ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Topology-based Scheduling in Serverless Computing Platforms

Relatore:
Chiar.mo Prof.
Gianluigi Zavattaro

Correlatori:
Prof. Jacopo Mauro
Dott. Saverio Giallorenzo
Dott. Giuseppe De Palma

Presentata da:
Matteo Trentin

Sessione II - Secondo appello
Anno Accademico 2020/2021

Abstract

In the past few years, Function as a Service (FaaS) solutions, and Serverless computing in general, have become a significant topic both in terms of general interest and research effort. Allowing users to run stateless code in the cloud without worrying about the underlying infrastructure for scheduling, management and scaling, the ease of use of these approaches still comes with various trade-offs and challenges. In this thesis, the issue of data locality is observed, using an extension of the Apache OpenWhisk framework to provide users the ability to select the node they wish to use to schedule some of their functions, allowing the code to be run closer to the data it manipulates. Additionally, a topology-based scheduling approach is implemented for the framework, where load balancers are instructed to prioritize nodes in their same topological zone; this way, users can specify a preferred load balancer for different functions, with no need to know the position and name of all other nodes in the cluster. This modified version of the OpenWhisk framework is then compared with the standard OpenWhisk implementation, along with two other serverless frameworks, Fission and OpenFaaS, using a test suite composed of different use cases, using both existing projects from the Wonderless dataset and custom-built functions targeting different aspects of the paradigm. The role of data locality considerations and topology-based policies is analyzed, showing their importance in a multi-zone cluster with nodes in various geographical locations, where latency between them and the remote data used by the functions can be significant.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Cloud Computing

According to NIST in [21], cloud computing is “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction”.

In other words, computing in the cloud means having on-demand access to delocalized resources managed by a cloud service provider, and using those resources to perform different tasks, depending on the service model and the kind of resource provisioned.

Cloud computing has many different applications, with the main advantage of removing the onus of configuring and maintaining data centers for computationally intensive tasks, and delegating it to the provider; this translates in a cost-effective solution most of the time, reducing IT costs and introducing usage-based pricing.

1.2 Cloud Service Models

Cloud service providers can offer different capabilities and computing resources, matching different end users and necessities. In this section the main service models are described; the first three are defined in [21] and further clarified in [27], while the fourth one has more recently emerged as a potential alternative to the previous models, with its own advantages and challenges.

1.2.1 Infrastructure as a Service

In this service model, the user can manage fundamental computing resources, such as storage, processing power and networks, and deploy arbitrary software on them; usually these resources are in the form of either Virtual Machines or containers, deployed on the provider's physical resources.
While it is possible for the end user to manage some networking components, such as firewalls for hosts and subnets, no capabilities over the underlying cloud infrastructure are given; nevertheless, the IaaS model offers the lowest-level control of resources out of the service models here described.

The typical end users are either developers, or IT roles in general, who want direct control over the computational resources in use.
Some examples of IaaS include EC2 from Amazon Web Services (AWS), Google Compute Engine (GCE) and DigitalOcean.

1.2.2 Platform as a Service

The PaaS model is the following level of abstraction, where the end user has no control over the underlying infrastructure and computing resources, and has some configuration options over the environment of his choice.
The environment is then used as a computing platform to deploy and manage one

or more applications, using languages, libraries and additional services supported by the selected environment (and provider).

In this model, the provider is responsible not only for their physical resources, but also for middleware, database systems, operating systems and other additional services necessary for the consumer application; the end user only has control over the deployed applications and their data.

Typical end users are developers who wish to publish their applications into the cloud; services such as AWS Elastic Beanstalk, Google App Engine and Heroku are notable examples of PaaS.

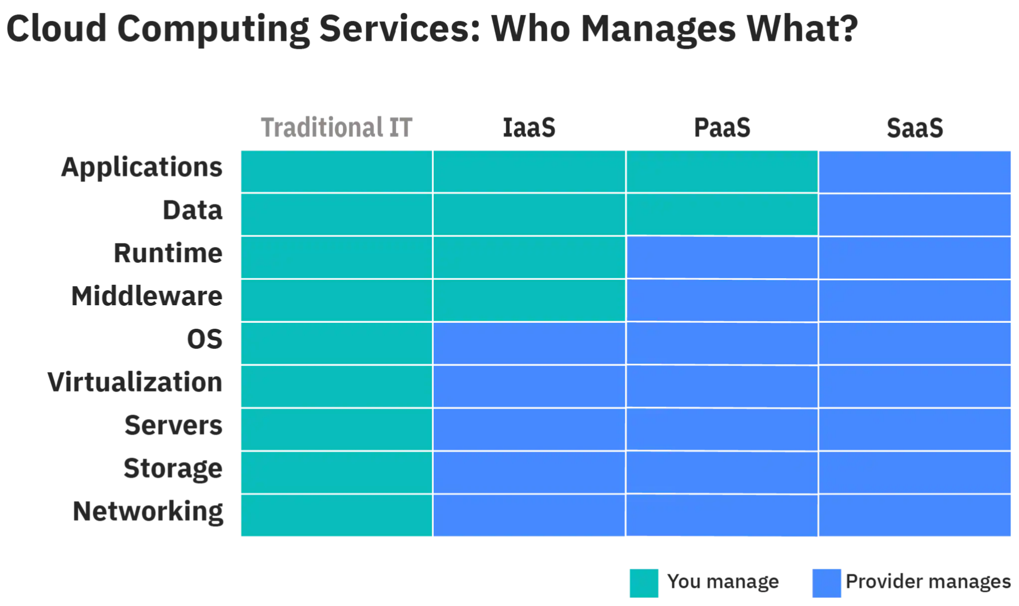
1.2.3 Software as a Service

The SaaS model encompasses most web applications, where the provider manages all resources, including the code run on the cloud, and the consumer only has the capability of using the application offered as a service.
The client used to access the application can be both a web browser or a standalone program interface, but the actual application code is still run in the cloud.

Popular examples of SaaS are GMail, Slack or Netflix; compared to the previous two models, typical end users are required to possess less technical skills, and are able to simply use the provided software application.

1.2.4 Function as a Service

The FaaS model is not described by NIST in [21], and emerged around 2010 as a category of cloud computing services allowing the end user to execute code in response to events, without the responsibility of managing any part of the underlying infrastructure.



Cloud Computing Services: Who Manages What?

	Traditional IT	IaaS	PaaS	SaaS
Applications	🟩	🟩	🟩	🟦
Data	🟩	🟩	🟩	🟦
Runtime	🟩	🟩	🟦	🟦
Middleware	🟩	🟩	🟦	🟦
OS	🟩	🟦	🟦	🟦
Virtualization	🟩	🟦	🟦	🟦
Servers	🟩	🟦	🟦	🟦
Storage	🟩	🟦	🟦	🟦
Networking	🟩	🟦	🟦	🟦

🟩 You manage 🟦 Provider manages

Figure 1.1: An illustration of the different levels of abstraction in the various cloud service models; it can be seen how IaaS, PaaS and SaaS progressively move management and operation responsibilities from the end user to the cloud provider. Picture from `https://www.ibm.com/cloud/learn/iaas-paas-saas`.

.

From the user's point of view, FaaS platforms have better "built-in scaling" compared to PaaS and especially IaaS solutions, since not only the physical hardware, but also the OS and any additional software are handled by the provider; this effectively removes the need for user interaction and tuning when it comes to scaling.

Advantages, challenges and applications of FaaS and serverless in general are described in the next section.

1.3 FaaS and Serverless Computing

FaaS is a central part of serverless computing, focusing on the paradigm where the function's code is only executed in response to specific events or requests; although the two are often conflated together, FaaS is technically a subset of serverless computing, as the latter also encompasses every service category (e.g. databases or storage services) "where configuration, management, maintenance and billing of servers are invisible to the user"[41].

As such, serverless computing in general refers to the cloud computing execution model where resources are provisioned on demand, and every management task is responsibility of the cloud provider.

While the name can be misleading, servers are of course still part of the serverless architecture; the denomination acts as a description from the customers' perspective: servers are invisible to them, and never part of any direct interaction.

1.3.1 Characteristics

Applications based on serverless computing, and more specifically on FaaS solutions, usually share some common characteristics. As found in [7], a significant majority of analyzed serverless applications have a very short runtime, in the order of seconds or milliseconds, while less than a fourth have a runtime of minutes; in the same paper, it is shown that serverless applications very often have "bursty" workloads (i.e. characterized by a pattern that includes sudden load spikes) and are invoked using mostly HTTP-based or cloud-based triggers, the latter including events such as file uploads to a cloud-based storage.

Another characteristic usually associated with FaaS-based applications is the use of small, isolated functions, each performing specific tasks; scaling small functions is generally easier and less expensive, since usually only specific components of an application actually need to be scaled; if bigger and less "specialized" functions are used instead, many parts of the application might end up being scaled unnecessarily.

Smaller functions, in this case functions that don't require many libraries, also have faster loading times; this comes both from the size of the uploaded archive containing the function code and the instantiation time for the libraries; all of this contributes to the *cold start* issue (1.3.2), making the use of as few libraries as possible a general guideline for functions in FaaS applications.

1.3.2 Advantages and issues

As mentioned before, serverless computing allows developers to focus entirely on the actual application, with no responsibility over the management of the underlying infrastructure; this particular perk, combined with the multiple language runtimes generally available in serverless platforms, helps streamlining the development and deployment process; developers can code in many programming languages, with various libraries, without the need to know the infrastructure on which the applications will run.

Compared to IaaS solutions, serverless computing is also usually more cost-effective, since the customers only need to pay for the function execution; this is especially useful in case of dynamic or sporadic workloads, with alternating short periods of computation and long periods of idling; with a IaaS-based approach a consumer would have to pay for the provisioned resources until their explicit removal (more precisely, decommission), while with a FaaS-based approach only the time of execution would be accounted for payment.

At the same time, various challenges and shortcomings exist in different areas of serverless and FaaS; the authors of [19] present a summary of findings regarding current behaviors and constraints of these solutions, such as:

- FaaS generally offers a single function abstraction, with no distinction for different function “types”; this tends to limit the amount and type of resources available to the function handler’s runtime, failing to adapt to more computationally intensive applications. Moreover, if platforms impose a maximum execution time on a function, errors can be raised in case of long tasks; a popular example of this issue are deep learning applications, requiring both long execution times and better control on the hardware running the function (since GPUs are in many cases required for such computations).
- Users have no control over the platform’s elasticity controller; because of this, the standard provisioning and de-provisioning time of FaaS solutions might not suit the application’s scaling needs (e.g. the service may not scale fast enough to respond to spikes in traffic, or might de-provision resources too greedily, impacting performance when it comes to caching results from previous executions). This creates an even bigger issue when combined with the relatively limited “elasticity” of many services, as noted by the authors of [17]; the resulting ability to rapidly scale according to the volume of requests might consequently not meet the demands of serverless applications.
- While FaaS tends to be more cost-effective compared to other solutions, since idle time is not accounted for in its cost model, the actual price of execution can still be inflated by other factors; firstly, developers can still specify the “size” of the functions (i.e. the amount of memory the function should use), meaning they still need to evaluate the cost-performance trade-off for the application (since the default amount might either not suffice for the invocation or exceed the actual needs); secondly, since function composition is an integral part of FaaS applications, cost increases related to it are to be considered (e.g. synchronous calls between functions, where the waiting time is still billed as execution time).

Similar problems are also pointed out by the authors of “Serverless Computing: One Step Forward, Two Steps Back” [13], namely the inability to access specialized hardware and the limited function lifetime in the AWS Lambda platform.

An issue strongly related to commercial solutions is the possibility of vendor lock-in; while some serverless applications can be deployed on different cloud providers, and a number of open source frameworks are available for use in private clouds, proprietary solutions are still a very popular choice (as shown in [19], 80% of the surveyed cases are deployed on AWS); moreover, if the application requires interaction with specific workflows or services, even migrating to different providers can be very difficult; as the authors of [17] discuss, vendor lock-in may be stronger with serverless computing compared to “serverful” solutions. Finally, deploying an open source solution on a private cluster requires the actual management of the cluster, which is in contrast with the main appeal of the paradigm; all of this can make users rely on a single cloud provider, thus creating a situation of vendor lock-in.

Especially in the context of hybrid and multi-zone clusters, the issue of locality arises; as described in [14], *session locality*, *code locality* and *data locality* need to be considered by load balancers. The issue of locality will be better explored in Chapter 2, along with other research themes.

Finally, one of the most discussed issues of FaaS is handling *cold starts*, i.e. the time needed to initialize the environment used to run the code; the startup latency can be noticeable, especially when functions are packaged with a higher number of libraries, leading to a delay of seconds before the actual code execution. This particular issue can make serverless approaches unfeasible for applications where fast response times are a priority (e.g. financial trading); nonetheless, as found by the authors of [7], use cases where a degree of stable latency is required (e.g. interaction with human users) are still present, regardless of cold starts.

1.3.3 Use cases

According to IBM[42], the main use cases for serverless architecture are microservices, API backends, data and stream processing, and embarassingly parallel

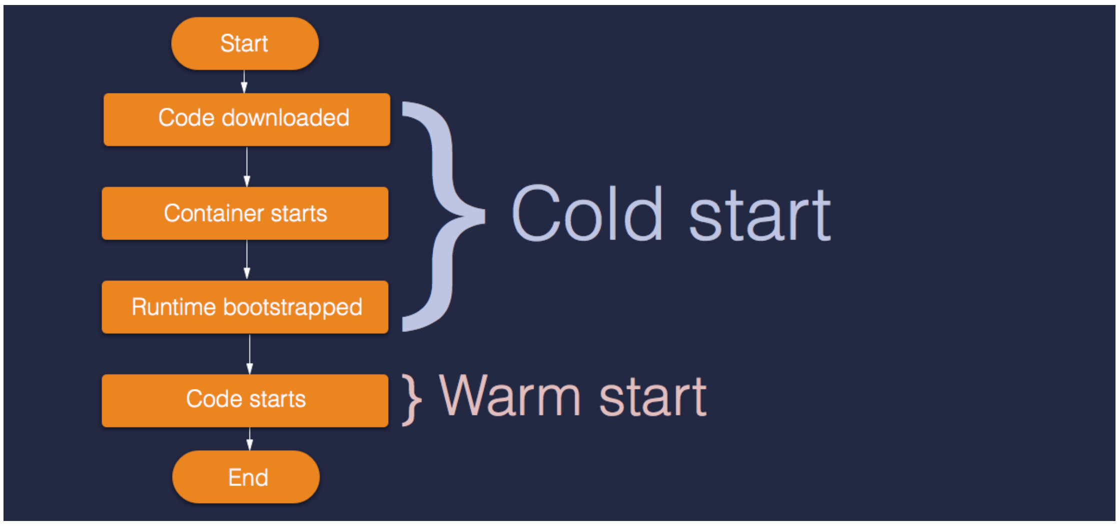


Figure 1.2: Illustration of what constitutes the *cold start* in serverless functions; the actual code execution is preceded by several bootstrapping operations, which can considerably affect latency times. Picture from `https://medium.com/ssense-tech/the-trade-offs-with-serverless-functions-71ea860d446d`.

.

tasks.
The first category is focused on creating small services that do a single job, and has requirements of scalability and modularity; given the advantages of serverless computing when it comes to automatic scaling and quick provisioning and de-provisioning of resources, this model is well suited for the microservices use case.
The second category is a natural consequence of the possibility, for serverless platform, to invoke functions based on external triggers, such as HTTP requests; every action can then become an HTTP endpoint, and be combined with others to effectively become an API.
The third category makes use of the simplicity and fast scalability of serverless solutions, and as shown in “An Evaluation of Serverless Data Processing Frameworks”[32], serverless frameworks can perform better and be more cost-effective compared to approaches such as Amazon EMR.
The fourth category is well suited for serverless, and specifically the FaaS paradigm, as it consists of applications where tasks are almost, if not completely, independent;

each parallelizable sub-task can then be handled by a single action invocation.

Similarly, the authors of [13] list three main categories of applications: embarassingly parallel functions, orchestration functions and function composition. The first category has already been described, and the authors note how it is inherently limited in scope and complexity; the second one consists of functions used to orchestrate calls to other services (e.g. preprocessing data before sending it to analytics applications); the last one encompasses "collections of functions that are composed to build applications, and thus need to pass along outputs and inputs" (e.g. groups of functions chained together by triggering events on storage services).

Apache OpenWhisk documentation also offers a list[1] of common use cases for their framework, including IoT applications and support for cognitive technologies (e.g. extracting frames from an uploaded video, which are then to be processed by visual recognition software).

As noted by the authors in [7], serverless can still be seen as a broadly applicable solution, and their finding show that serverless applications are quite evenly distributed between APIs, asynchronous and stream processing, and applications assisting in monitoring, operating and automating software systems.

1.3.4 Providers and platforms

When talking about FaaS platforms, a distinction can be made between open-source frameworks, available for deployment on both public and private clouds, and proprietary platforms, tied to specific cloud providers.

The main examples of the latter category are *AWS Lambda*, *Google Cloud Functions* and *Azure Functions*, offered by Amazon, Google and Microsoft respectively. The main upsides of using a proprietary solution are the robust underlying infrastructure, which grants a very high level of reliability, and the ability to interact

[1]`https://github.com/apache/openwhisk/blob/master/docs/use_cases.md`

very easily with the provider's ecosystem (e.g. using the addition of an object in an S3 bucket as a trigger to invoke a function on AWS Lambda[2]).

The main downside of proprietary solutions is having to rely on the provider both for framework updates and infrastructure; this reduces flexibility for developers, and relying on commercial services can create vendor lock-in. A greater control over the infrastructure and the deployment of the serverless platform can also help optimizing resource usage and latency (e.g. with data locality issues); moreover, having a better knowledge of the infrastructure facilitates testing and benchmarking.

For these reasons, only open source frameworks were considered for the purpose of this thesis; a deeper analysis of the three that were chosen is given in 2.2.

1.4 Objective of the Thesis

The main purpose of this thesis is to analyze and optimize the behaviour of serverless platforms in a multi-zone cluster, building upon the work done in [6]; specifically, the authors' aim is to improve on the flexibility of a configurable load balancer for the Apache OpenWhisk platform, extending its behavior in presence of multiple replicas of the service.

