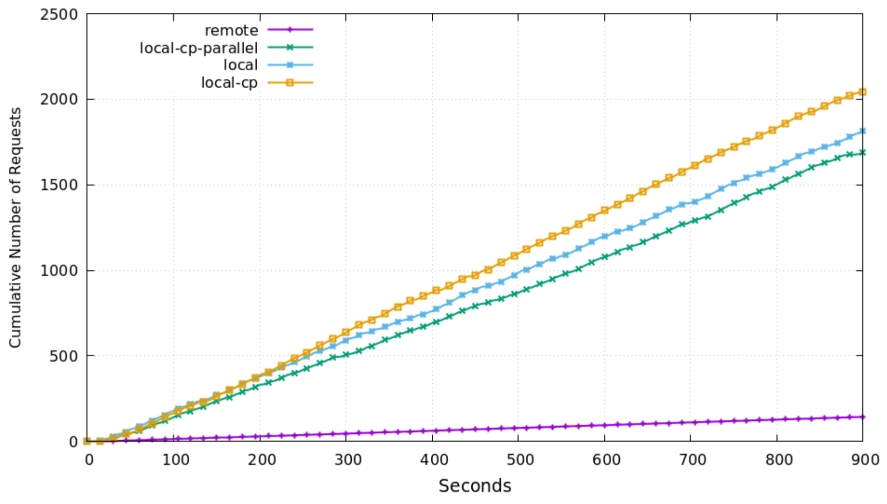

Figure A.12.: Partial1k

A.4. 30 Minutes

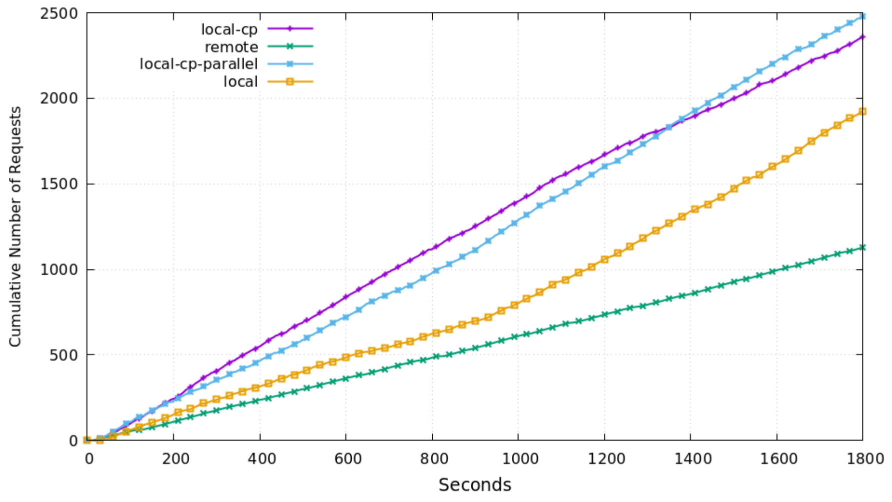

Figure A.13.: Single

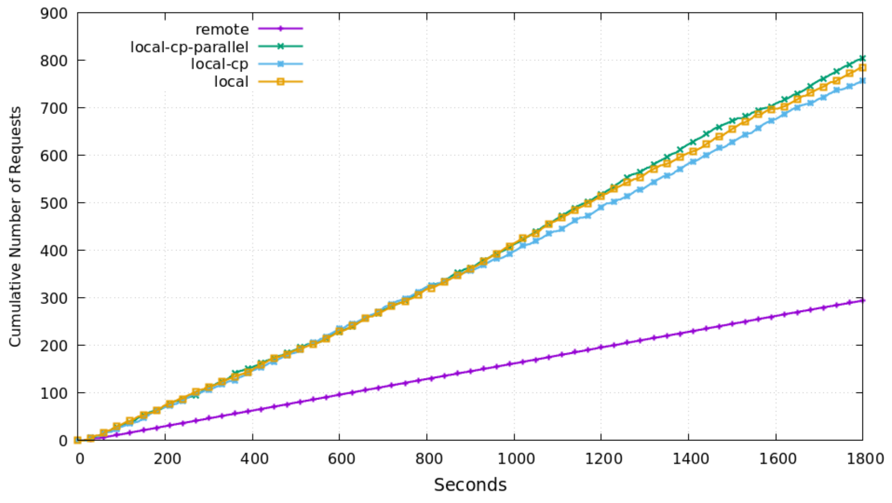

Figure A.14.: Multiple

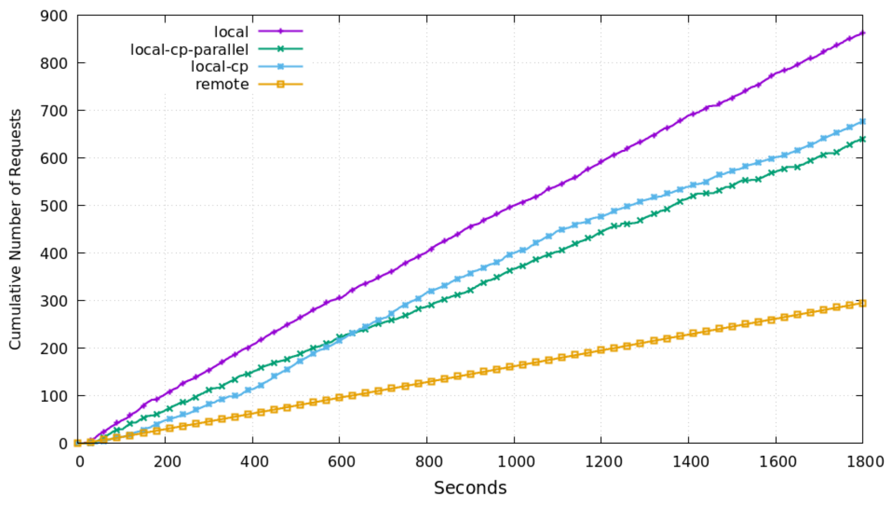

Figure A.15.: Partial

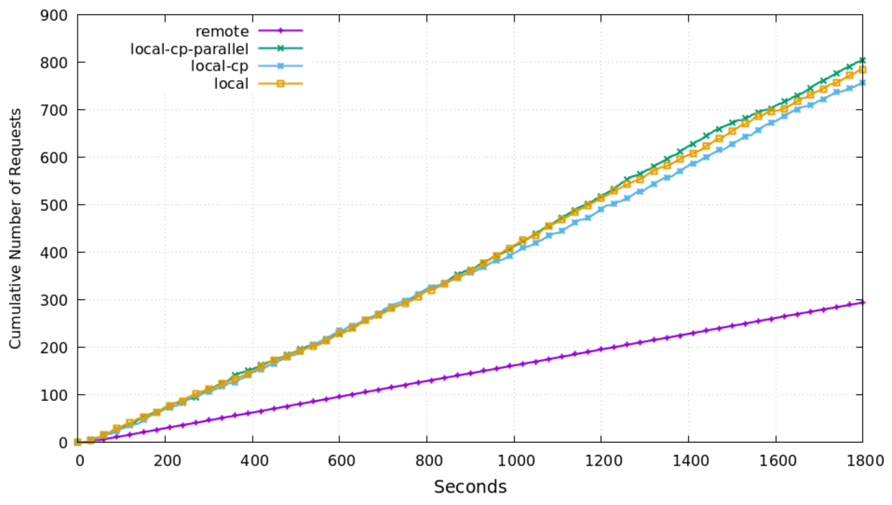

Figure A.16.: Partial1k

List of Figures

List of Tables

Acronyms

AWS Amazon Web Services. 1, 6, 48

DAFS Data Aware Function Scheduler. 30–33, 36, 54, 56, 66

DNS Domain Name System. 24

FaaS Function-as-a-Service. iv, 1, 2, 5, 6, 26–28, 57

FaaST FaaS Scheduler Tool. 32, 33, 56, 57

FQDN fully qualified domain name. 13

HCL HashiCorp Configuration Language. 12

IoT Internet of Things. 56

JSON JavaScript Object Notation. 6, 10, 36, 45, 49, 52

LAFS Latency Aware Function Scheduler. 28–30, 32, 33, 35, 56, 66

LRZ Leibniz-Rechenzentrum. 12, 24, 28, 45, 48, 51

NFS Network File System. 39, 40, 44, 57

PV Persistent Volume. 39, 40, 44

PVC Persistent Volume Claim. 40, 41

SCP Secure Copy Protocol. 44, 46

SLO Service Level Objective. 27

VCS Version Control System. 12

VM	Virtual Machine. 5, 21–23, 26, 27, 32, 43, 44, 50, 51, 55
VMM	Virtual Machine Monitor. 21
VU	Virtual User. 46, 48

Bibliography

[1] *About Ansible*. 2020. URL: https://docs.ansible.com/ansible/latest/index.html#about-ansible (visited on 07/09/2020).

[2] *About Playbooks*. 2020. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html#playbooks-intro (visited on 07/09/2020).

[3] Z. Al-Ali, S. Goodarzy, E. Hunter, S. Ha, R. Han, E. Keller, and E. Rozner. "Making Serverless Computing More Serverless". In: July 2018, pp. 456–459. DOI: 10.1109/CLOUD.2018.00064.

