Allocation Priority Policies for Serverless Function-execution Scheduling Optimisation

Giuseppe De Palma[1], Saverio Giallorenzo[1,2, (former) 3], Jacopo Mauro[3], Gianluigi Zavattaro[1,2]

[1]Università di Bologna, IT [2]INRIA, FR [3]University of Southern Denmark, DK

Abstract. Serverless computing is a Cloud development paradigm where developers write and compose stateless functions, abstracting from their deployment and scaling. In this paper, we address the problem of *function-execution scheduling*, i.e., how to schedule the execution of Serverless functions to optimise their performance against some user-defined goals. We introduce a declarative language of *Allocation Priority Policies* (APP) to specify policies that inform the scheduling of function execution. We present a prototypical implementation of APP as an extension of Apache OpenWhisk and we validate it by *i*) implementing a use case combining IoT, Edge, and Cloud Computing and *ii*) by comparing its performance to an alternative implementation that uses vanilla OpenWhisk.

Keywords: Serverless · Function-execution Scheduling · Optimisation.

1 Introduction

Serverless computing [1], also known as Functions-as-a-Service, is a new development paradigm where programmers write and compose stateless functions, leaving to Serverless infrastructure providers the duty to manage their deployment and scaling. Hence, although a bit of a misnomer—as servers are of course involved—the "less" in Serverless refers to the removal of some server-related concerns, namely, their maintenance, scaling, and expenses deriving from their sub-optimal management (e.g., idle servers). Serverless computing was first proposed as a deployment modality for Cloud architectures [1] that pushed to the extreme the per-usage model of Cloud Computing, letting users pay only for the computing resources used at each function invocation. However, recent industrial and academic proposals, such as platforms to support Serverless development in Edge [2] and Internet-of-Things [3] scenarios, confirm the rising interest of neighbouring communities to adopt the Serverless paradigm.

While Serverless providers have become more and more common [4,5,6,7,8,9,10] the technology is still in its infancy and there is much work to do to overcome the many limitations [9,11,12,1] that hinder its wide adoption. One of the main challenges to address is how should Serverless providers schedule the functions on the available computation nodes. To visualise the problem, consider for example Fig. 1 depicting the availability of two **Worker**s—the computation nodes where functions can execute. One **Worker** is in Italy (**Site 1**) and the other in Greece

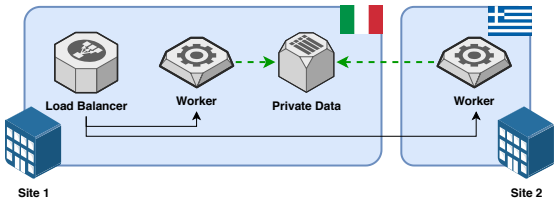

Fig. 1. Example of function-execution scheduling problem.

(**Site 2**). Both **Worker**s can execute a function that interacts (represented by the dashed green lines) with the **Private Data** storage located at **Site 1**. When the **Load Balancer** (acting as function scheduler) receives a request to execute the function, it must decide on which **Worker** to execute it. To minimise the response time, the **Load Balancer** should consider the different computational loads of the two **Workers**, which influence the time they take to execute the function. Also, the latency to access the **Private Data** storage plays an important role in determining the performance of function execution: the **Worker** at **Site 1** is close to the data storage and enjoys a faster interaction with it while the **Worker** at **Site 2** is farther from it and can undergo heavier latencies.

In this paper, we address the problem of *function-execution scheduling optimisation* [9] by proposing a methodology that provides developers with a declarative language, called *Allocation Priority Policies* (APP). Developers can use APP to specify a scheduling policy for their functions that the scheduler later uses to find the worker that, given the current status of the system, best fits the constraints specified by the developer of a given function. To substantiate our proposal, we extended the scheduler of OpenWhisk [5], a well-known open-source distributed Serverless platform, to use APP-defined policies in the scheduling of Serverless functions. In Section 3 we detail the APP language and present our prototypical implementation as an extension of Apache OpenWhisk [5]—in Section 2 we provide some introductory notions of the Serverless paradigm and an overview of the OpenWhisk platform. To validate our extension, in Section 4, we present a use case combining IoT, Edge, and Cloud Computing and we contrast an implementation of the use case using our APP-based prototype with a naïve one using three coexisting installations of the vanilla OpenWhisk stack to achieve the same functional requirements. We present the data on the performance of the two deployments, providing empirical evidence of the performance gains offered by the APP-governed scheduling. We conclude comparing with related work in Section 5 and discussing future and concluding remarks in Section 6.