Through this, the objective is to address the issue of data locality, allowing functions to be invoked near the data, regardless of the amount of nodes and zones in the network; at the same time, developers can impose a degree of tolerance on the load balancers' behavior, requesting the code to only be executed on certain nodes, and only be scheduled by certain replicas (e.g. private code should only be scheduled by private load balancers, and should fail everywhere else).

Following this additions, the authors focused on comparing performance and

[2]`https://docs.aws.amazon.com/lambda/latest/dg/with-s3-example.html`

behavior of three different open source serverless frameworks; this was done by building a test suite composed of basic functions, realistic use cases taken from the Wonderless [8] dataset, and *critical cases*, addressing some known issues with the platforms and serverless in general. The same test suite was then used to evaluate the modified version of OpenWhisk, analyzing the improvements.

1.4.1 Contributions

The contributions of this thesis, mainly described in Chapter 3, were:

- The extension of the APP language described in [6], to include configuration options in situations with replicated Controllers and load balancers.
- The implementation of a topology-based Invoker distribution in a multi-zone cluster case, having load balancers prioritize nodes in the same topological zone.
- The implementation of a script influencing Nginx behavior when handling requests to replicated Controllers, to assist the aforementioned topology-based distribution.
- The definition of a test suite, to compare serverless platforms behavior and performance in various use cases, including some known pitfalls of both the platforms and the paradigm.

1.5 Structure of the Thesis

The first chapter of this thesis was intended as an introduction to the purpose, the challenges and the advantages of serverless computing.

The second chapter covers the state of the art of serverless computing, focusing on current themes of research regarding the improvements of serverless platforms; three specific open source FaaS frameworks are then described and analyzed, showing similarities and differences in their architecture and approaches.

The third chapter presents the contributions of this thesis, more precisely, the additions made to the Apache OpenWhisk project and its deployment configuration on the Kubernetes system.

The fourth chapter illustrates the design of the test suite used on the platforms described in Chapter 2, and on the customized version obtained with the additions from Chapter 3; it includes the considered use cases, the critical cases and the testing results.

The last chapter contains both the conclusions of the thesis and possible future works.

Chapter 2

State of the Art

Serverless computing and FaaS are continuously growing areas of research, despite the technologies' relatively young age (AWS Lambda was only introduced at the end of 2014[43]); in this time span, many papers were published regarding a plethora of issues, challenges and innovations for the paradigm, ranging from formal models to scheduling optimizations with IoT and edge computing, along with different benchmarks and comparison of different FaaS platforms.

At the same time, many of these platforms are in active development; from commercial solutions, such as AWS Lambda, Google Cloud Functions and Azure Functions, to open source frameworks, such as Apache OpenWhisk, OpenFaaS, Kubeless, Fission and more; as expected, different platforms have different architectures and technology stacks, resulting in various approaches to function scheduling and deployment.

The first section of this chapter gives an overview of some of the popular research themes, with references to related recent publications. The second section is composed of four more subsections, with the first three respectively containing a description of the Apache OpenWhisk[37], OpenFaaS[53] and Fission[40] platforms; the last subsection contains a note on the Kubeless platform, which was discarded during the writing of this thesis.

2.1 Research overview

As previously mentioned, the young age and growing popularity of the serverless computing paradigm result in numerous and varied areas of research.

2.1.1 Architecture

A popular theme is the optimization of the underlying architecture, targeting components such as virtualization technologies and orchestration systems. Docker[38] has, for a long time, been considered the most common virtualization and containerization technology; as Docker by itself lacks a way to easily orchestrate container deployments, especially in cloud systems, technologies such as Docker Swarm and the vastly more popular Kubernetes[47] exist. While Docker is still a very commonly chosen technology for containerization, alternatives have been developed in the last few years, such as:

- Firecracker[2], the virtual machine monitor (VMM) powering AWS Lambda and AWS Fargate. Firecracker uses the Linux Kernel's KVM infrastructure to provide *microVMs*, extremely lightweight virtual machines, with the advantage of a better process isolation compared to containers (since virtual machines offer an additional level of sandboxing for applications, instead of relying directly on the kernel's sandboxing capabilities). Firecracker is also generally seen as an alternative to QEMU[5], since its only purpose is to run serverless functions and containers; this reduced versatility results in better performance and a lower memory footprint.
- LightVM[20] is a solution developed by NEC and, similarly to Firecracker, has the main purpose of being a lightweight and secure virtualization tech-

nique, minimizing the virtual machines' overhead and optimizing instantiation times. It is based on the Xen hypervisor.

- Kata Containers[45] are another approach to obtain better isolation; containers are executed withing QEMU-based virtual machines, using the hypervisor as a way to further separate the container runtime from the host system's kernel. Kata Containers also support the use of Firecracker as hypervisor.
- Faaslets[25], a recent solution targeting the issue of stateful serverless computing. Running on the FAASM runtime, developed by the same authors, this approach uses a distributed state on both a local (in-memory sharing) and global (across hosts) level to reduce data access and serialization overheads.

Regarding orchestration systems, and interesting alternative to Kubernetes is Nomad[51]; the developers offer a comparison to Kubernetes, including simplicity and a good scalability as advantaged of their solution[1]. In the serverless world, Nomad is used as an orchestrator by the Koyeb[46] Serverless Engine, along with the aforementioned Firecracker for virtualization; Koyeb's developers also offer an explanation for their choice of Nomad over Kubernetes[2].

2.1.2 Cold starts

Many publications focus on techniques to reduce the impact of cold starts in the initial latency. The authors of [28] offer a solution based on multiple queues, at the same time "pre-warming" containers and keeping multiple copies of the ones that were recently used; this way, concurrent invocations can be scheduled on previously initialized containers. The results in the paper show that this approach accomplishes very good startup times at the cost of a moderately higher memory usage.

[1] `https://www.nomadproject.io/docs/nomad-vs-kubernetes`

[2] `https://www.koyeb.com/blog/the-koyeb-serverless-engine-from-kubernetes-to-nomad-firecracker-and-kuma`

A work from the end of 2020, “Prebaking Functions to Warm the Serverless Cold Start”[26], describe a technique based on restoring snapshots from previously executed function processes; this way, when a new function instance is created (or when a function replica needs to be instanced), the platform can restore the snapshot; the only time a new snapshot is created is when the user deploys a new function version, generally reducing the overhead from the snapshot creation. The authors then integrate the prebaking solution with the OpenFaaS platform, describing the necessary steps in the paper.

On the other hand, a work from 2019[22] focuses on the impact of initializing network elements in containers; the authors find that network creation and connection account for a large part of cold start times, and develop an approach based on pre-creating networks and connecting them to function containers. They accomplish this by creating a pool of “pause containers” (PC), which have their initialization paused after the network creation step; during function invocation, a newly launched container (built for a specific runtime) is attached to an available PC, removing it from the pool and exploiting the pre-initialized network components. The solution is then evaluated by modifying the Apache OpenWhisk platform, and comparing it with an unmodified version, showing significant improvements in startup times.

2.1.3 Formal models

Other papers propose formal models and definitions for serverless computing: in [9] the Serverless Kernel Calculus (SKC) is defined, combining ideas from λ-calculus (for functions) and π-calculus (for communication); SKC defines a serverless architecture as a pair $\langle S, D \rangle$ where S is the system of running functions and D is a definition repository. The authors then use an extension of SKC to encode a portion of Tailor[56], an architecture for user registration developed over AWS Lambda.

In the same year, a different paper[15] proposed formal tools for serverless computing, defining an operational semantics for of serverless platforms called λ_{λ}; this model also captures low-level details such as cold starts, storage, transactions and function restarts. The authors then extend λ_{λ} with a domain specific language for composing serverless functions, called “serverless programming language” (SPL), and implement different programs as case studies to identify possible features that might be missing from the language.

2.1.4 Scheduling

Additionally, one of the main areas of research is focused on optimizing function scheduling and workflows. While works such as [30] define algorithms for load balancing by acting on the controller level (i.e. scheduling functions on specific nodes), approaches such as the ETAS[4] scheduling scheme also exist. The authors of ETAS implement a new scheduling policy on the worker level, influencing the order in which the invocations should be removed by a worker’s waiting queue when resources become available; this algorithm estimates a function’s execution time from previous invocations and calculates the ideal finish time (defined as the sum of arrival time and execution time); functions are then executed according to their finish time, ensuring that small functions are not stuck behind larger ones (since parallel invocations prioritize shorter execution times), and that larger functions are not starved.

A different approach to scheduling optimization is given in “Faastlane: Accelerating Function-as-a-Service Workflows”[18], where the authors define a solution based on scheduling functions of a single workflow as threads within a single process of a container instance. This helps reducing the data sharing overhead, as functions can simply exchange data via simple load/store instructions; this represents both a study on optimizing function scheduling and on handling the issue of managing state in a specific workflow.

2.1.5 State management

State management is in itself another theme of research, and surveys such as [23] and [12] have recognised it both as a challenge for the paradigm and a difficult task for developers; the aforementioned Faasm[25] runtime tackles this by supporting both global and local state access, aiming at performance improvements for data-intensive applications.

In 2021, a paper introducing the Boki[16] FaaS runtime approaches the issue of state management by associating each function invocation with a “LogBook” (i.e. an abstraction allowing serverless functions to access shared logs), and having all functions of a Boki application sharing one; the aim of the authors is to address some demands in serverless computing, namely fault-tolerant workflows, durable object storage and shared message queues among serverless functions (allowing them to directly communicate with each other).

Another example of a stateful FaaS runtime is Cloudburst[29], a framework built on top of the Anna[33] key-value storage (KVS); a performance comparison between Boki and Cloudburst is also provided by the first runtime’s authors in their paper.

2.1.6 IoT and Edge

Many papers also discuss issues related to IoT and edge computing, and what challenges they present for serverless solutions. An interesting study combining serverless computing with edge and IoT devices is presented in [10], where the authors introduce the concept of “deviceless edge computing”; this paradigm allows functions to be executed on devices close to the user, instead of server, effectively integrating those devices in the serverless infrastructure.

This concept is also expanded in a study[11], where the possibility of “infrastructureless” computing is explored, evaluating challenges and advantages of possibly including embedded systems in the paradigm, while also using concepts from dew computing (executing functions on a local smart device, and offloading it to a server or a neighboring smart device if specific conditions are met, such as the presence of an internet connection or a low battery level); it has to be noted that the author only provides an analysis of this approach, not an implementation.

2.1.7 Locality

Finally, an important aspect in serverless computing, especially when working on hybrid clusters, is locality; in "Serverless Computation with OpenLambda"[14], three main types of locality are described: session locality, code locality and data locality.

- Session locality, which plays a role when function invocations are part of long-running sessions with open TCP connections. In this case, it is preferred to dispatch the invocations to the same worker node, if they share the same TCP connection; the authors comment on how this is beneficial, as it avoids diverting traffic through a proxy.

- Code locality, which should be considered by schedulers especially when handling functions which share many dependencies; as functions requiring a high number of libraries can incur in considerable delay due to cold starts, dispatching their invocations to nodes where the dependencies are already cached can result in considerable improvements.

  Works such as "Package-Aware Scheduling of FaaS Functions"[1] and its follow-up "Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms"[3] handle this by defining a package-aware scheduling algorithm, which assigns functions to nodes where a required package is already cached (in case of multiple package affinity, the largest package is chosen). The authors find that their solution greatly outperforms a least-loaded balancer in [3], with a slightly higher node imbalance as a trade-off (i.e. load is not evenly distributed between all worker nodes).

- Data locality, which consists in moving the computation closer to the data it uses; this is especially important when a function handles large amounts of data, or has to be otherwise run alongside databases. Data locality can have a significant impact in performance when working with hybrid and multi-zone clusters or edge computing, as in these cases nodes can be in different geographical areas, and network latency can become significantly different between zones.

The work upon which this thesis is built, "Allocation Priority Policies for Serverless Function-execution Scheduling Optimisation", introduces a declarative language to specify function scheduling policies, and implements an extension for Apache OpenWhisk which uses a configurable load balancer; this way, developers can specify different scheduling policies on different functions, requiring the invocation to be dispatched to specific nodes.

A different work[31] published in 2020 introduces Skippy, a scheduling system to be integrated with existing container orchestration; this prototype adds runtime components and domain concepts to make the orchestration system aware of device capabilities (e.g. network context, container images present on the node, availability of a GPU). The scheduler then analyzes function metadata to obtain information such as data consumption/production and capability requirements; this information is then given as input to various *priority functions*, the outputs of which influence the nodes where the invocation will be dispatched. The authors couple their solution with a modified version of the OpenFaaS platform, where function deployments indicate they should be scheduled by Skippy (instead of the default Kubernetes scheduler); the paper's results show that this modified version yields higher data throughput and less network traffic, at the cost of decreased scheduler performance (caused by the higher complexity).

Two previously mentioned solutions also tackle the issue of data locality by co-locating caches (Cloudburst[29]) and log index copies (Boki[16]) with the function executors.

2.2 Open Source Serverless Frameworks

This section contains an overview of the most popular (according to stars and contributors on the respective GitHub repositories) Open Source Serverless frameworks. First, the three that were used and analyzed in this thesis (Apache OpenWhisk, Fission and OpenFaaS) will be described in more detail: a deeper level of explanation will be reserved for OpenWhisk load balancing and scheduling algorithms, as the main contribution of this thesis was also based on modifying their behavior; then, a brief note on the Kubeless framework is given, along with the reasons why it was not used in this work.

2.2.1 Apache OpenWhisk

Apache OpenWhisk[37] is an open source serverless platform, initially announced by IBM and later donated to the Apache Software Foundation. OpenWhisk is currently maintained on GitHub with over 190 contributors and 5.4k stars. While the main core of OpenWhisk is written in Scala, the project “stands on the shoulders of giants”, to quote the main documentation; OpenWhisk’s main components include Nginx, Apache Kafka and CouchDB, which will be briefly described in the next paragraph; a simplified representation of OpenWhisk architecture is also shown in Figure 2.1.

Components

Nginx[50] is an open source HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server; Nginx is known for its performance

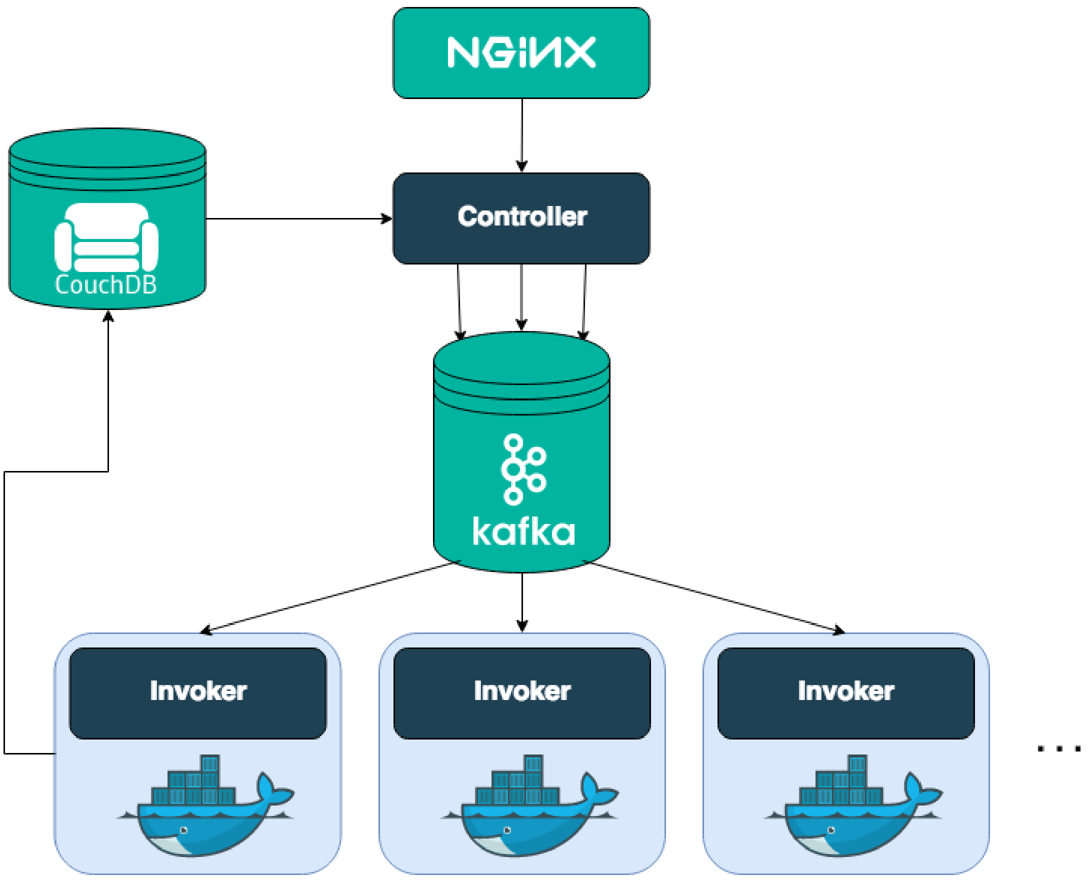


Figure 2.1: OpenWhisk architecture. Nginx is the system's main entry points, and forwards request to a Controller; the Controller then uses Kafka to communicate with the Invokers, and CouchDB to obtain the necessary information about the function and the user requesting it. Invokers then use CouchDB to store the function execution's result. Picture from `https://openwhisk.apache.org/documentation.html`.

and stability, and was written "specifically to address the performance limitations of Apache web servers"[3]. As a part of the OpenWhisk architecture, its main purpose is to expose the public HTTP endpoints and act as a reverse proxy and an SSL termination proxy; as such, requests made to the platform are received by Nginx and then forwarded to the next components.

Apache Kafka[36] is an open source distributed event streaming platform; its role in the OpenWhisk architecture consists in allowing Controllers and Invokers, two of OpenWhisk core components, to communicate. Requests coming from a Controller are first buffered as messages by Kafka, and subsequently delivered to the requested Invoker; the buffering phase is necessary, as Invokers may be momentarily busy, or may have crashed and be in the process of restarting, and as such be unable to process the message as soon as it's sent; for this purpose, Kafka's "publish and subscribe" architecture allows requests to be stored until they are ready to be executed.