[4] *AWS Lambda Pricing*. 2020. URL: https://aws.amazon.com/lambda/pricing/ (visited on 05/31/2020).

[5] *AWS Lambda releases*. 2020. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html (visited on 05/31/2020).

[6] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing*. Ed. by S. Chaudhary, G. Somani, and R. Buyya. Singapore: Springer Singapore, 2017, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8_1.

[7] D. Bermbach, S. Maghsudi, J. Hasenburg, and T. Pfandzelter. "Towards Auction-Based Function Placement in Serverless Fog Platforms". In: Apr. 2020, pp. 25–31. DOI: 10.1109/ICFC49376.2020.00012.

[8] D. Bermbach, F. Pallas, D. Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai. "A Research Perspective on Fog Computing". In: June 2018, pp. 198–210. ISBN: 978-3-319-91763-4. DOI: 10.1007/978-3-319-91764-1_16.

[9] D. Bermbach, F. Pallas, D. G. Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai. "A Research Perspective on Fog Computing". In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Ed. by L. Braubach, J. M. Murillo, N. Kaviani, M. Lama, L. Burgueño, N. Moha, and M. Oriol. Cham: Springer International Publishing, 2018, pp. 198–210.

[10] *Creating Modules*. 2020. URL: https://www.terraform.io/docs/modules/index.html#module-structure (visited on 07/07/2020).

[11] *Docker architecture*. 2020. URL: https://docs.docker.com/get-started/overview/#docker-architecture (visited on 07/19/2020).

[12] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha. "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement". In: *CoRR* abs/1811.09721 (2018). arXiv: 1811.09721.

[13] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 171–172.

[14] G. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. *Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research*. Aug. 2017. DOI: 10.13140/RG.2.2.15007.87206.

[15] *Google Trends*. 2020. URL: https://trends.google.com/trends/explore?date=all&q=serverless (visited on 11/20/2020).

[16] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das. "Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019, pp. 199–208. DOI: 10.1109/CLOUD.2019.00043.

[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". In: *CoRR* abs/1812.03651 (2018). arXiv: 1812.03651.

[18] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016.

[19] *How OpenWhisk works*. 2020. URL: https://github.com/apache/openwhisk/blob/master/docs/about.md#how-openwhisk-works (visited on 07/20/2020).

[20] *How Terraform Works - CLI*. 2020. URL: https://www.terraform.io/ (visited on 07/07/2020).

[21] *How to build your inventory*. 2020. URL: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#how-to-build-your-inventory (visited on 07/09/2020).

[22] *Introduction to Modules*. 2020. URL: https://docs.ansible.com/ansible/latest/user_guide/modules_intro.html#introduction-to-modules (visited on 07/09/2020).

[23] *Kubernetes Components*. 2020. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 07/19/2020).

[24] H. Lee, K. Satyam, and G. Fox. *Evaluation of Production Serverless Computing Environments*. July 2018. DOI: 10.13140/RG.2.2.28642.84165.

[25] *OpenWhisk - application.conf*. 2020. URL: https://github.com/apache/openwhisk/blob/master/core/invoker/src/main/resources/application.conf#L85 (visited on 06/01/2020).

[26] *OpenWhisk - KubernetesClient.scala*. 2020. URL: https://github.com/apache/openwhisk/blob/master/core/invoker/src/main/scala/org/apache/openwhisk/core/containerpool/kubernetes/KubernetesClient.scala#L84 (visited on 06/01/2020).

[27] *Roles*. 2020. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse_roles.html#roles (visited on 07/09/2020).

[28] M. Sciabarrà. *Learning Apache OpenWhisk: Developing Open Serverless Solutions*. O'Reilly Media, Inc., 2019. Chap. 1, pp. 3–22. ISBN: 9781492046165.

[29] *Serverless Computing*. 2020. URL: https://researcher.watson.ibm.com/researcher/view_group.php?id=8294 (visited on 05/31/2020).

[30] A. Suresh and A. Gandhi. "FnSched: An Efficient Scheduler for Serverless Functions". In: Dec. 2019, pp. 19–24. ISBN: 978-1-4503-7038-7. DOI: 10.1145/3366623.3368136.

[31] *What is Apache OpenWhisk?* 2020. URL: https://openwhisk.apache.org/ (visited on 07/20/2020).

[32] *What is Kubernetes?* 2020. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes (visited on 07/19/2020).

[33] *What's a Container?* 2020. URL: https://www.docker.com/resources/what-container (visited on 07/19/2020).

[34] *What's a Linux container?* 2020. URL: https://www.redhat.com/en/topics/containers/whats-a-linux-container (visited on 07/19/2020).

[35] *YAML format reference*. 2020. URL: https://docs.openfaas.com/reference/yaml/#function-constraints (visited on 06/01/2020).