2 Preliminaries

In this section, we give some preliminary information useful to better understand the motivations and technical details of our contribution. First, we outline the problems that motivate our research—as found in the literature. Then, we give an overview of the OpenWhisk Serverless platform, which we use in Section 3 to implement a prototype of our solution to the function scheduling problem.

Serverless Function Scheduling The Serverless development cycle is divided in two main parts: *a*) the writing of a function using a programming language supported by the platform (e.g. JavaScript, Python, C#) and *b*) the definition of an event that should trigger the execution of the function. For example, an event is a request to store some data, which triggers a process managing the selection, instantiation, scaling, deployment, fault tolerance, monitoring, and logging of the functions linked to that event. A Serverless provider—like IBM Cloud Functions [10] (using Apache OpenWhisk [5]), AWS Lambda [4], Google Cloud Functions [7] or Microsoft Azure Functions [6]—is responsible to schedule functions on its workers, to control the scaling of the infrastructure by adjusting their available resources, and to bill its users on a per-execution basis.

When instantiating a function, the provider has to create the appropriate execution environment for the function. Containers [13] and Virtual Machines [14] are the main technologies used to implement isolated execution environments for functions. How the provider implements the allocation of resources and the instantiation of execution environments impacts on the performance of the function execution. If the provider allocates a new container for every request, the initialisation overhead of the container would negatively affect both the performance of the single function and heavily increase the load on the worker. A solution to tackle this problem is to maintain a "warm" pool of already-allocated containers. This matter is usually referred to as *code locality* [9]. Resource allocation also includes I/O operations that need to be properly considered. For example, the authors of [15] report that a single function in the Amazon serverless platform can achieve on average 538Mbps network bandwidth, an order of magnitude slower than single modern hard drives (the authors report similar results from Google and Azure). Those performance result from bad allocations over I/O-bound devices, which can be reduced following the principle of *session locality* [9], i.e., taking advantage of already established user connections to workers. Another important aspect to consider to schedule functions, as underlined by the example in Fig. 1, is that of *data locality*, which comes into play when functions need to intensively access (connection- or payload-wise) some data storage (e.g., databases or message queues). Intuitively, a function that needs to access some data storage and that runs on a worker with high-latency access to that storage (e.g., due to physical distance or thin bandwidth) is more likely to undergo heavier latencies than if run on a worker "closer" to it. Data locality has been subject of research in neighbouring Cloud contexts [16,17].

Apache OpenWhisk Apache OpenWhisk [5] is an open-source Serverless platform initially developed by IBM—at the core of the company's Serverless offer [10]—and subsequently donated to the Apache Software Foundation. It is a production-ready Serverless platform and it supports the execution of functions written in many programming languages, including JavaScript, Python, Java, Go, and C#.

OpenWhisk is an event-driven system that runs code in response to events (e.g., changes to a database, an HTTP request or IoT sensors readings) or direct invocations. To pick up an event from a source, OpenWhisk defines a feed that activates triggers linked to a set of rules and actions to be executed.

OpenWhisk systems include one *controller* and a pool of *invokers*. The controller is a load balancer that, given an action to be executed, forwards the execution request to one selected invoker. The invokers execute actions using isolated Docker containers. Invokers are the OpenWhisk equivalent of the **Worker**s mentioned in our presentation. Latency-wise, container instantiation is by far the most relevant overhead endured by the invokers. One of the most effective mechanisms to reduce such overhead is to reuse containers, i.e., when a function is invoked multiple times, the system can reuse the container of a terminated invocation of that function rather than creating a fresh one.

The load balancing policy followed by the controller aims at maximising reuse. When the controller needs to schedule the execution of a function, a numeric hash h is calculated using the action name. An invoker is then selected using the remainder of the division between h and the total number of invokers n. The controller checks if the invoker is overloaded. If the chosen invoker is overloaded, the index is incremented by a step-size, which is any of the co-prime numbers smaller than the amount n of available invokers.

When no invoker is available after cycling through the entire invoker pool, the load balancer randomly selects an invoker from those that are considered "healthy"—able to sustain the workload. This happens when there are invokers that are healthy but have no capacity available when the scheduling algorithm was searching for an invoker. When there are no healthy invokers, the load balancer returns an error stating that no invokers are available for executing the function.