CouchDB[34] is an open source, document-oriented NoSQL database, and it's currently a project of the Apache Software Foundation. Its purpose in the OpenWhisk architecture is to maintain and manage the state of the entire system, such as credentials, function definitions, triggers and rules. The Controller uses CouchDB for user authentication and authorization, and obtains the function code and metadata when a specific action is executed; the action's result is then also stored in CouchDB, as part of the `activations` database and identified by a specific *activation ID*.

Programming model

The OpenWhisk programming model is event-driven, and code can be run either in response to these events or to direct invocations. The main components of this model are *actions*, *triggers*, *rules* and *feeds*.

[3]`https://www.nginx.com/blog/nginx-vs-apache-our-view/`

Actions are the main entity in the OpenWhisk FaaS architecture, constituting the actual code being run. Actions are, at their core, stateless functions, generally written in one of the supported languages (actions can also be created directly from executables). Actions are *invoked* in response to events, and produce observable outputs; actions can also be invoked directly from the `wsk` CLI or from their endpoint as a REST API call. Every action invocation results in an activation ID, which can be queried to obtain results and logs in non-blocking invocations.

Triggers are similar to actions as they can also be activated (while the term for actions is *invoked*, the term for triggers is *fired*) and return an activation ID. Triggers can be either explicitly fired by a user, or on behalf of a user by an external event source (such as a *feed*).
When a trigger is fired without any accompanying *rule*, no visible output or effect is produced. Common examples of triggers are location updates or changes in a cloud storage service; they are used to define classes of events, which can in turn cause action invocations.

Rules are used to associate triggers and actions; a single rule associates one trigger and one action, with every firing of the trigger resulting in an invocation of the action, with the trigger event as input. By defining different rules, triggers can be used to invoke multiple actions, and multiple triggers can all result in the same action being invoked.

Feeds are specially crafted actions, which need to respect a certain architectural pattern. Feeds are used as event sources for triggers, and generally acts as a connection between a stream of events produced by a package and a user-defined trigger.

The OpenWhisk programming model and its components can be seen in Figure 2.2: an external event source is connected to a trigger by a feed; the former is then used to invoke an action according to a specific rule; the action then produces an observable result (such as a JSON output). Actions, triggers and rules are all divided in specific namespaces.

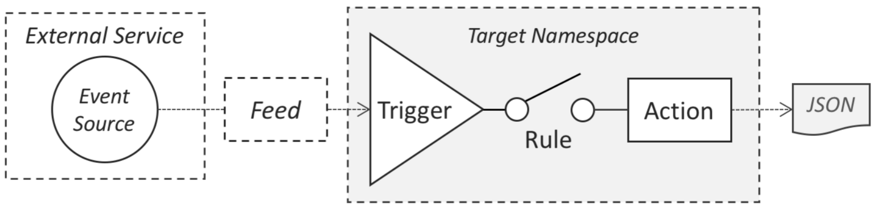


Figure 2.2: OpenWhisk programming model. Picture from `https://openwhisk.apache.org/documentation.html`.

Internal flow of processing

After an action is invoked, whether directly or in response to the firing of a trigger, a specific processing flow is started, involving all of the previously mentioned components; a representation of this flow can be seen in Figure 2.3.

1. Firstly, the action invocation is handled as an HTTP request against the system's entry point, **Nginx**. The proxy server analyzes the request path, and forwards it to the next component, the Controller; if more than one Controller is available (e.g. when multiple replicas are present), Nginx uses a round-robin load balancing algorithm, marking single Controllers as unavailable for the request after a specified timeout.

2. After that, the request is processed by the **Controller**; this component first has to disambiguate the kind of request that was received (e.g. an action invocation is a POST request), and then authenticates and authorizes it by verifying the user identity on **CouchDB**, in the `subjects` database.

   Following this, the Controller actually loads the action from the `whisks` database, along with possible metadata; the record also includes parameters in the invoke request, which are visible to the Controller.

3. Then, a component of the Controller, the **Load Balancer**, chooses a particular Invoker to execute the action; the Load Balancer has a global view of all Invokers, and so can choose an healthy one according to a specific load

balancing algorithm.

In case of multiple Controllers (and consequently, multiple Load Balancers) the default approach is to evenly distribute each Invoker's capacity (i.e. the maximum amount of memory of actions running in parallel on that Invoker) between all of them, while still maintaining the global view; e.g. if N Controllers and M Invokers are present, with each Invoker having capacity C, the resulting view for every Load Balancer is M Invokers with capacity N/C.

4. The Controller then uses **Kafka** to communicate with the chosen Invoker; this removes from both the Controller and the Invoker the responsibility of buffering the message in memory, and ensures the messages are not lost in case of a crash. In the case of a non-blocking invocation a response is sent for the initial HTTP request by the user, containing the relevant activation ID.

5. After the message has been successfully delivered, the next phase is handled by the **Invoker**; a Scala-based component, like the Controller, its role is to execute actions in an isolated way by using Docker containers.

   When an action is invoked, Docker is used to setup a new container, in which the action code is injected and executed; after the code has finished its execution, the result is obtained and the container is destroyed. The result is then stored in the `activations` database in CouchDB, along with the action logs.

Load balancing algorithms

- As previously mentioned, Nginx uses a simple round-robin load balancing algorithm; as such, requests are evenly distributed between multiple Controllers, with no specific priority rules. Controllers are ignored only if they result unavailable at the time of the request. As shown in Section 3.1, in this thesis the Nginx default configuration was modified, allowing users to prioritize certain Controllers in their invocations.

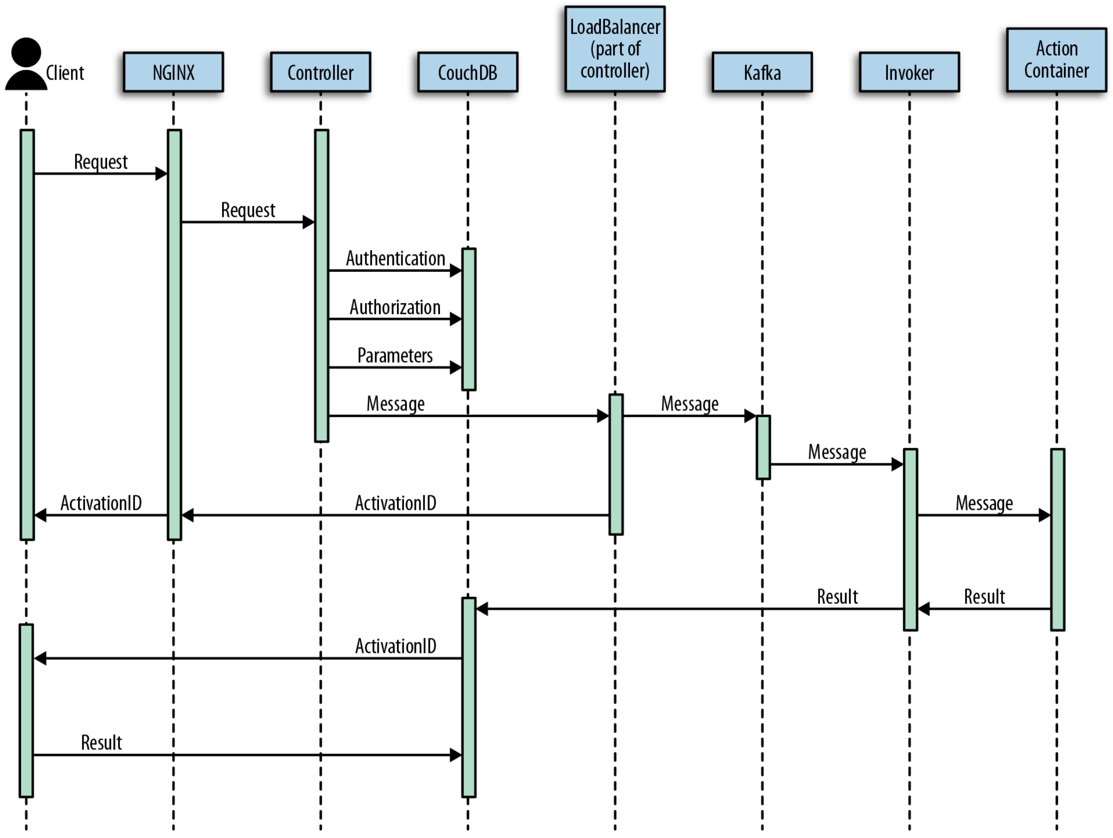


Figure 2.3: How OpenWhisk processes an action. Picture from "Learning Apache OpenWhisk"[24].

- The Controller's default load balancing algorithm is currently described in the project's code and documentation on GitHub[4]; what follows is a summary of that description.

  Firstly, when an invocation request is received, a hash is calculated from the action's name and namespace; the hash is then use to pick an Invoker (identified in the Invoker list by a numeric index); the formula used is `hash % numInvokers`. The resulting index is the "home Invoker", from which the following progression will start.

  If the home Invoker is healthy (i.e. no more than 3 out of the last 10 activations on that Invoker contained system errors, and the Invoker has successfully sent a periodic ping to the Load Balancer before a certain timeout) and has sufficient capacity to handle the action, then the request is scheduled to it.

  If one of these conditions is not met, the index is incremented by a step-size; the available step-sizes are all the coprime numbers smaller than `numInvokers`, and a specific one is selected using the formula: `hash % numStepSizes`; the Invoker relative to the new index is then both health-checked and capacity-checked, verifying its availability for the action invocation.

  The procedure is repeated until all Invokers have been checked, after which a random healthy Invoker will be selected; if none are available, the Load Balancer returns an error, and the request is not queued.

  The main logic behind this approach is that, since a certain action in a certain namespace will *always* generate the same hash and progression, the Load Balancer will generally schedule functions on Invokers that already have a "warm" container; as also noted in the code, this is not always true: for instance, two computationally heavy actions can override each other on a specific Invoker, with each successive invocation resulting in a cold start (as the previous warm container was evicted to make space for the other function).

[4]`https://github.com/apache/openwhisk`

- The modified load balancing algorithm used as a starting point for this thesis was defined in [6]; in the paper, a configurable Load Balancer was implemented, allowing users to prioritize specific Invokers for specific function classes; different strategies for selecting from the list of Invokers are available, along with the possibility of default to the standard load balancing algorithm as a fallback. The load balancing algorithm used in this thesis largely follows the same principle, while allowing for more flexibility in selecting Invokers that are "nearby" a specific Controller.

2.2.2 Fission

Fission[40] is an open source serverless framework, designed for Kubernetes; like OpenWhisk, it has an active community on GitHub, with 6.6k stars and over 120 contributors. Unlike OpenWhisk, its core is written in Go, and has a less complex stack of additional components; this is because Fission is heavily dependent on Kubernetes features, and is meant to be run and installed on a Kubernetes cluster, while OpenWhisk on the other hand can be deployed in various different ways (including a "standalone" option).

Programming model

Fission programming model is built upon three main concepts, visible in Figure 2.4: functions, environments and triggers.

Functions are, analogously to OpenWhisk's actions, the actual code that is executed by Fission; like actions, functions are generally stateless and event-driven programs, written in one of the supported programming languages. Functions generally have one entry point, which is an asynchronous function written according to a certain interface (e.g. in NodeJS, the entry point receives a `context` input, containing information such as the HTTP request body and headers).

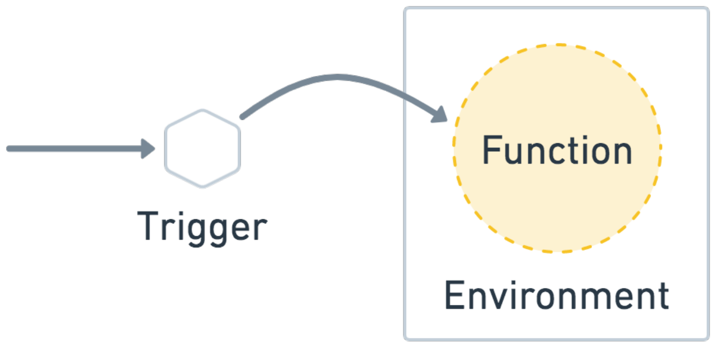


Figure 2.4: Fission core concepts: functions can be invoked according to triggers connecting them to certain events; functions are defined and isolated in environments, depending on the required language runtime. Picture from `https://fission.io/docs/concepts/`.

Environments are the basis for function deployment; they consist in a pool of containers, defined by a Docker image which has an HTTP server and a dynamic loader for the selected language; environments can also include a builder image and builder command, which will be used for building from source code (instead of uploading all dependencies as a single archive). When functions are created, an environment is specified, along with the function name and code.

Triggers are the connection between certain events and the invocation of a function; various types of triggers are available, such as HTTP triggers (in response HTTP requests), Kubernetes Watch triggers (in response to changes in the cluster) or Message Queue triggers (in response to services such as Kafka, with a publish-subscribe approach). It has to be noted that, regardless of the trigger, all functions are invoked through HTTP requests; for instance, in Figure 2.5 the trigger detects a message from the external service (to which it is subscribed), and sends a POST request to invoke the assigned function.

In practice, all of these abstractions are Kubernetes **CRD**s (Custom Resource Definitions); as previously mentioned, this completely integrates Fission with the orchestration system.

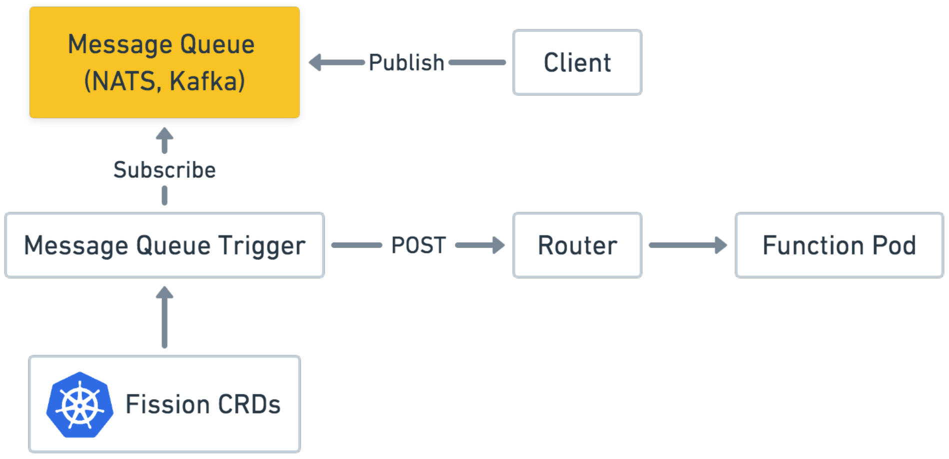


Figure 2.5: Example of a Message Queue trigger; when a message is received, the trigger sends a POST request to invoke the connected function. Picture from `https://fission.io/docs/usage/triggers/`.

Architecture

The Fission architecture is described in the project's documentation[5], and is divided in *Core Components* and *Optional Components*; while all of the elements in both categories play a role in Fission's processing flow, four of them are central to the invocation and scheduling of functions, and are here described.

The **Controller** is the component that the Fission CLI talks to; it keeps track of all functions, routes, triggers and environments, while containing their CRUD APIs; the Controller also contains proxy APIs to communicate with internal 3rd-party services. A representation of the Fission Controller flow is given in Figure 2.6.

When an HTTP requests is instead sent to invoke a function, the **Router** is used to forward it to the right pod (i.e. a default Kubernetes resource consisting in a group of containers and a specification on how to run them); depending on

[5]`https://fission.io/docs/architecture/`

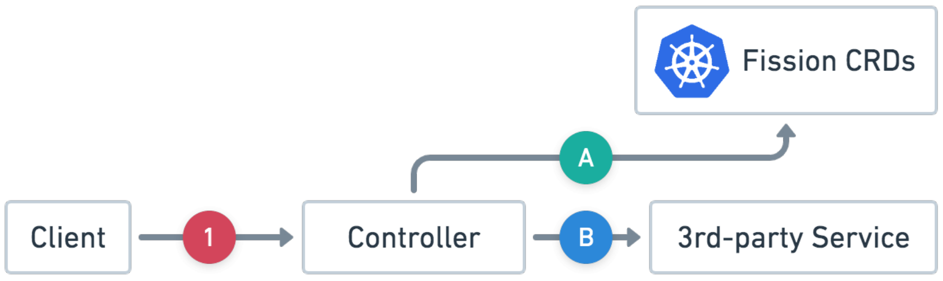


Figure 2.6: How the Fission Controller works: (1) The client send requests to endpoints on Controller, (2) The Controller either (A) operates CRDs according to the request or (B) proxies the request to an internal service. Picture from `https://fission.io/docs/architecture/controller/`.

the Executor type, the Router can either:

- First check whether a function service record exists in its cache (and forward the request to the relevant pods); if no running service is found, one is requested from the Executor; once the service is ready, the original request if forwarded to its pod.
- Directly forward the request to the Executor

The Router processing flow is shown in Figure 2.7.

The **Executor** is the component used to launch Kubernetes pods for functions; when a request for a function service is sent to the Executor, the component retrieves function information from the Kubernetes CRD and invokes one of the Executor types to "spin up" the pods; once these are up, a function service record containing the relevant address is returned. At the time of writing, Fission supports two Executor types:

- The **PoolManager** watches the environment CRD for changes, and creates pools of generic containers for each environment; these generic containers

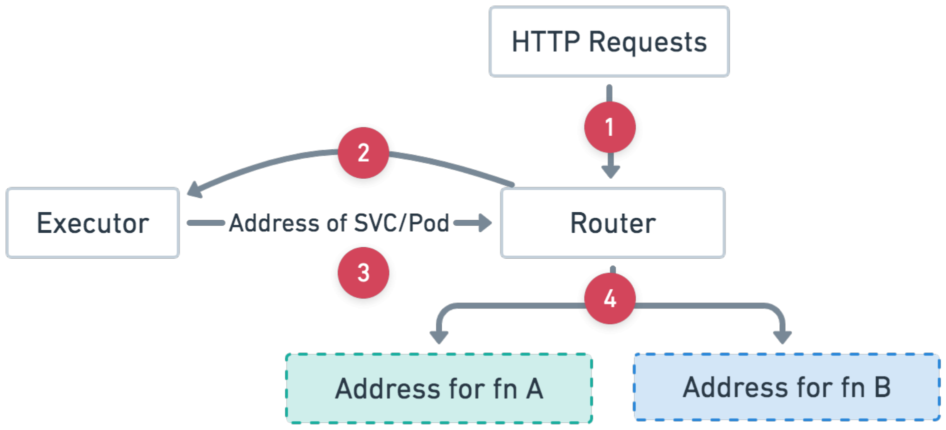


Figure 2.7: How the Fission Router works: (1) The client send requests to a specific URL, (2) The Router returns 404 is no matching HTTP Trigger exists; otherwise, it either (3) asks the Executor for the Function service address or (4) forwards the request to the address found in its cache. Picture from `https://fission.io/docs/architecture/router/`.

are then "specialized" when a the Executor is asked for the service address of a function, loading the function code inside the container. After a certain idle duration, the function pod is cleaned up, and subsequent requests to the function will cause a new pod to be specialised from the pool.
With the PoolManager type, no router-side caching is used, and requests are directly forwarded to the Executor. The Fission documentation advises the use of this strategy for short-living functions that need a short cold start time.

- The **NewDeploy** type instead creates a Kubernetes *Deployment*, along with a *Service* and an *HorizontalPodAutoscaler* (HPA); this enables the autoscaling of function pods, and the load balancing of requests between pods.
The replicas of a function deployment are initially scaled to a minimum amount (defined by a tunable setting), and are further scaled up according to user-defined conditions (such as in response to traffic spikes); after a certain idle duration, unused pods are cleaned up. The Fission documentation

advises the use of this strategy for function designed to serve massive traffic; if latency requirements are stringent, the minimum amount of replicas can be set higher than zero to keep some pods ready when a function is created (without clearing them if they are idle). This approach minimizes latency at the cost of resource consumption, as the pods are kept running regardless of the amount of traffic.

The two different Executor types are also shown in Figure 2.8.

Finally, the **Function Pod** is responsible for the actual execution of the function; the pod is made from two containers, the Fetcher and the Environment Container, with a shared volume. The **Fetcher** pulls the function's deploy archive from the storage service, and then saves it to the shared volume; then, the **Environment Container** loads the function from the volume and starts serving the fowarded requests from the Router. The structure of the Function Pod is also shown in Figure 2.9.

The other components, not described here, are the **Builder Manager** and **Builder Pod**, the **Storage Service**, the **Logger**, the **KubeWatcher**, the **Message Queue Trigger**, and the **Timer**.

2.2.3 OpenFaaS

OpenFaaS is an open source serverless framework, currently the most popular on GitHub in terms of stars from the community (almost 21k), and one of the most active in terms of contributors (around 160, with many of them having over 20 commits); OpenFaaS main aim, similarly to other frameworks, is to provide developers with an easy to use platform where they can deploy event-driven functions.

OpenFaaS is written in Go, like Fission, and is part of the so-called "PLONK"[6]

[6]`https://www.openfaas.com/blog/plonk-stack/`

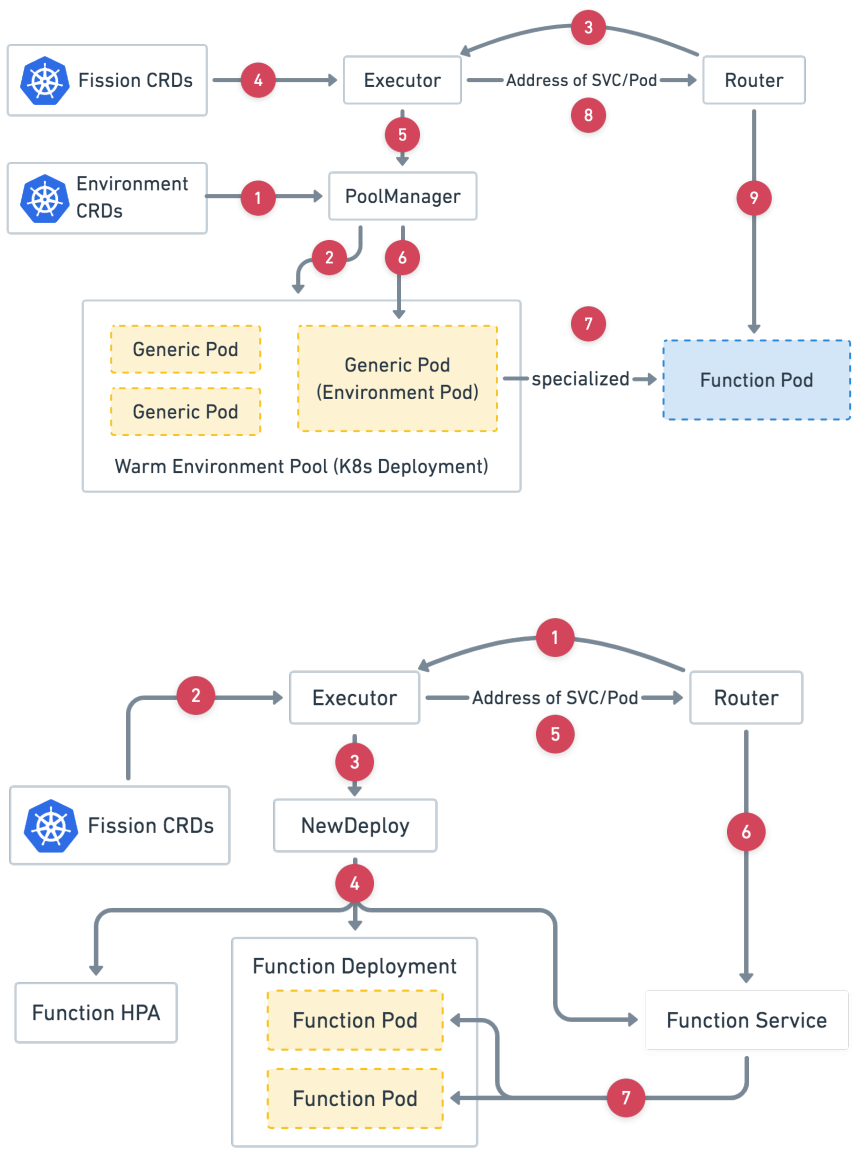


Figure 2.8: The *PoolManager* (top) and *NewDeploy*(bottom) Executor types. Picture from `https://fission.io/docs/architecture/executor/`; a description of each step in the processing flow is available on the same page.

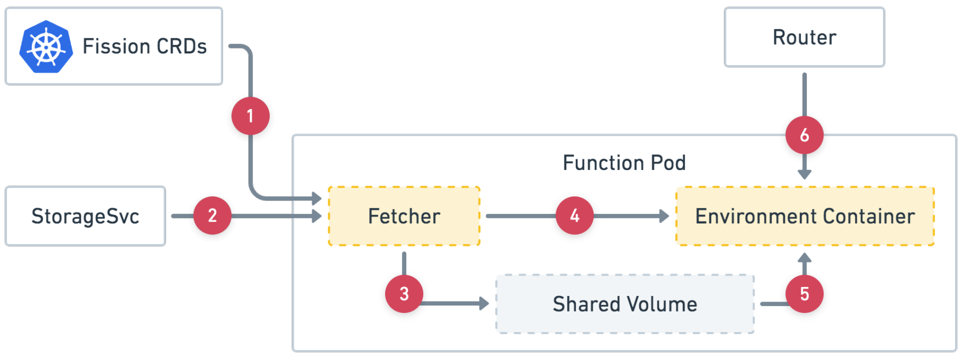


Figure 2.9: How the Function Pod works. Picture from `https://fission.io/docs/architecture/function-pod/`; a description of each step in the processing flow is available on the same page.

Stack, composed by Prometheus[54] (metrics), Linux/Linkerd[48] (service mesh), OpenFaaS, NATS[49] (message bus/queue) and Kubernetes; since Linkerd is technically an optional part in the stack, a description of the service's role is not given in this section. While the stack only lists Kubernetes, the platform is also deployable on a single host with `faasd`[39], a lighter version of the same solution; a deployment option for Docker Swarm was also present, but has been deprecated by the authors.

The deployment used in this thesis in the one on Kubernetes, executed using the official `faas-netes`[7] provider developed by the platform's authors.

Triggers and function deployment

Similarly to other frameworks, OpenFaaS functions can be triggered by various kinds of events; unlike Fission and OpenWhisk though, OpenFaaS does not have an internal definition of triggers as a specific resource. The general way of using triggers in OpenFaaS is through the **event-connector pattern**[8], where a

[7]`https://github.com/openfaas/faas-netes`
[8]`https://docs.openfaas.com/reference/triggers/`

separate microservice maps functions to topics, and invokes functions using the OpenFaaS Gateway; examples of these connectors are the `cron-connector` and the `nats-connector`; the latter is an alternative to using the built-in queue system based on NATS Streaming, and uses the publish/subscribe mechanism of NATS. The "standard" way of invoking functions is still via HTTP requests, like all other platforms.

Unlike both previously described frameworks, OpenFaaS handles the deployment of functions by using Docker/OCI[52]-format images; the function image is built and pushed to a repository of choice, and is then deployed using the platform's CLI `faas-cli`; still, multiple language runtimes are supported, with the chosen function's runtime being specified in the build files.

Additional components

As mentioned in the introduction, OpenFaaS is part of a technology stack that includes Prometheus and NATS; while both of these are not part of the core OpenFaaS architecture, they are used in the framework's flow of processing for autoscaling and asynchronous invocations.

Prometheus is an open source monitoring system, which is used by OpenFaaS to collect metrics and handle autoscaling; more specifically, the Prometheus AlertManager reads metrics regarding the number of requests per second to a specific function, and fires an alert to the platform's API Gateway according to a configurable rule. Other configurable options for the autoscaling functionality include the minimum and maximum number of replicas, and the percentage of the maximum number of replicas to be deployed with each alert; the scale-to-zero functionality is also present (i.e. bringing a component down to zero replicas if it has been idle for some time), but it's not available in the standard version of the platform (the `faas-idler` component is only available with OpenFaaS Pro, the paid version of the framework).
While OpenFaaS uses Prometheus for its built-in autoscaling mechanism, the Ku-

bernetes *HorizontalPodAutoscaler* (HPA) can also be used instead; this allows functions to be scaled depending on the CPU and memory utilization on single nodes, instead of the amount of inbound requests in a time span.

NATS is "a connective technology that powers modern distributed systems"; at its core, it's a message queue system based on a publish/subscribe mechanism. OpenFaaS uses another service, **NATS Streaming**[9], built on top of the NATS protocol; NATS Streaming adds features such as *at-least-once* delivery of messages, in contrast with the *at-most-once* policy of the standard NATS system (i.e. messages are not persisted in memory or secondary storage, and there is no re-delivery).
In the OpenFaaS system, NATS Streaming is used to handle asynchronous invocations: initially the request is serialized and enqueued by the Gateway when it is received, waiting to be processed at a later moment; after this, when the function is ready to be invoked, the `queue-worker`[10] component acts as a subscriber and dequeues the request, sending it to the actual container (either directly or via the Gateway, using a synchronous call).
The comparison between synchronous and asynchronous invocations can be seen in Figure 2.10.

Core Architecture

The core architecture of OpenFaaS is built on two main components: the API Gateway and the watchdog for each function; the way the various parts of the infrastructure interact is shown in a diagram in Figure 2.11, which includes the additional components described in the previous paragraph.

The **API Gateway** provides the external route to the functions and collects metrics through Prometheus; the autoscaling of functions is also handled by this component. The Gateway interacts with the provider (in our case, `faas-netes`)

[9]`https://nats.io/blog/introducing-nats-streaming/`
[10]`https://github.com/openfaas/nats-queue-worker/`

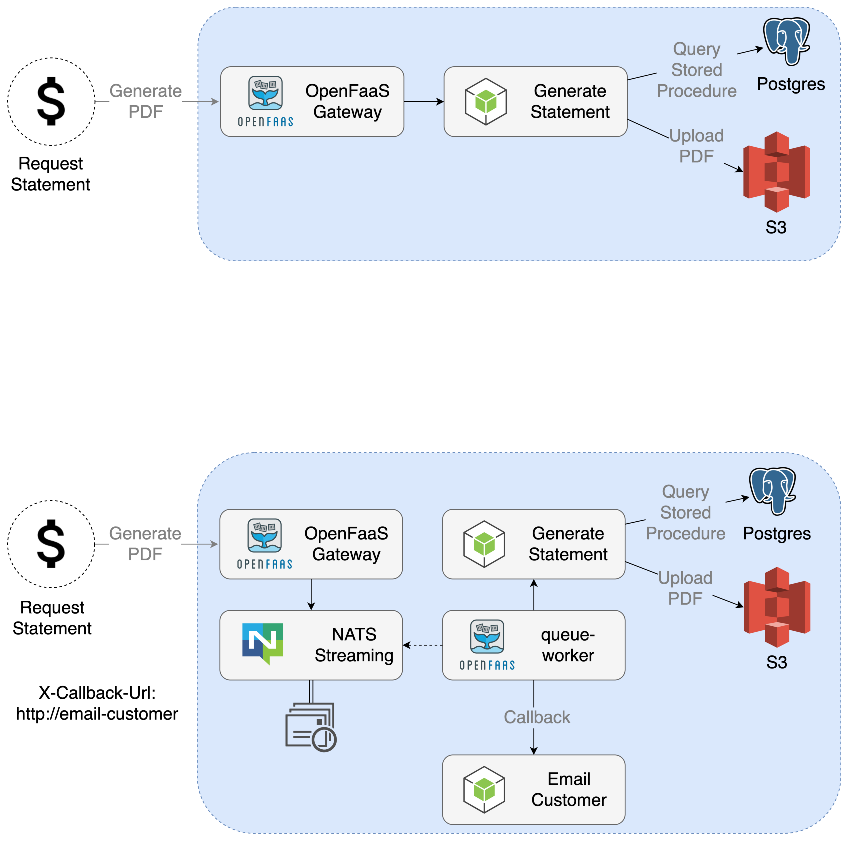


Figure 2.10: Comparison between synchronous (top) and asynchronous (bottom) calls in OpenFaaS: in the synchronous case, a connection is established between the components at each step; in the asynchronous case, NATS Streaming is used to store the messages until the `queue-worker` requests a dequeue and forwards them to the function. Pictures from `https://docs.openfaas.com/reference/async/`.

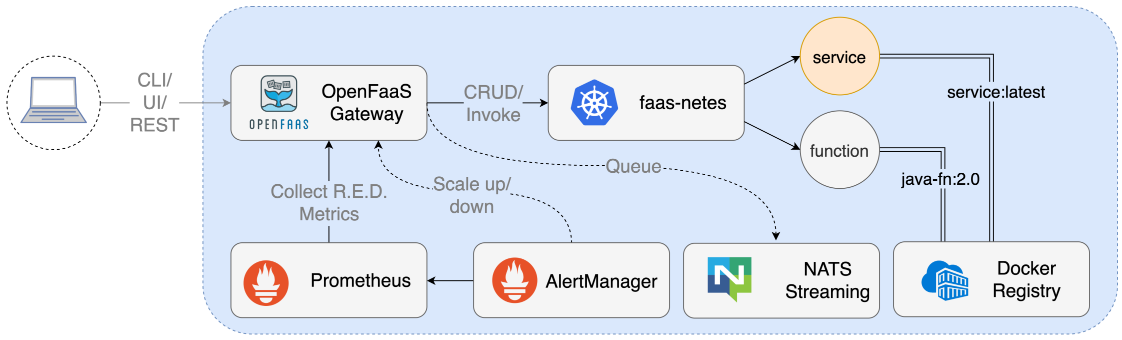


Figure 2.11: The OpenFaaS processing flow; while not shown in the diagram, the watchdog is included in the "function" block, since the function's process never interacts directly with the outside world. Picture from `https://docs.openfaas.com/architecture/stack/`

both for CRUD operations and invocations; the former type of interaction is required only in deployments where the `operator` mode was specified for the provider, causing functions to be defined via Kubernetes CRDs; in this case the functions are also manageable using the `kubectl` CLI for Kubernetes, resulting in a more integrated version of the framework, similar to Fission or Kubeless.

Additionally, for every function a **watchdog** is used as the actual endpoint where requests are forwarded; the watchdog embeds a lightweight HTTP server, translating the communication flow to and from `stdin` and `stdout` respectively. The actual function is a process inside the container, which uses the embedded webserver to interact with the outside world; as such, the container image built when the function is deployed actually contains the watchdog service, with the function code as a forked process to be launched when the container is started. A representation of the watchdog's role is shown in Figure 2.12.

2.2.4 A note on Kubeless

At the time of writing, Kubeless is one of the most popular open source serverless frameworks (6.8k stars and over 100 contributors) on GitHub, along with

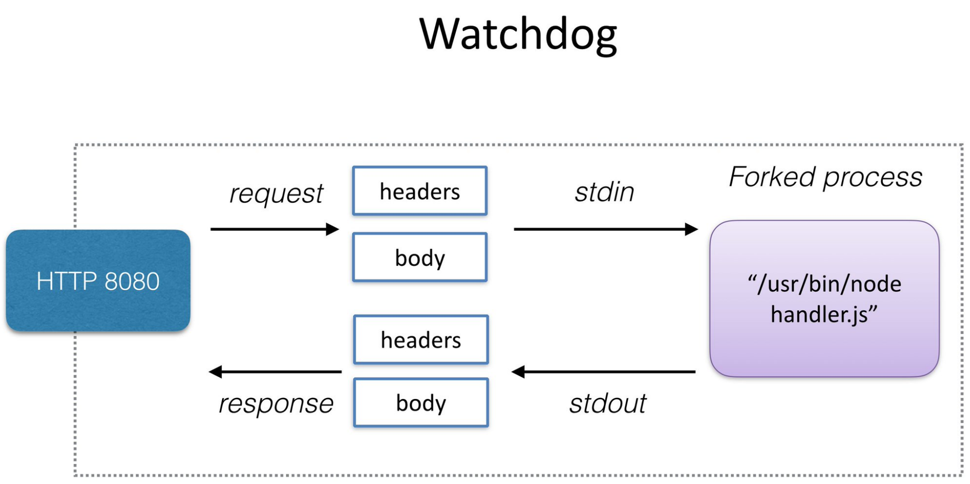


Figure 2.12: The OpenFaaS watchdog; while the function writes to `stdout` and reads from `stdin`, the embedded HTTP server translates this to responses and from requests respectively. Picture from `https://docs.openfaas.com/architecture/watchdog/`

OpenFaaS, Fission and OpenWhisk. The main idea behind Kubeless is, not unlike Fission, to offer a Kubernetes-native serverless framework, using the orchestrator capabilities to achieve autoscaling, monitoring and load balancing. Similarly to Fission, Kubeless defines its main entities with Kubernetes CRDs, integrating functions, http triggers and cronjob triggers as Kubernetes resources.