Motivation As discussed, at least three aspects related to function scheduling affect the performances of function execution in Serverless platforms: code, session, and data locality. Load balancing policies adopted by state-of-the-art Serverless platforms like Apache OpenWhisk take advantage only of code locality, but they currently have no way to integrate also information on other types of locality. To take advantage of other forms of locality, the load balancer should have knowledge on the way functions access external resources, like I/O-bound devices or databases, whose usage depends on the implementation of functions.

Our work aims at bridging that information gap, presenting a language that any Serverless platform can use in its scheduling policies to consider those factors. Our approach is conservative: with its default settings (explained in the next section) it can capture the status of current Serverless platforms. Then, more advanced Serverless users and platform providers can use the features offered by our proposal to optimise the execution of functions.

Moreover, optimised scheduling policies could be the outcome of automatic heuristic/inference systems applied to the functions to be executed. Automatic synthesis of optimized scheduling policies is the long-term objective of our research and this paper addresses the first fundamental step, i.e., showing the feasibility of Serverless platforms instructed with customized load balancing rules. Given this objective, we narrow the current exposition to manually-defined configurations and we leave the exploration of automatic configuration to future work.

3 The APP Language

Current serverless platforms, like OpenWhisk, come equipped with hard-coded load balancing policies. In this section, we present the *Allocation Priority Policies* (APP) language, intended as a language to specify customised load balancing policies and overcome the inflexibility of the hard-coded load balancing ones. The idea is that both developers and providers can write, besides the functions to be executed by the platform, a policy that instructs the platform what workers each function should be preferably executed on. Function-specific configurations are optional and without them the system can follow a default strategy.

As an extension of the example depicted in Fig. 1, consider some functions that need to access a database. To reduce latency (as per data locality principle), the best option would be to run those functions on the same pool of machines that run the database. If that option is not valid, then running those functions on workers in the proximity (e.g., in the same network domain) is preferable than using workers located further away (e.g., in other networks). We comment below an initial APP script that specifies the scheduling policies only for those workers belonging to the pool of machines running the database.

```
couchdb_query:
  - workers:
    - DB_worker1
    - DB_worker2
  strategy: random
  invalidate: ↩
    capacity_used: 50%
  followup: fail
```

At the first line, we define the *policy tag*, which is `couchdb_query`. As explained below, tags are used to link policies to functions. Then, the keyword `workers` indicates a list of *worker labels*, which identify the workers in the proximity of the database, i.e., `DB_worker1` and `DB_worker2`. As explained below, labels are used to identify workers. Finally, we define three parameters: the `strategy` used by the scheduler to choose among the listed worker labels, the policy that `invalidate`s the selection of a worker label, and the `followup` policy in case all workers are `invalidate`d. In the example, we select one of the two labels `random`ly, we `invalidate` their usage if the workers corresponding to the chosen label are used at more than the `50%` of their capacity (`capacity_used`) and, in case all workers are invalidated (`followup`), we let the request for function execution `fail`.

The APP syntax and semantics We report the syntax of APP in Fig. 2. The basic entities considered in the APP language are *a*) scheduling policies, identified by a *policy tag* identifier to which users can associate their functions—the policy-function association is a one-to-many relation—and *b*) workers, identified by a *worker label*—where a label identifies a collection of computation nodes. An APP script is a YAML [18] file specifying a sequence of policies. Given a tag, the corresponding policy includes a list of `workers` blocks, possibly closed with a `followup` strategy. A `workers` block includes three parameters: a collection of worker labels, a possible scheduling `strategy`, and an `invalidate` condition. A `followup` strategy can be either a `default` policy or the notification of `fail`ure.