Unfortunately, since August 2021 Kubeless is no longer maintained; this fact, along with compatibility issues with the latest version of Kubernetes, led to the decision of discarding Kubeless in favor of Fission for the purpose of this thesis, as its role in a performance comparison might not be as relevant. The authors stated on GitHub that a decision about the future of the platform will be taken by the end of 2021.

Chapter 3

Contributions

In this chapter, a description of the main contributions of the thesis is given; as the aim of this work is both to increase the flexibility of the Configurable Load Balancer implemented in [6] and to introduce an automated topology-based mapping between Controllers and Invokers in OpenWhisk, the chapter is divided in three main sections:

- First, the approach chosen to handle multiple Controller replicas is described; this mainly required modifications to Nginx's behaviour, and the addition of a *watcher* service in the platform deployment on Kubernetes.

- Then, the contributions regarding the topology-based Invoker distribution are listed; this required changes to the Load Balancer and also the use of the previously mentioned *watcher* service.

- Finally, a description of the additions to the configuration language is given, along with the different behaviours of the platform according to the configuration options.

3.1 Multiple Controller load-balancing

3.1.1 Motivation

As mentioned in the introduction, the main starting concern of this work was to extend the flexibility and the scalability of the approach described in [6]; as that modification to the OpenWhisk platform proved, the ability to configure the specific node where a function should be invoked can result in a visible advantage in terms of data locality, especially in hybrid clusters (where the latency between nodes can vary considerably).

One downside of that work was considering each "worker" as a single node, resulting in a lack of flexibility when a single zone of the cluster could possibly be composed of dozens of nodes; in that case, the configuration would have either needed to account for each node in the desired zone, or ignored many of them (thereby obtaining a suboptimal result).

For this reason, the possibility of using groups of Invokers in the configuration was explored; to do this, a simple way was to assign a Controller replica to each zone, and subsequently assign Invokers to the various Controllers in an automated way (more on this in Section 3.2); after this, the main concern was instructing Nginx to forward requests to the required Controller, which would then have the responsibility of scheduling the function execution on one of its Invokers.

3.1.2 Upstream modifications

The first necessary modification was applied to Nginx's configuration file; this was modified in the framework's deployment on Kubernetes, where the webserver's configuration is defined as a ConfigMap[1].

[1]`https://kubernetes.io/docs/concepts/configuration/configmap/`

In the default setup, only one *upstream* (i.e. a group of servers, to which requests can be forwarded) is defined, including all Controller replicas and using a round-robin load balancing algorithm. In this thesis, multiple upstreams were defined, dynamically generated according to the number of replicas; each upstream would have a specific Controller as the main server and the others as backup, in case of issues with the main one. Thanks to this setup, requests could be forwarded to a specific upstream, which would then use the main Controller as the default destination, while still being able to fall back on the other replicas and maintain availability.

After these modifications, Nginx's configuration file defined N+1 upstreams, with N being the number of Controller replicas; every upstream has its main server's name (i.e. the replica's name, such as `controller-0` for the first one, `controller-1` for the second one and so on), except for the default upstream (which keeps the `controllers` name). This configuration was chosen to allow both the configured (where invocations can be redirected to a specific Controller) and the default behaviour (where the round-robin load balancing algorithm is used on all replicas, with no server used as backup).

3.1.3 Watcher service

To allow a correct mapping between the nodes and the Controller names, two possible choices were considered:

- Give more permissions to an existing service, allowing it to access cluster-level properties such as node names and labels.
- Implement an additional service, giving it the required permissions and having it act like an API for the already existing services and pods.

As the former option might present security issues by giving cluster-scoped permissions to externally accessible services (such as Nginx), the latter approach

```
{
  "azureworker": "owdev-controller-0",
  "azureworker2": "owdev-controller-1"
}
```

Figure 3.1: Example of a node-pod map output from the Watcher service; as described, it's a simple JSON mapping, used by the other services to avoid querying the Kubernetes API.

was utilized; the additional service was called `watcher` in the deployment, and simply consists of a periodical query to the Kubernetes API, asking for pod names and the respective nodes where they are deployed. The resulting output is parsed using *jq*[44] and written on a JSON file; the file is then stored in a shared volume between Nginx, the Watcher and the Controllers. This way, the service with the highest level of permission in inaccessible from outside, and the other services only interact with it using the shared volume, and reading only a selected subset of the information available via the Kubernetes API.

The output of the Watcher service is used by Nginx to connect the upstreams (which are named like the various Controller pods) and the nodes (which have their own names, and can dynamically host different Controller replicas depending on the deployment); this way, the user can configure functions to be scheduled by specific Controller-hosting *nodes*, without the need to specify which Controller *pod* they wish to target (the position of which is more susceptible to changes).

3.1.4 Extending Nginx behaviour: njs script

As a way to extend its functionality, Nginx offers *njs*[2], a subset of the Javascript language; scripts written in this language interact with requests and responses alongside the server's normal behaviour, performing actions such as security checks

[2] `https://nginx.org/en/docs/njs/index.html`

and redirects.

In this work, a script written in njs was used to analyze all invocations passing through Nginx, in order to extract tags from the request parameters and compare them with the user-defined configuration file; if the extracted tag matches a configuration option, the resulting node name is used as a key in the mapping produced by the Watcher service, obtaining the desired upstream's name.

As this part of the processing flow has to be as lightweight as possible, the configuration file and the mapping are only read and loaded if a tag is specified in the invocation; the variables in the request body are always available to Nginx and can be easily parsed as JSON and used in the script.

From the user's point of view, the only change in the processing flow is the addition of a `tag` parameter in the invocation if they wish to schedule on a specific Controller; otherwise, the request is simply forwarded to the default upstream with no additional computations.

3.2 Topology-based Invoker distribution

3.2.1 Motivation

The main reason for this addition was to implement the natural completion of the previously described Nginx modifications; that is, after allowing users to select a specific Controller to handle their invocations, having all Controllers prioritize scheduling on the nearest Invokers, effectively grouping them and allowing users to target Invoker sets instead of single workers.

The concept of "nearest Invokers" was defined in practice with the use of **topology** labels on Kubernetes; these are specifically named labels, assigned to nodes by the cluster administrators, which can be used as keys in various ways (e.g. defining where pods should be scheduled); they also offer an intuitive way to describe the cluster structure, assigning nodes to specific regions and zones.

In this work, each node in the cluster was assigned a specific *zone* label, depending on its geographical location; after this, Controller replicas were distributed with a special *anti-affinity rule*: the pods should not be scheduled on nodes in the same zone, and should first be distributed amongst all zones. This rule was kept as something that is *preferred* during scheduling, instead of required, as the number of zones can be lower than the number of Controller replicas (e.g. if redundancy is required in specific zones); this possibility is also considered in the Load Balancer during the Invoker distribution.

3.2.2 Use of the Watcher service

Since topological information is stored in the form of node labels, the Controller needs to be able to access the nodes' metadata to handle the Invoker distribution correctly; as with the Nginx situation, extending the Controller pod's permissions (allowing it to read cluster-scoped data such as node labels) seemed an unsafe approach, so the Watcher service was utilized in a similar fashion. In this case, the outputted file contains a mapping between nodes and topological zones; similarly to Nginx, the Controller replicas can only access this limited information about nodes, and lack the necessary permissions to query the Kubernetes API directly. The mapping is produced periodically, to account for the addition and modification of nodes.

3.2.3 Invoker distribution

When the platform is deployed, every Load Balancer (each associated to a Controller replica) obtain information about the currently "healthy" Invokers and their metadata; each Invoker has the name of the node it's currently deployed on, and this information is visible to the Load Balancer. When a new Invoker joins the set, its name is used to retrieve the relevant node's topological zone (inspecting the output from the Watcher service); the Load Balancer then acquires control of the Invoker's memory, sharing it with all other Load Balancers in the same topological zone; Invokers in different zones are handled by requiring a configurable portion of their memory (similarly to the default behaviour, where every Load Balancer has access to a fraction of all Invokers' memory).

Different policies are available for Invoker distribution, and one of them can be required during deployment:

- The `shared` policy is to maintain the original behaviour, but give full control over Invokers in the same topological zone; this is prone to **Invoker overloading** (i.e. the required memory from the various Load Balancer is higher than the available memory on the Invoker, which may result in longer queues) and **resource grabbing** (other Load Balancers can schedule on Invokers outside their zone, effectively taking the resources away from the nearest Load Balancer); since Load Balancers prioritize scheduling on the nearest Invokers, the ideal situation is a scheduling pattern where each zone is completely exploited, and resources are borrowed only when a smaller zone becomes saturated, and a larger zone has idle nodes.
- The `min_memory` policy is similar to the previous one, but aims to minimize both overloading and resource grabbing by making Load Balancers require only a minimal fraction of Invokers' memory from other zones; the fraction (referred in code as `MIN_MEMORY`) is the minimal memory necessary for one invocation (256MB). When Invokers have no Controller in their topological zone, or no topological zone at all, their memory is shared between all Load Balancers. This policy is safer in terms of overloading, but can create situations where smaller zones quickly become saturated, and are therefore unable to handle requests.
- The `isolated` policy gives complete control over Invokers in the same topo-

logical zone, but no access over all other Invokers; no resource grabbing and no overloading are present, and Load Balancers only schedule on the nearest Invokers; this is a less flexible approach compared to the previous two, and can exacerbate situations where functions are often scheduled in smaller zones; nevertheless, it's the safest options in that it allows all Load Balancers to have a realistic representation of the available memory on their Invokers. Similarly to the previous option, Invokers with no assigned Controller are shared between all Load Balancers.

- The `default` policy simply maintains the original sharing model, where Load Balancers have access to a fraction of all Invokers' memory; the topology-based priority is still present, but no complete memory availability is granted.

When it comes to actually scheduling the invocations, the Load Balancer uses the policies defined in [6] (`best-first`, `random`, `default`, `next-coprime`), and if no workers are explicitly specified in the configuration file, prioritizes Invokers in its topology zone. This approach is used for all policies except for `best-first`, where a list of workers has to be specified.

3.3 Configuration language extension

3.3.1 Motivation

Since the work of this thesis is mostly built upon the results obtained in [6], extending the configuration language used in that paper was preferred to defining a completely new language; as the functionalities added in this thesis do not substitute the ones already defined, only two simple additions were necessary: the new `controller` tag and the concept of `topology_tolerance` for topology-based scheduling. An example of a configuration file written with the extended language can be seen in Figure 3.2.

```
default:
  - controller: "*"
  - workers: "*"
    strategy: random
    invalidate:
      - max_concurrent_invocations: 100
      - capacity_used: 50
mongoDB:
  - controller: "AzureWorker2"
    topology_tolerance: "same"
  - workers: "*"
    strategy: random
    invalidate:
      - max_concurrent_invocations: 100
      - capacity_used: 50
  - followup: 'default'
```

Figure 3.2: Example of a configuration file written with the extended syntax; both the `controller` and the `topology_tolerance` parameters are used.

3.3.2 Controller Tag

The first addition to the configuration language is a new tag, `controller`; intuitively, this specifies the Controller node to be used for scheduling the invoked functions bearing a certain tag. The option to specify all Controllers is also given, with the standard "*" symbol; in case all Controllers are specified, both Nginx and the Load Balancer simply ignore the specification; it has to be noted that while the behaviour is correct, specifying a tag for an invocation with no Controller specification (or equivalently, using the "all Controllers" option) has an impact on performance, as Nginx still has to load the configuration file in response to the provided tag.

In case both the Controller and the Invokers are specified (using the `workers` field), the Load Balancer's behaviour is to prioritize the specified workers instead of using the same-topology policy; this allows users to still maintain the behaviour defined in [6], and invoke functions only on the specified nodes; similarly, in no Controller is specified and workers are specified, the same rule applies.

To summarize, the topology-based scheduling is only used when a Controller is specified, and when the `workers` field is "*"; this instructs the Load Balancer to schedule the invocation on Invokers in its topology zone, with various levels of tolerance in case the required Controller is unavailable, and the function is being handled by a different one.

3.3.3 Topology tolerance

The `topology_tolerance` tag was added to handle cases where the specified Controller is unavailable, and the invocation is forwarded to another Controller's Load Balancer; in these situations, different policies were defined, depending on the user's needs:

- The `all` tolerance policy, indicating that the request will be handled with a topology-based approach, using the Invokers in the same topology zone of the Controller handling the invocation; this is the most tolerant option, requiring no adaptation from the Load Balancer.
- The `same` tolerance policy, which instructs the Load Balancer to schedule the invocation on the Invokers in the specified Controller's topology zone; this was designed for functions with no strong requirement to be handled by a single Controller, but still requiring invocation in a certain zone (e.g. for data locality reasons).
- The `none` tolerance policy, which forces the Load Balancer to drop the request if it gets forwarded to the "wrong" Controller; this simply creates a stronger constraint compared to the previous policy, and avoids the function being handled by any non-specified Controller.

When testing the OpenWhisk platform, all tolerance options were compared to better understand how they impact performance when a large number of requests are sent.

Chapter 4

Platform performance analysis

This chapter contains a description of the test suite used to evaluate the different Serverless frameworks that were deployed for the purpose of this thesis, and the results for each test case. The first section contains a description of the general structure of the suite, along with an explanation of the purpose and content of the single cases; the second section contains the actual results for all cases on the different frameworks.

4.1 Test design

The test suite is divided in three subsets:

- **Basic cases**, which are small functions that don't perform a realistic operation; they have no interaction with external services, and simply represent examples of possible parts of real applications.
- **Realistic use cases**, which are functions taken from the Wonderless[8] dataset to represent realistic applications of serverless computing (and FaaS in particular).

- **Critical cases**, which are similar to basic cases, but target specific issues of serverless computing (cold starts for heavy functions, data locality for large queries and scale-to-zero for idling functions).

4.1.1 Basic test cases

- The **hellojs** function consists of a simple parametric *Hello World* application; it should showcase the frameworks' performance with very simple functions with a semi-fixed behaviour (only returns a string, but has to evaluate and parse parameters).

- The **mongoDB** function executes a query of a single document from a remote mongoDB database; the document is very lightweight (only three JSON fields), an as such has little impact on the function's performance. This test case should showcase the frameworks' performance when the access to delocalized data is required.

- The **sleep** function is a simple parametric `sleep` command, instructing the container to wait a certain number of seconds. This test case shows the frameworks' ability to handle functions running for several seconds, minimizing response errors if many of them queue up.

- The **matrixMult** function multiplies two matrices, returning the result to the caller. This is an example of a computationally heavy function, showing the frameworks' base performance in this case. For the purpose of these tests, two matrices of 100x100 size are used as input.

All the basic cases are implemented in Javascript, and use the respective framework's `nodejs` environment; the `mongoDB` test case uses the `node12` environment in every framework, for compatibility reasons.

4.1.2 Realistic use cases: Wonderless dataset

For the realistic use cases, the Wonderless[8] dataset was used; this dataset contains a total of almost 2000 repositories, scraped from GitHub by searching for projects developed using the Serverless Framework[55], and was created with the purpose of being a data source for further research in the serverless ecosystem. The dataset is available both as an archive and as source code, and is divided in seven different folders: *AWS, Azure, Cloudflare, Google, Kubeless, OpenWhisk, Other.*

The main advantage of using such a dataset is that the code it contains comes from actual repositories; this allowed us to extend the test suite with existing code that has a practical use, instead of developing ad-hoc applications (which may end up being very similar to the basic scenarios).

The structure and distribution of Wonderless are very unbalanced: out of all the repositories, 1836 are in the *AWS* folder, consituting a very significant majority of the available code; despite this, because of the scale of this folder, priority was given to the other six, as the main purpose of this phase was to extract a small number of realistic use cases from the dataset; a complete analysis of the entire dataset is beyond the scope of this thesis, and will be reserved for future work.

Selection

Since even the reduced portion of Wonderless considered was still composed by many different projects, a set of rules was defined for this work in order to filter out a vast majority of the repositories; what follows is a list of the rules used, with Rule 0 being the only one applied automatically, and the respective reason for each of them.

0. The project must have a `README.md` file, and must not have more than 10 immediate subfolders (excluding hidden and common auxiliary folders, i.e.

img, *env*, *screenshot*, *doc* and the respective plurals). Both rules were applied automatically before any further inspection.

The reason for the first rule is to filter out repositories that contain no explanation on their inner workings; as the nature of the scraped projects is highly varied, a basic description of the projects' purpose is in many case necessary to avoid the complete analysis of the code. The second rule was added to filter out overly complex or unorganized projects, especially if they are composed of many different subprojects; the commonly occuring auxiliary subfolders were not counted in this filtering, since they do not contribute to the project's complexity, and do not represent independent subprojects.

1. The inspected project must work as-is; this means, no compilation or execution errors are accepted, and the only necessary modifications allowed are to configuration files, environment files (such as API keys, credentials and certificates) and the `serverless.yml` file (as part of the migration from different platforms). No significant changes have to be made to the running code, except for migration-related issues (such as parsing function parameters).

   The reason for this rule is to limit the amount of time spent bugfixing and inspecting the projects' code; many of the scraped repositories are not maintained, and as such might present a high number of compatibility issues and general bugs; as a complete debugging and renewal of all projects is beyond the scope of this thesis, only working samples were used.

2. The inspected project must not use any paid service, or service connected with paid subscriptions; these services include all AWS and Google Cloud features, such as storage on S3 or deployment dependent on GCF.

   This rule was added to guarantee that the selected test cases were available for personal use, with no economic requirement.

3. The inspected project must not be a simple "Hello World"; example projects and simple boilerplate code are discarded, as they do not represent realistic

use cases. The project's code must implement a function accepting input and producing an output as a result of either an internal transformation (such as code formatting or the calculation of a complex mathematical expression), or the interaction with an external service.

The reason for this rule is to simply filter out all projects which do not represent actual use cases, and are only examples or serverless functions.

4. The inspected project's `README.md` file must be in English, and must contain at least a simple description of the project's purpose.

   This rule is complementary to the first one, and the filtering based on it was performed manually; similarly to other rules, its purpose is to avoid the complete analysis of all projects, limiting the selection to repositories where the purpose of the code was made clear; the English language alone was chosen for consistency with the language used in this work.

Use cases

The three following projects passed the previously described selection; they were taken from the analysed folders, and translated for each of the deployed platforms. No modifications were done to the actual running code, save for changes when parsing the functions' parameters (which are handled differently in the various frameworks).

- **bespinian/k8s-faas-comparison** is a simple project, written for different platforms; it contains instructions for deployment and a function consisting of a POST to the Slack API, sending a message. While not very complex, it is a very common example of a FaaS application, acting as the endpoint for a Slack Bot.
- **hellt/pycatj-web** is a Python-based function, requiring pre-packaged code to work; it consists of a formatter, which takes incoming JSON as a string

and returns it as a series of equivalences. The repository contains both the web version and the serverless version of the service; only the latter was modified and deployed. As a sporadically invoked web-based function, it represents a frequent use case for serverless applications.

- **terraindata/terrain** is a complex repository, containing a serverless application as a stress-test for a deployed backend; this backend was deployed on another machine, separated from the main cluster, and was used as the target for this stress-test. The performance evaluation in this thesis doesn't contain the stress-test results, as the backend's performance is not part of the deployed frameworks. As with the two previous applications, this is a common example of a FaaS use case (monitoring and benchmarking external systems).

4.1.3 Critical cases

The following test cases are designed to target a specific "issue" of both the single platforms and the serverless paradigm as a whole; they are not designed to be optimized or realistic use cases, but rather to create suboptimal conditions for the platforms to perform.

- The **cold start** test case consists of a simple *Hello World* function, returning a predetermined string; no request parameter is analyzed or parsed. The function has relatively heavy dependencies (42.8 MB of libraries), which are all required and instantiated when the function starts. This type of function is deliberately against the usual guidelines, and aims to showcase how the various frameworks handle "heavy" functions in term of cold starts.
- The **scale-to-zero** test case uses the same setup as the cold start; with both this and the previous case, the invocation pattern is an integral part of the test, as the function is called every 5 minutes for a defined number of times,

and the resource usage in terms of memory and CPU in each of the cluster nodes is the object of the measurement. This test case will show how the different frameworks handle “idling" functions (i.e. if they are able to avoid wasting computational resources when a function is not being invoked).

- The **data locality** test case is both a memory-heavy function and a data querying function; it requests a large document (124.38MB) from the same database as the `mongoDB` test case, and extracts a property from the returned JSON. In this case, the performance impact of querying the document from a nearby node should be significant.

4.1.4 Test execution with JMeter

As all of the deployed actions were easily accessible as web endpoints, Apache JMeter[35] was chosen as the load testing and benchmarking tool; this open source project was originally designed to test web applications, and has subsequently expanded to other functions.

The way JMeter works is by sending a certain amount of requests towards a service, simulating the presence of multiple users with the use of threads; JMeter can also be used in a distributed fashion, with a main node coordinating the testing and various worker nodes actually sending the requests. JMeter tests are defined in XML files containing the *test plans*, which are easily editable using the GUI.

For the purpose of this thesis, separate test plans were defined for each platform and each group of test cases; also, all tests were executed in headless mode to minimize memory consumption by the GUI, while metrics were automatically collected by the tool; as the `scale-to-zero` test case also needed measurements regarding the cluster nodes activity (and not only the invocation performance), the Kubernetes default metrics API[1] was used to sample CPU and memory usage.

[1]`https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/`

Configuration

For each category of test cases, different JMeter configurations were used; each of these configurations included multiple Thread Groups, executed consecutively (one at a time), with differing numbers of repeated requests and parallel users.

For **basic** actions, the default amount of parallel threads (users) was 4, with a 10 seconds ramp-up time (the time needed to reach the total number of threads) and 200 requests per user; the exception was the `sleep` test case, which was reduced to 25 requests per user (a larger sample size was not needed, as the function had a very predictable behaviour).

For the **Wonderless** use cases, different setups were used for each one:

- `terrain` had 1 parallel user, 5 repetitions and a 20 seconds pause between each repetition; as the task was already a stress test, the amount of parallel computation needed on the node was already high.
- `slackpost` had 1 parallel user, 100 repetitions and a 1 second pause between each request, to accomodate Slack API's rate limits.
- `pycatj` had 4 parallel users, 200 repetitions and a 10 seconds ramp-up time, similarly to *basic* test cases; as it was a very light function, more parallel requests could be easily handled by the platforms.

Finally, for the **critical** test cases the following setups were used:

- The `dataLocality` function was invoked with 4 parallel threads and 50 repetitions, with a 10 seconds ramp-up time
- The `coldStart` and `scale-to-zero` cases (using the same function) were configured with 1 thread, 3 repetitions and pauses of 11 minutes between each repetition; this was done to accomodate OpenWhisk default idling tolerance

of 10 minutes, allowing the platform to properly deallocate the containers even with unmodified settings.

Metrics

In all test cases, the following metrics were considered:

- Median, average and maximum **latency** (time between just before sending the request and just after the first response has been received)
- Percent **error rate**
- Average **connection time** (time needed to establish the connection, including the SSL handshake)
- Average **throughput** (inverse of the average elapsed time, calculated similarly to the latency, but with the last response received)

Alongside these, the **CPU and Memory usage** were also considered when handling the `scale-to-zero` test case, and compared in three different stages: *Starting*, when the platform had been deployed and no functions were create; *Idle*, when functions were scaled down to zero (so functions were created, but hadn't been invoked in some time); and *Processing*, when the function was being invoked.

4.1.5 Cluster configuration

The various frameworks were all deployed on the same cluster, built from six different machines, in two different regions. Every machine in the cluster was hosted on Azure, since the hybrid nature of the cluster was not necessary for the performance analysis (the geographical and latency differences were enough for data locality tests, and the private/public duality of hybrid clusters was not part of the analysed characteristics). The cluster structure is shown in Table 4.1.

Additionally, two machines were deployed using the EC2 service on AWS, both of them in the same region; these were used respectively for hosting the mongoDB database and the Terrain backend used in the test cases. Neither of the two machines was part of the main cluster, and both of them were meant to be used as examples of external services; as the two nodes' hostnames are not related to their purpose, they are omitted from Table 4.2.

After the cluster deployment, the latency between its nodes and the machines hosting the external services was measured; as expected, nodes in the same geographical region had a much lower latency; the results of these measurements can be seen in Table 4.3.

The intra-cluster latency was also observed, with a ping of around 82 between nodes in different regions and around 1 between nodes in the same region; this shows that the same-region Virtual Machines were probably deployed in the same LAN, if not on the same physical host. Intra-cluster latency has to be considered among the issues with multi-zone clusters, as communication between nodes can become a significant bottleneck. Additionally, the latency from the JMeter host to the *AzureMaster* endpoint was measured, with an average ping of 28.1ms and a maximum ping of 28.4ms.

4.2 Results

The following sections contain the results in the various test cases for all platforms; since the modified version of OpenWhisk implemented in this thesis allowed for the use of different policies and levels of tolerance (in case of tagged functions), only the best result in terms of average latency was reported in the tables; all the other results were reported in the last section (Section 4.2.4) separately, divided by

Hostname	CPU Cores	CPU Clock Speed	Memory	Region
AzureMaster	2	2.30GHz	4GB	EU Central
AzureWorker	2	2.30GHz	4GB	EU Central
AzureWorker1	1	2.60GHz	3.5GB	EU Central
AzureWorker2	2	2.30GHz	4GB	US East
AzureWorker3	1	2.60GHz	3.5GB	US East
AzureWorker4	1	2.60GHz	3.5GB	US East

Table 4.1: Cluster configuration; the *AzureWorker* and *AzureWorker2* machines were used as Controllers in OpenWhisk deployment, while *AzureWorker1*, *AzureWorker3* and *AzureWorker4* were used as Invokers. *AzureMaster* was not used for scheduling by any of the frameworks (since it acts as Kubernetes master node, and the various framework's pods are not deployed on it).

Role	CPU Cores	CPU Clock Speed	Memory	Region
mongoDB host	1	2.40GHz	1GB	US East
Terrain host	2	2.40GHz	4GB	US East

Table 4.2: Service nodes configuration; hostnames are not shown, as they bore no connection with the nodes' purpose. Both nodes were purposely placed in the same region as two of the main cluster's nodes, to properly exploit the lower latency.

Target	Cluster Node	Avg. ping	Max. ping
mongoDB host	AzureMaster	80.702	83.316
	AzureWorker	80.73	81.29
	AzureWorker1	80.757	81.276
	AzureWorker2	2.04	2.16
	AzureWorker3	2.093	2.354
	AzureWorker4	2.18	2.33
Terrain host	AzureMaster	80.708	83.561
	AzureWorker	80.61	82.4
	AzureWorker1	80.707	81.775
	AzureWorker2	2.05	2.6
	AzureWorker3	2.056	2.606
	AzureWorker4	2.4	2.73

Table 4.3: Latency between cluster nodes and services; as expected, nodes located in US East have a much lower latency compared to the ones in EU, being in the same geographical region as the services.

Invoker distribution policy; additionally, the parameter *CTag* was used to indicate whether the Controller tag was sent to Nginx during the invocation (instructing the proxy to forward the request to the correct upstream), having value *True* when the tag was included in the request, and *False* otherwise.

Two different situations were also considered for OpenFaaS, as the platform allows to specify label constraints in function deployments; as such, functions were deployed both on nodes in US East and EU Central, analyzing the difference in results for tests where locality was significant (i.e. `mongoDB`, `terrain` and `datalocality`). The results for functions deployed in US East are listed as *OpenFaaS (local node)* in the tables.

For the `coldstart` test case, both an analysis of the response times and of cpu/memory usage was given; in the plots, the performance for all policies of the modified OpenWhisk platform is shown, alongside all the other platforms' performance.

The test names used in the following tables mirror the ones used in the various test cases and action definitions, and are slightly different from the ones previously listed; all Wonderless test cases had the repository name removed (and as such became `pycatj`, `slackpost` and `terrain`), and the critical cases were renamed to `datalocality` and `coldstart` (since scale-to-zero and cold start both used the same test, and measured different parameters).

4.2.1 Basic test cases

hellojs

In the simplest test case, it's easy to see that both Fission and OpenFaaS outperform each version of OpenWhisk; this is because of the generally lighter nature of both platforms, as OpenWhisk has to duplicate many Kubernetes functionalities. It can also be seen from the *Latency (max)* field that Fission and OpenWhisk both experience cold starts, with OpenWhisk being slightly slower, while OpenFaaS (having no idling mechanism in the free version) constantly keeps a pod alive, and as such has an uninterrupted warm start for every invocation.

It's interesting to see that the modified version of OpenWhisk performs better compared to the standard one; this is probably due to the strategy used in the best-performing case (`shared`), which greedily uses both local and non-local Invokers, grabbing resources if necessary; as the function is particularly lightweight, this probably reduces waiting times, as Invokers don't get overloaded and Controllers can benefit from a non-conservative approach.

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	180.661 ms	5030 ms	114.0 ms
OpenFaaS	159.132 ms	390 ms	157.0 ms
OpenWhisk (standard)	1133.439 ms	6876 ms	1092.0 ms
OpenWhisk (modified)	724.469 ms	6738 ms	623.0 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	5.5325 req/s	0.00%	0.5775 ms
OpenFaaS	6.2806 req/s	0.00%	0.2437 ms
OpenWhisk (standard)	0.8822 req/s	0.00%	5.5575 ms
OpenWhisk (modified)	1.3802 req/s	0.00%	4.6025 ms

Table 4.4: Results on `hellojs` test case.

matrixMult

While heavier from a computational standpoint, this function still shows very similar results to the previous case; the only interesting note is that the best performing modified version of OpenWhisk obtains slightly worse results compared to the standard one; this can be seen as a consequence of the computational overhead introduced by the modifications; yet again the greedy approach of the `shared` distribution performs better than the others, with the `default` approach performing very poorly (as seen in Table 4.14), showing the necessity for further optimizations.

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	551.179 ms	8116 ms	633.0 ms
OpenFaaS	344.654 ms	770 ms	306.0 ms
OpenWhisk (standard)	1323.608 ms	6796 ms	1253.5 ms
OpenWhisk (modified)	1452.904 ms	4627 ms	1385.5 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	1.3525 req/s	0.00%	1.6675 ms
OpenFaaS	2.3864 req/s	0.00%	0.4363 ms
OpenWhisk (standard)	0.6513 req/s	0.00%	6.8712 ms
OpenWhisk (modified)	0.6007 req/s	0.00%	7.3187 ms

Table 4.5: Results on `matrixMult` test case.

sleep

As this function has a fixed duration, not much can be seen in terms of execution and instantiation performance (no additional library is used, and the only parameter is the number of milliseconds to wait); all platforms have an average latency of around 3 seconds (the waiting time). The interesting parameter is, as in the `hellojs` case, the maximum latency, which again shows the advantage of the OpenFaaS approach in terms of cold starts; anyway, the median and average latency both show that these differences are easily erased with the subsequent invocations.

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	3321.720 ms	8164 ms	3120.0 ms
OpenFaaS	3215.290 ms	3407 ms	3205.5 ms
OpenWhisk (standard)	3775.190 ms	7799 ms	3632.0 ms
OpenWhisk (modified)	3554.280 ms	5281 ms	3595.0 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	0.3010 req/s	0.00%	8.9300 ms
OpenFaaS	0.3110 req/s	0.00%	3.5300 ms
OpenWhisk (standard)	0.2649 req/s	0.00%	22.7900 ms
OpenWhisk (modified)	0.2813 req/s	0.00%	22.7200 ms

Table 4.6: Results on `sleep` test case.

mongoDB

As expected, in this case the latency between the cluster and the target service (the mongoDB host node) played a significant role; both for OpenWhisk and OpenFaaS (with the modified version and the node label constraint respectively), the consideration for locality provided a performance boost; again, the `shared` Invoker distribution was the best performing one, with a tolerance of `none` (only the Invokers in the desired topology zone were used), and the passing of the Controller tag to Nginx.

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	184.509 ms	2535 ms	128.0 ms
OpenFaaS	624.819 ms	794 ms	621.0 ms
OpenFaaS (local node)	141.833 ms	404 ms	140.0 ms
OpenWhisk (standard)	1053.322 ms	4260 ms	1071.5 ms
OpenWhisk (modified)	793.616 ms	11099 ms	730.0 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	5.4188 req/s	0.00%	0.5500 ms
OpenFaaS	1.6004 req/s	0.00%	0.6450 ms
OpenFaaS (local node)	7.0495 req/s	0.00%	0.2150 ms
OpenWhisk (standard)	0.9494 req/s	0.00%	5.1063 ms
OpenWhisk (modified)	1.2599 req/s	0.00%	4.4575 ms

Table 4.7: Results on `mongoDB` test case.

4.2.2 Wonderless use cases

pycatj

A very lightweight function, the observed behaviour is similar to the basic use cases; it's interesting to see that the modified version of OpenWhisk, while still suffering from heavy cold starts (more than four times both Fission and OpenFaaS), performs better than the standard version; as the best performing policy is `min_memory`, with `shared` being a close second (as seen in Table 4.17), the Invoker distribution probably plays a role by allowing resource grabbing and reusing Invokers in a greedy way (while with the default distribution Invokers may be discarded in scheduling when they're seen as full).

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	223.681 ms	1922 ms	113.0 ms
OpenFaaS	574.400 ms	1664 ms	544.0 ms
OpenWhisk (standard)	830.872 ms	4193 ms	835.0 ms
OpenWhisk (modified)	605.663 ms	8017 ms	600.0 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	4.4697 req/s	0.00%	0.5613 ms
OpenFaaS	1.7408 req/s	0.00%	0.4263 ms
OpenWhisk (standard)	1.2035 req/s	0.00%	4.0438 ms
OpenWhisk (modified)	1.6510 req/s	0.00%	3.4100 ms

Table 4.8: Results on `pycatj` test case.

slackpost

Similarly to previous cases, Fission and OpenFaaS generally outperform OpenWhisk, especially in terms of maximum latency; with the `shared` distribution policy the modified version of OpenWhisk manages to obtain a better performance, reaching a lower average latency then OpenFaaS, albeit with a very significant cold start.

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	824.150 ms	2580 ms	882.5 ms
OpenFaaS	1232.550 ms	2429 ms	1193.0 ms
OpenWhisk (standard)	1542.950 ms	5466 ms	1441.5 ms
OpenWhisk (modified)	1101.100 ms	7363 ms	1431.5 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	1.2129 req/s	0.00%	3.4800 ms
OpenFaaS	0.8112 req/s	0.00%	1.9700 ms
OpenWhisk (standard)	0.6480 req/s	0.00%	15.1100 ms
OpenWhisk (modified)	0.9081 req/s	0.00%	13.0900 ms

Table 4.9: Results on `slackpost` test case.

terrain

Another test where locality plays an important role, as a high number of parallel HTTP requests has to be forwarded to a remote service (the *Terrain host* node on AWS); the heavy nature of the action created issues with both versions of OpenWhisk, possibly incurring in bottlenecks due to the low specifications of the cluster, and resulting in various errors. Nevertheless, the impact of manually selecting the node can be easily observed in OpenFaaS, where the consideration of locality reduced the latency to less than half.

Given the high percentage of errors in a very small number of requests (only five invocations in this test case), the average latency results for OpenWhisk can be ignored, as they probably do not offer a significant representation of the platform's performance.

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	6074.200 ms	8201 ms	5537.0 ms
OpenFaaS	5641.800 ms	6199 ms	5379.0 ms
OpenFaaS (local node)	2471.200 ms	3172 ms	2323.0 ms
OpenWhisk (standard)	11932.000 ms	12906 ms	11932.0 ms
OpenWhisk (modified)	6363.750 ms	9465 ms	6465.0 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	0.1646 req/s	0.00%	44.0000 ms
OpenFaaS	0.1772 req/s	0.00%	43.0000 ms
OpenFaaS (local node)	0.4046 req/s	0.00%	41.8000 ms
OpenWhisk (standard)	0.0838 req/s	60.00%	168.0000 ms
OpenWhisk (modified)	0.1571 req/s	20.00%	147.5000 ms

Table 4.10: Results on `terrain` test case.

4.2.3 Critical cases

coldstart/scale-to-zero

This test was, as mentioned, divided in two parts; the *cold start* was evaluated with the same metrics as the other test cases (Table 4.1), while the *scale-to-zero* used memory and CPU metrics, to observe how the different frameworks handled the pod/container deprovisioning, and their resource consumption in general (Figure 4.1).

Expectedly, OpenFaaS outperformed all platforms both in maximum and average latency, as its strategy in the free version is to keep one replica of each function always active and ready for requests; as such, the structure of the test (very small number of invocations, very long interval) heavily favored this approach. Fission demonstrates a rather good performance in terms of starting times, with the standard version of OpenWhisk obtaining slightly worse results. The modified version of OpenWhisk introduced in this thesis shows a worse performance, consistently with the other test cases in terms of cold starts; the best performing policy was `min_memory`, with the other three obtaining similar results to each other (Table 4.20).

Regarding resource usage, OpenWhisk shows a significantly higher load both in terms of memory and CPU; the modified version is heavier when it comes to memory (as expected from the addition of the watcher service and the NJS script, both described in Section 3.1), and a comparable load in terms of CPU. No clear peaks are present in the CPU plot, indicating the relatively low impact the function invocation has compared to the other services; the memory usage on the other hand shows rather regular increases following the invocations, and subsequent decreases after the containers are removed.

On the other hand, both Fission and OpenFaaS show clear peaks in terms of CPU usage, with an extremely low idle load; both platforms essentially only performed computations after the invocations had been received. From a memory standpoint, OpenFaaS requires a slightly lower amount, but shows no decrease in memory usage during the entire test, as expected from the absence of an idling handler.

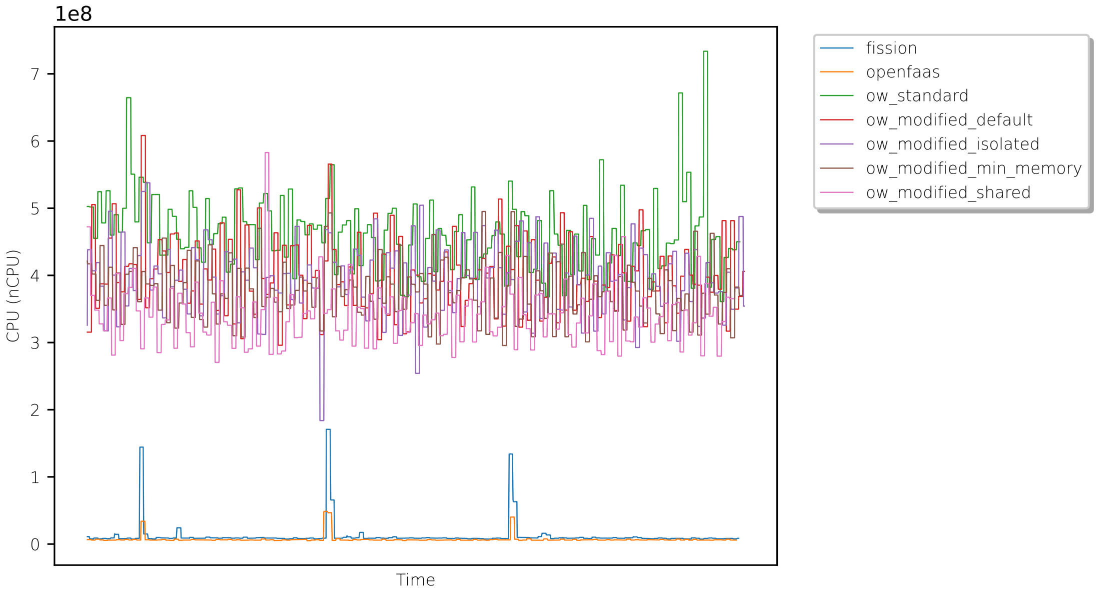
In Figure 4.1, the CPU usage is in $nCPU$[2] and the Memory usage is in *KiB*. The resource usage was measured using Kubernetes metrics API, filtering the pods by namespace.

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	5058.000 ms	5494 ms	5337.0 ms
OpenFaaS	1878.000 ms	3949 ms	843.0 ms
OpenWhisk (standard)	6758.333 ms	8002 ms	6544.0 ms
OpenWhisk (modified)	8612.333 ms	11210 ms	8204.0 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	0.1977 req/s	0.00%	38.3333 ms
OpenFaaS	0.5321 req/s	0.00%	117.3333 ms
OpenWhisk (standard)	0.1480 req/s	0.00%	457.3333 ms
OpenWhisk (modified)	0.1161 req/s	0.00%	412.3333 ms

Table 4.11: Results on `coldstart/scale-to-zero` test case.

[2]`https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/#cpu-units`



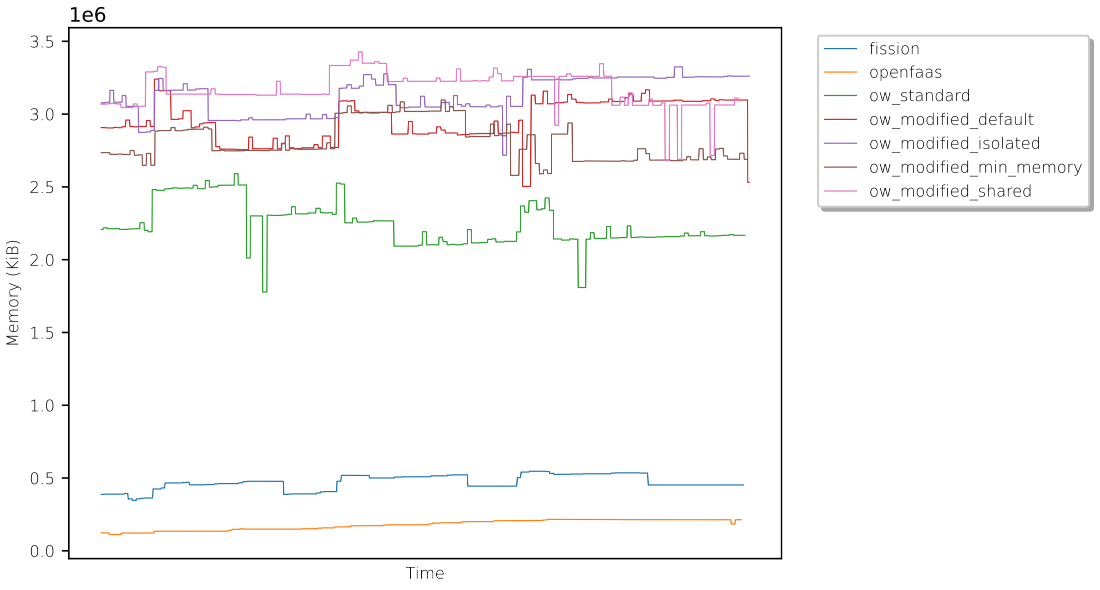


Figure 4.1: CPU and Memory usage for the `coldstart`/`scale-to-zero` test case.

datalocality

For this test case, both the ability to handle heavier workloads in terms of memory and the consideration for locality were important factors.

Fission achieves a very low average latency, with a rather high maximum latency, yet again showing very good results, possibly influenced by the use of "correct" nodes. OpenFaaS as usual has very close numbers in terms of its average and maximum latency, thanks to the virtually absent cold starts; as with previous locality-based tests, the node specification introduces a massive increase in performance. The standard version of OpenWhisk performs better than the non-local version of OpenFaas, but still achieves a rather poor performance compared to all other options; on the other hand, the modified version (using the `default` distribution policy, with a tolerance of `all` for non-local nodes) is almost even with both Fission and the node-constrained version of the OpenFaaS function.

Platform	Latency (avg)	Latency (max)	Latency (median)
Fission	2046.040 ms	16278 ms	1077.0 ms
OpenFaaS	14873.880 ms	15625 ms	14849.0 ms
OpenFaaS (local node)	2223.505 ms	3101 ms	2283.5 ms
OpenWhisk (standard)	9869.705 ms	23191 ms	15190.0 ms
OpenWhisk (modified)	2817.495 ms	13663 ms	2581.0 ms

Platform	Throughput	Error rate	Conn. time (avg)
Fission	0.4887 req/s	0.00%	5.0450 ms
OpenFaaS	0.0672 req/s	0.00%	22.7650 ms
OpenFaaS (local node)	0.4497 req/s	0.00%	4.4000 ms
OpenWhisk (standard)	0.1013 req/s	0.00%	12.8900 ms
OpenWhisk (modified)	0.3549 req/s	0.00%	16.8550 ms

Table 4.12: Results on `datalocality` test case.

4.2.4 Modified OpenWhisk: all results

In the following paragraphs, the results obtained by the modified OpenWhisk platform in all tests will be listed; unlike the previous section, the comparison will be between different Invoker distribution policies, along with the tolerance level and the presence/absence of the Controller tag for Nginx, both specified in data locality-based tests (i.e. `mongoDB`, `terrain`, `datalocality`).

hellojs

As previously mentioned, the best performance is achieved by the `shared` Invoker distribution policy; an interesting detail is that, while both the `min_memory` and the `shared` achieve better results in terms of average latency, they also achieve significantly worse results in terms of maximum latency; this might point to a property of resource grabbing between Controllers, which can allow a better overall performance, but can also create long waiting times due to load balancers requiring more resources than are actually available; nevertheless, considering the results in the other tests, no definitive conclusion can be drawn on this fact.

Policy	Latency (avg)	Latency (max)	Latency (median)
default	1600.190 ms	4616 ms	1541.0 ms
isolated	1346.665 ms	3088 ms	1310.0 ms
min_memory	741.325 ms	7194 ms	697.5 ms
shared	**724.469 ms**	**6738 ms**	**623.0 ms**

Policy	Throughput	Error rate	Conn. time (avg)
default	0.6249 req/s	0.00%	7.6500 ms
isolated	0.7425 req/s	0.00%	2.7363 ms
min_memory	1.3489 req/s	0.00%	4.2800 ms
shared	**1.3802 req/s**	**0.00%**	**4.6025 ms**

Table 4.13: Modified OpenWhisk results on `hellojs` test case.

matrixMult

Similarly to the previous case, the policies allowing for resource grabbing between Controllers achieve better results; unlike the previous case, the `shared` policy also obtains a lower maximum latency than the others.

Policy	Latency (avg)	Latency (max)	Latency (median)
default	2383.347 ms	5969 ms	2530.0 ms
isolated	1880.938 ms	7633 ms	1826.0 ms
min_memory	1583.634 ms	12389 ms	1458.0 ms
shared	**1452.904 ms**	**4627 ms**	**1385.5 ms**

Policy	Throughput	Error rate	Conn. time (avg)
default	0.3844 req/s	0.00%	10.7188 ms
isolated	0.5049 req/s	0.00%	2.4588 ms
min_memory	0.5552 req/s	0.00%	7.6063 ms
shared	**0.6007 req/s**	**0.00%**	**7.3187 ms**

Table 4.14: Modified OpenWhisk results on `matrixMult` test case.

sleep

One interesting result in this test case is the large difference in maximum latency between the `default` policy and the other three, which are relatively close in this regard; this might point to an issue with only allowing the Controller to require half of each Invoker's memory, as it might lead to them being marked as busy, causing the load balancer to schedule the invocation on a different node (causing a new cold start, as the container needs to be instantiated again); since only 4 parallel threads were used, the other approaches probably allowed for more repeated invocations on the same node.

Policy	Latency (avg)	Latency (max)	Latency (median)
default	4224.300 ms	8825 ms	4147.0 ms
isolated	4078.310 ms	5578 ms	4077.5 ms
min_memory	3565.670 ms	4960 ms	3604.5 ms
shared	**3554.280 ms**	**5281 ms**	**3595.0 ms**

Policy	Throughput	Error rate	Conn. time (avg)
default	0.2367 req/s	0.00%	22.7200 ms
isolated	0.2452 req/s	0.00%	6.6500 ms
min_memory	0.2805 req/s	0.00%	22.6600 ms
shared	**0.2813 req/s**	**0.00%**	**22.7200 ms**

Table 4.15: Modified OpenWhisk results on `sleep` test case.

mongoDB

As with previous tests, the `min_memory` and `shared` policies achieve better results in terms of average latency, while showing comparable or worse results in maximum latency; the different tolerance levels show a relatively low impact in all runs, while the use of the Controller tag to influence Nginx generally produces better results, especially with the `isolated` policy.

Policy	Tol	CTag	Latency (avg)	Latency (max)	Latency (median)
default	all	True	1135.152 ms	7472 ms	1059.5 ms
	all	False	1548.334 ms	8810 ms	1646.5 ms
	none	True	1053.924 ms	6912 ms	992.0 ms
	none	False	1563.419 ms	17754 ms	1595.5 ms
	same	True	1232.984 ms	5892 ms	1158.5 ms
	same	False	1645.590 ms	14854 ms	1651.0 ms
isolated	all	True	1099.134 ms	13290 ms	1026.0 ms
	all	False	1786.374 ms	14942 ms	1702.0 ms
	none	True	1091.286 ms	10715 ms	1036.0 ms
	none	False	1754.541 ms	14876 ms	1561.5 ms
	same	True	1363.170 ms	15828 ms	1119.5 ms
	same	False	1743.003 ms	15463 ms	1712.5 ms
min_memory	all	True	826.844 ms	14293 ms	731.0 ms
	all	False	844.663 ms	10928 ms	807.0 ms
	none	True	850.220 ms	14267 ms	771.5 ms
	none	False	902.420 ms	12257 ms	814.5 ms
	same	True	1069.070 ms	20285 ms	955.5 ms
	same	False	958.774 ms	14559 ms	915.5 ms
shared	all	True	832.296 ms	15067 ms	765.5 ms
	all	False	898.861 ms	15990 ms	811.0 ms
	none	**True**	**793.616 ms**	**11099 ms**	**730.0 ms**
	none	False	884.416 ms	15392 ms	805.0 ms
	same	True	1097.438 ms	20713 ms	994.0 ms
	same	False	949.717 ms	19749 ms	821.0 ms

Policy	Tol	CTag	Throughput	Error rate	Conn. time (avg)
default	all	True	0.8809 req/s	0.00%	5.4325 ms
	all	False	0.6458 req/s	0.00%	7.0650 ms
	none	True	0.9488 req/s	0.00%	5.4037 ms
	none	False	0.6396 req/s	0.00%	7.0975 ms
	same	True	0.8110 req/s	0.00%	6.2250 ms
	same	False	0.6077 req/s	0.00%	7.3450 ms
isolated	all	True	0.9098 req/s	0.00%	1.8462 ms
	all	False	0.5598 req/s	0.00%	2.1587 ms
	none	True	0.9163 req/s	0.00%	1.7737 ms
	none	False	0.5699 req/s	0.00%	2.1575 ms
	same	True	0.7336 req/s	0.00%	2.1113 ms
	same	False	0.5737 req/s	0.00%	2.1587 ms
min_memory	all	True	1.2093 req/s	0.00%	4.5400 ms
	all	False	1.1839 req/s	0.00%	4.1225 ms
	none	True	1.1761 req/s	0.00%	4.3650 ms
	none	False	1.1081 req/s	0.00%	4.3575 ms
	same	True	0.9354 req/s	0.00%	5.0912 ms
	same	False	1.0430 req/s	0.00%	5.1063 ms
shared	all	True	1.2014 req/s	0.00%	4.1562 ms
	all	False	1.1125 req/s	0.00%	4.4550 ms
	none	**True**	**1.2599 req/s**	**0.00%**	**4.4575 ms**
	none	False	1.1306 req/s	0.00%	4.1513 ms
	same	True	0.9112 req/s	0.00%	5.0862 ms
	same	False	1.0529 req/s	0.00%	5.0963 ms

Table 4.16: Modified OpenWhisk results on `mongoDB` test case.

pycatj

As already stated in the framework comparison section, this test case is very similar in structure and load to the basic ones; as such, it's not surprising that the `min_memory` and `shared` polcies achieve better results compared to the other two.

Policy	Latency (avg)	Latency (max)	Latency (median)
default	1324.940 ms	10175 ms	1270.0 ms
isolated	1569.459 ms	10762 ms	1481.0 ms
min_memory	**605.663 ms**	**8017 ms**	**600.0 ms**
shared	619.634 ms	10728 ms	591.5 ms

Policy	Throughput	Error rate	Conn. time (avg)
default	0.7547 req/s	0.00%	6.2438 ms
isolated	0.6371 req/s	0.00%	2.4337 ms
min_memory	**1.6510 req/s**	**0.00%**	**3.4100 ms**
shared	1.6137 req/s	0.00%	3.9588 ms

Table 4.17: Modified OpenWhisk results on `pycatj` test case.

slackpost

Similarly to the previous case, the `shared` and `min_memory` produce better results in terms of average latency, although like with the `hellojs` case, they both have worse results in terms of maximum latency. Analyzing the difference between the median and the average latency, along with the further inspection of the raw results, it was seen that the `default` and `isolated` policies produced very even results, with an almost constant latency, while the `shared` and `min_memory` policies had many extremely low-latency responses (around 500ms), alternated with some higher-latency ones (around 1.7-2.0k ms).

Policy	Latency (avg)	Latency (max)	Latency (median)
default	1691.860 ms	4650 ms	1646.0 ms
isolated	1492.500 ms	5320 ms	1450.5 ms
min_memory	1180.960 ms	7825 ms	1453.5 ms
shared	**1101.100 ms**	**7363 ms**	**1431.5 ms**

Policy	Throughput	Error rate	Conn. time (avg)
default	0.5910 req/s	0.00%	14.6100 ms
isolated	0.6700 req/s	0.00%	5.9400 ms
min_memory	0.8467 req/s	0.00%	12.4200 ms
shared	**0.9081 req/s**	**0.00%**	**13.0900 ms**

Table 4.18: Modified OpenWhisk results on `slackpost` test case.

terrain

Rather oddly, the addition of the Controller tag for Nginx resulted in more errors for this test case; an almost identical performance was achieved with all policies, both regarding the errors and the latency. The nature of these results (very similar to the ones obtained by the standard version of the platform) point to the need of further investigation of both the test case and the platform's behaviour with similar loads.

Policy	Tol	CTag	Latency (avg)	Latency (max)	Latency (median)
default	all	True	5337.500 ms	7173 ms	5337.5 ms
	all	False	8485.500 ms	11850 ms	8012.0 ms
	none	True	4882.500 ms	6591 ms	4882.5 ms
	none	False	7177.250 ms	12363 ms	6555.0 ms
	same	True	5191.500 ms	7241 ms	5191.5 ms
	same	False	7551.250 ms	11851 ms	7581.5 ms
isolated	all	True	6001.500 ms	9155 ms	6001.5 ms
	all	False	7460.750 ms	12135 ms	6432.5 ms
	none	True	5985.000 ms	8710 ms	5985.0 ms
	none	False	7444.500 ms	11703 ms	6718.5 ms
	same	True	5676.000 ms	8182 ms	5676.0 ms
	same	False	7699.250 ms	13948 ms	6431.0 ms
min_memory	all	True	6982.500 ms	11053 ms	6982.5 ms
	all	False	7178.750 ms	11407 ms	7170.0 ms
	none	True	6705.500 ms	9246 ms	6705.5 ms
	none	**False**	**6363.750 ms**	**9465 ms**	**6465.0 ms**
	same	True	5690.500 ms	8272 ms	5690.5 ms
	same	False	6977.500 ms	11162 ms	6518.0 ms
shared	all	True	7481.500 ms	11570 ms	7481.5 ms
	all	False	6957.500 ms	11500 ms	6672.5 ms
	none	True	6799.500 ms	10232 ms	6799.5 ms
	none	False	7136.000 ms	12269 ms	6556.5 ms
	same	True	6049.500 ms	8869 ms	6049.5 ms
	same	False	7270.750 ms	10860 ms	7697.5 ms

Policy	Tol	CTag	Throughput	Error rate	Conn. time (avg)
default	all	True	0.1873 req/s	60.00%	276.0000 ms
	all	False	0.1178 req/s	20.00%	145.0000 ms
	none	True	0.2048 req/s	60.00%	276.5000 ms
	none	False	0.1393 req/s	20.00%	142.5000 ms
	same	True	0.1926 req/s	60.00%	169.5000 ms
	same	False	0.1324 req/s	20.00%	141.5000 ms
isolated	all	True	0.1666 req/s	60.00%	178.0000 ms
	all	False	0.1340 req/s	20.00%	44.2500 ms
	none	True	0.1671 req/s	60.00%	149.0000 ms
	none	False	0.1343 req/s	20.00%	42.0000 ms
	same	True	0.1761 req/s	60.00%	168.0000 ms
	same	False	0.1299 req/s	20.00%	43.5000 ms
min_memory	all	True	0.1432 req/s	60.00%	299.5000 ms
	all	False	0.1393 req/s	20.00%	146.0000 ms
	none	True	0.1491 req/s	60.00%	269.0000 ms
	none	**False**	**0.1571 req/s**	**20.00%**	**147.5000 ms**
	same	True	0.1757 req/s	60.00%	170.0000 ms
	same	False	0.1433 req/s	20.00%	142.7500 ms
shared	all	True	0.1336 req/s	60.00%	283.5000 ms
	all	False	0.1437 req/s	20.00%	147.0000 ms
	none	True	0.1470 req/s	60.00%	310.5000 ms
	none	False	0.1401 req/s	20.00%	153.0000 ms
	same	True	0.1653 req/s	60.00%	301.0000 ms
	same	False	0.1375 req/s	20.00%	143.7500 ms

Table 4.19: Modified OpenWhisk results on `terrain` test case.

coldstart/scale-to-zero

As also seen in Table 4.11, the modified framework's performance greatly suffers in terms of cold starts; the best performing policy is still slower than the standard version, with the other three showing a significant decrease in performance. From the plots in Figure 4.1 it can be seen how the resource usage is almost identical for all policies, and slightly higher than the standard version in terms of memory.

Policy	Latency (avg)	Latency (max)	Latency (median)
default	11406.000 ms	13607 ms	13059.0 ms
isolated	11709.333 ms	13470 ms	11337.0 ms
min_memory	**8612.333 ms**	**11210 ms**	**8204.0 ms**
shared	12374.333 ms	13032 ms	13018.0 ms

Policy	Throughput	Error rate	Conn. time (avg)
default	0.0877 req/s	0.00%	422.0000 ms
isolated	0.0854 req/s	0.00%	171.3333 ms
min_memory	**0.1161 req/s**	**0.00%**	**412.3333 ms**
shared	0.0808 req/s	0.00%	412.0000 ms

Table 4.20: Modified OpenWhisk results on `coldstart` test case.

datalocality

Unlike many of the previous test cases, in this one both the `default` and the `isolated` policies obtain better performances than the other two, with the former achieving the best results in terms of both average and maximum latency. The addition of the Controller tag for Nginx also plays an important part, reducing the average latency of almost 8 seconds in some runs; this is probably due to the prioritization, by each Controller, of its local Invokers; as such, if the targeted Controller is the one that was requested, the invocation proceeds on the "correct" Invokers, according to the data locality requirements.

Policy	Tol	CTag	Latency (avg)	Latency (max)	Latency (median)
default	**all**	**True**	**2817.495 ms**	**13663 ms**	**2581.0 ms**
	all	False	10098.215 ms	24536 ms	14425.5 ms
	none	True	2821.820 ms	17130 ms	2531.0 ms
	none	False	10116.085 ms	24924 ms	13661.0 ms
	same	True	2872.480 ms	12181 ms	2633.0 ms
	same	False	10049.095 ms	22842 ms	15244.0 ms
isolated	all	True	3495.620 ms	17343 ms	3372.5 ms
	all	False	10058.985 ms	32000 ms	9900.0 ms
	none	True	3279.130 ms	15845 ms	3286.5 ms
	none	False	10028.000 ms	30027 ms	10529.5 ms
	same	True	3405.945 ms	16990 ms	3351.0 ms
	same	False	10006.395 ms	33245 ms	10285.0 ms
min_memory	all	True	4164.575 ms	19658 ms	3634.5 ms
	all	False	10050.355 ms	26861 ms	9928.5 ms
	none	True	4021.315 ms	20438 ms	3556.0 ms
	none	False	10075.885 ms	26557 ms	11360.0 ms
	same	True	3776.275 ms	15610 ms	3467.5 ms
	same	False	9959.625 ms	26570 ms	10312.5 ms
shared	all	True	3854.320 ms	17578 ms	3566.0 ms
	all	False	10122.625 ms	28337 ms	10243.5 ms
	none	True	3885.750 ms	16424 ms	3586.0 ms
	none	False	10229.940 ms	37892 ms	10122.5 ms
	same	True	3997.275 ms	19130 ms	3680.0 ms
	same	False	10321.500 ms	77023 ms	15061.0 ms

Policy	Tol	CTag	Throughput	Error rate	Conn. time (avg)
default	**all**	**True**	**0.3549 req/s**	**0.00%**	**16.8550 ms**
	all	False	0.0990 req/s	0.00%	38.4350 ms
	none	True	0.3544 req/s	0.00%	17.0050 ms
	none	False	0.0989 req/s	0.00%	38.6000 ms
	same	True	0.3481 req/s	0.00%	17.4100 ms
	same	False	0.0995 req/s	0.00%	38.5150 ms
isolated	all	True	0.2861 req/s	0.00%	5.8200 ms
	all	False	0.0994 req/s	0.00%	11.3450 ms
	none	True	0.3050 req/s	0.00%	5.8800 ms
	none	False	0.0997 req/s	0.00%	10.8500 ms
	same	True	0.2936 req/s	0.00%	5.9450 ms
	same	False	0.0999 req/s	0.00%	11.3150 ms
min_memory	all	True	0.2401 req/s	0.00%	22.2950 ms
	all	False	0.0995 req/s	0.00%	37.9250 ms
	none	True	0.2487 req/s	0.00%	22.0550 ms
	none	False	0.0992 req/s	0.00%	39.2250 ms
	same	True	0.2648 req/s	0.00%	22.9800 ms
	same	False	0.1004 req/s	0.00%	39.3550 ms
shared	all	True	0.2594 req/s	0.00%	22.8850 ms
	all	False	0.0988 req/s	0.00%	39.8100 ms
	none	True	0.2573 req/s	0.00%	22.7250 ms
	none	False	0.0978 req/s	0.00%	41.8850 ms
	same	True	0.2502 req/s	0.00%	21.2850 ms
	same	False	0.0969 req/s	0.00%	38.8000 ms

Table 4.21: Modified OpenWhisk results on `datalocality` test case.

Chapter 5

Conclusions

The OpenWhisk modifications introduced in this thesis proved effective in achieving a better performance for data locality-dependent applications, along with better results in some general use cases; the possibility to target multiple Invokers by only specifying their “assigned” Controller also allowed for more flexibility compared to the original extension defined in [6]. Although some of the results are promising, this modified version showed issues such as more severe cold starts in some cases, and a generally higher memory usage for an already memory-heavy framework.

The comparison with other platforms showed that, while the contributions in this thesis allowed OpenWhisk to obtain better results in some cases, the base framework is still noticeably slower than the other two; furthermore, while OpenFaaS demonstrated very good results in most test cases, Fission reached a comparable level of performance, without the potential issue of having a paid version (which inherently limits customization and tuning options).

Finally, both with OpenFaaS and OpenWhisk the importance of data locality-aware scheduling, be it via topology information (OpenWhisk) or simple node constraints (OpenFaaS), was found to be paramount for the frameworks' performance; because of this, giving users the ability to somewhat influence function scheduling according to their needs can be a good way to reduce overall latency in invocations, with relatively few implementative challenges.

Future work

The work discussed in the thesis can be improved from various points of view: the test results showed that, while some configurations of the modified versions achieved significantly better performances compared to the standard one, they also introduced memory consumption overheads and harsher cold starts in many invocations. This can be further optimized to align the modified framework's performance with its standard version when using the default configuration, effectively removing any additional overhead and only providing additional flexibility.

From another perspective, the tests performed in this thesis showed that, when working on small cluster and using Kubernetes as an orchestrator, both OpenFaaS and Fission greatly outperform OpenWhisk in almost all situations; it might be interesting to verify if this difference in performance persists with large-scale use cases, with thousands of functions deployed and better performing nodes. Additionally, the configuration language used both in this work and in [6] was designed specifically for OpenWhisk, and as much might not be very easily adaptable to different frameworks; extending the language to provide better adaptability to other platforms can facilitate possible modifications and improvements, while also allowing better testing of said modifications' effectiveness.

Defining a more abstract and general specification for scheduling manipulation can also allow the introduction of more complex policies, such as machine learning-based approaches (tuning the framework's behaviour on a per-user or per-function level); custom scheduling strategies might also be introduced and applied to multiple platforms, allowing programmers to easily transition from one to another according to their specific use cases (e.g. different orchestration systems, preferred languages or cluster configuration).

Bibliography

[1] Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. “Package-Aware Scheduling of FaaS Functions”. In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering.* ICPE ’18. Berlin, Germany: Association for Computing Machinery, 2018, pp. 101–106.

[2] Alexandru Agache et al. “Firecracker: Lightweight Virtualization for Serverless Applications”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20).* Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434.

[3] Gabriel Aumala et al. “Beyond Load Balancing: Package-Aware Scheduling for Serverless Platforms”. In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).* 2019, pp. 282–291.

[4] Ali Banaei and Mohsen Sharifi. “ETAS: predictive scheduling of functions on worker nodes of Apache OpenWhisk platform”. In: *The Journal of Supercomputing* (Sept. 2021).

[5] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator.” In: *USENIX Annual Technical Conference, FREENIX Track.* USENIX, May 7, 2007, pp. 41–46.

[6] Giuseppe De Palma et al. “Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation”. In: *ICSOC 2020 - 18 th International Conference on Service-Oriented Computing.* Dubai, United Arab Emirates, Dec. 2020, pp. 416–430.

[7] Simon Eismann et al. *A Review of Serverless Use Cases and their Characteristics*. 2021. arXiv: `2008.11110 [cs.SE]`.

[8] Nafise Eskandani and Guido Salvaneschi. "The Wonderless Dataset for Serverless Computing". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 2021, pp. 565–569.

[9] Maurizio Gabbrielli et al. "No More, No Less - A Formal Model for Serverless Computing". In: *COORDINATION 2019 - 21th International Conference on Coordination Languages and Models*. Ed. by Hanne Riis Nielson and Emilio Tuosto. Vol. LNCS-11533. Coordination Models and Languages. Part 3: Exploring New Frontiers. Kongens Lyngby, Denmark: Springer International Publishing, June 2019, pp. 148–157.

[10] Alex Glikson, Stefan Nastic, and Schahram Dustdar. "Deviceless Edge Computing: Extending Serverless Computing to the Edge of the Network". In: *Proceedings of the 10th ACM International Systems and Storage Conference*. SYSTOR '17. Haifa, Israel: Association for Computing Machinery, 2017.

[11] Marjan Gusev. "Serverless and Deviceless Dew Computing: Founding an Infrastructureless Computing". In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2021, pp. 1814–1818.

[12] Hassan Hassan, Saman Barakat, and Qusay Sarhan. "Survey on serverless computing". In: *Journal of Cloud Computing* 10 (July 2021).

[13] Joseph M. Hellerstein et al. *Serverless Computing: One Step Forward, Two Steps Back*. 2018. arXiv: `1812.03651 [cs.DC]`.

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

[15] Abhinav Jangda et al. "Formal Foundations of Serverless Computing". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019).

[16] Zhipeng Jia and Emmett Witchel. “Boki: Stateful Serverless Computing with Shared Logs”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP ’21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 691–707.

[17] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019.

[18] Swaroop Kotni et al. “Faastlane: Accelerating Function-as-a-Service Workflows”. In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 805–820.

[19] Jörn Kuhlenkamp, Sebastian Werner, and Stefan Tai. “The Ifs and Buts of Less is More: A Serverless Computing Reality Check”. In: *2020 IEEE International Conference on Cloud Engineering (IC2E)*. 2020, pp. 154–161.

[20] Filipe Manco et al. “My VM is Lighter (and Safer) than Your Container”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233.

[21] Peter M. Mell and Timothy Grance. *SP 800-145. The NIST Definition of Cloud Computing.* Tech. rep. Gaithersburg, MD, USA, 2011.

[22] Anup Mohan et al. “Agile Cold Starts for Scalable Serverless”. In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019.

[23] Johann Schleier-Smith et al. “What Serverless Computing is and Should Become: The next Phase of Cloud Computing”. In: *Commun. ACM* 64.5 (Apr. 2021), pp. 76–84.

[24] M. Sciabarrà. *Learning Apache OpenWhisk: Developing Open Serverless Solutions.* O’Reilly Media, 2019.

[25] Simon Shillaker and Peter Pietzuch. "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 419–433.

[26] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. "Prebaking Functions to Warm the Serverless Cold Start". In: *Proceedings of the 21st International Middleware Conference*. Middleware '20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 1–13.

[27] Eric Simmon. *Evaluation of Cloud Computing Services Based on NIST SP 800-145*. en. Feb. 2018.

[28] Khondokar Solaiman and Muhammad Abdullah Adnan. "WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing". In: *2020 IEEE International Conference on Cloud Engineering (IC2E)*. 2020, pp. 144–153.

[29] Vikram Sreekanti et al. "Cloudburst: Stateful Functions-as-a-Service". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2438–2452.

[30] Manuel Stein. *The Serverless Scheduling Problem and NOAH*. 2018. arXiv: `1809.06100 [cs.DC]`.

[31] Schahram Dustdar Thomas Rausch Alexander Rashed. "Optimized container scheduling for data-intensive serverless edge computing". In: *Future Generation Computer Systems* 114 (2021), pp. 259–271.

[32] Sebastian Werner, Richard Girke, and Jörn Kuhlenkamp. "An Evaluation of Serverless Data Processing Frameworks". In: WoSC'20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 19–24.

[33] Chenggang Wu et al. "Anna: A KVS for Any Scale". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 401–412.

Sitography

[34] *Apache CouchDB*. URL: https://couchdb.apache.org/.

[35] *Apache JMeter*. URL: https://jmeter.apache.org/.

[36] *Apache Kafka*. URL: https://kafka.apache.org/.

[37] *Apache OpenWhisk*. URL: https://openwhisk.apache.org/.

[38] *Docker*. URL: https://www.docker.com/.

[39] *faasd*. URL: https://github.com/openfaas/faasd/.

[40] *Fission*. URL: https://fission.io/.

[41] IBM Cloud Learn Hub. *What is FaaS?* URL: https://www.ibm.com/cloud/learn/faas.

[42] IBM Cloud Learn Hub. *What is Serverless Computing?* URL: https://www.ibm.com/cloud/learn/serverless.

[43] *Introducing AWS Lambda*. Nov. 2014. URL: https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/.

[44] *jq*. URL: https://stedolan.github.io/jq/.

[45] *Kata Containers*. URL: https://katacontainers.io/.

[46] *Koyeb*. URL: https://www.koyeb.com/.

[47] *Kubernetes*. URL: https://kubernetes.io/.

[48] *Linkerd*. URL: https://linkerd.io/.

[49] *NATS*. URL: https://nats.io/.

[50] *Nginx*. URL: https://nginx.org/.

[51] *Nomad*. URL: https://www.nomadproject.io/.

[52] *Open Container Initiative*. URL: https://opencontainers.org/.

[53] *OpenFaaS*. URL: https://www.openfaas.com/.

[54] *Prometheus*. URL: https://prometheus.io/.

[55] *Serverless Framework*. URL: `https://www.serverless.com/`.

[56] Alan Williams. *Tailor - the AWS Account Provisioning Service*. URL: `https://github.com/alanwill/aws-tailor`.