We discuss the APP semantics, and the possible parameters, by commenting on a more elaborate script extending the previous one, shown in Fig. 3. The

$$
\begin{array}{lll}
\textit{policy\_tag} \in & \textit{Identifiers} \cup \{\texttt{default}\} \qquad \textit{worker\_label} \in \textit{Identifiers} \qquad n \in \mathbb{N} & \\
\textit{app} & ::= \overline{\textit{tag}} & \\
\textit{tag} & ::= \textit{policy\_tag} : \overline{\texttt{-}\ \textit{block}}\ \textit{followup}? & \\
\textit{block} & ::= \texttt{workers}\ [\ \texttt{"*"} \mid \overline{\texttt{-}\ \textit{worker\_label}}\ ] & \\
 & \quad (\texttt{strategy}\ [\ \texttt{random} \mid \texttt{platform} \mid \texttt{best\_first}\ ])? & \\
 & \quad (\texttt{invalidate}\ [\ \texttt{capacity\_used} : n\texttt{\%} \mid \texttt{max\_concurrent\_invocations} : n \mid \texttt{overload}\ ])? & \\
\textit{followup} & ::= \texttt{followup} : [\ \texttt{default} \mid \texttt{fail}\ ] &
\end{array}
$$

Fig. 2. The APP syntax.

APP script starts with the tag `default`, which is a special tag used to specify the policy for non-tagged functions, or to be adopted when a tagged policy has all its members invalidated, and the `followup` option is `default`.

In Fig. 3, the `default` tag describes the default behaviour of the serverless platform running APP. The wildcard `"*"` for the `workers` represent all worker labels. The `strategy` selected is the `platform` default (e.g., in our prototype in Section 4 the `platform` strategy corresponds to the selection algorithm described in Section 2) and its `invalidate` strategy considers a worker label non-usable when its workers are `overload`ed, i.e., none has enough resources to run the function.

Besides the `default` tag, the `couchdb_query` tag is used for those functions that access the database. The scheduler considers worker blocks in order of appearance from top to bottom. As mentioned above, in the first block (associated to `DB_worker1` and `DB_worker2`) the scheduler randomly picks one of the two worker labels and considers a label invalid when all corresponding workers reached the `50%` of capacity. Here the notion of capacity depends on the implementation (e.g., our OpenWhisk-based APP implementation in Section 4 uses information on the CPU usage to determine the load of invokers). When both worker labels are invalid, the scheduler goes to the next `workers` block, with `near_DB_worker1` and `near_DB_worker2`, chosen following a `best_first` strategy—where the scheduler considers the ordering of the list of `workers`, sending invocations to the first until it becomes invalid, to then pass to the next ones in order. The `invalidate` strategy of the block regards the maximal number of concurrent invocations over the labelled workers—`max_concurrent_invocations`, which is set to `100`. If all the worker labels are invalid, the scheduler applies the `followup` behaviour, which is to `fail`.

Summarising, given a policy tag, the scheduler considers the corresponding `workers` blocks starting from the top. A block includes three parameters:

- `workers`: contains a non-empty list of worker labels or the `"*"` wildcard to encompass all of them;
- `strategy`: defines the policy of worker label selection. APP currently supports three strategies:
  - `random`: labels are selected in a fair random manner;
  - `best_first`: labels are selected following their order of appearance;

```
default:
  - workers: "*"
    strategy: platform
    invalidate: overload

couchdb_query:
  - workers:
    - DB_worker1
    - DB_worker2
    strategy: random
    invalidate: capacity_used: 50%
  - workers:
    - near_DB_worker1
    - near_DB_worker2
    strategy: best_first
    invalidate: max_concurrent_invocations: 100
  followup: fail
```

Fig. 3. Example of an APP script.

  - `platform`: labels are selected following the default strategy of the serverless platform—in our prototype (cf. Section 4) the `platform` option corresponds to the algorithm based on identifier hashing with co-prime increments explained in Section 2.

– `invalidate`: specifies when to stop considering a worker label. All invalidate options below include as preliminary condition the unreachability of the corresponding workers. When all labels in a block are invalid, the next block or the followup behaviour is used. Current `invalidate` options are:

  - `overload`: the corresponding workers lack enough computational resources to run the function;[1]
  - `capacity_used`: the corresponding workers reached a threshold percentage of CPU load (although not being overloaded);
  - `max_concurrent_invocations`: the corresponding workers have reached a threshold number of buffered concurrent invocations.

– `followup`: specifies the policy applied when all the blocks in a policy tag are considered invalid. The supported followup strategies are:

  - `fail`: stop the scheduling of the function;
  - `default`: follow what is defined in the `default` tag.

[1] The kind of computational resources that determine the `overload` option depends on the APIs provided by a given serverless platform. For example, in our prototype in Section 4 we consider a worker label `overload`ed when the related invokers are declared "unhealthy" by the OpenWhisk APIs, which use memory consumption and CPU load.

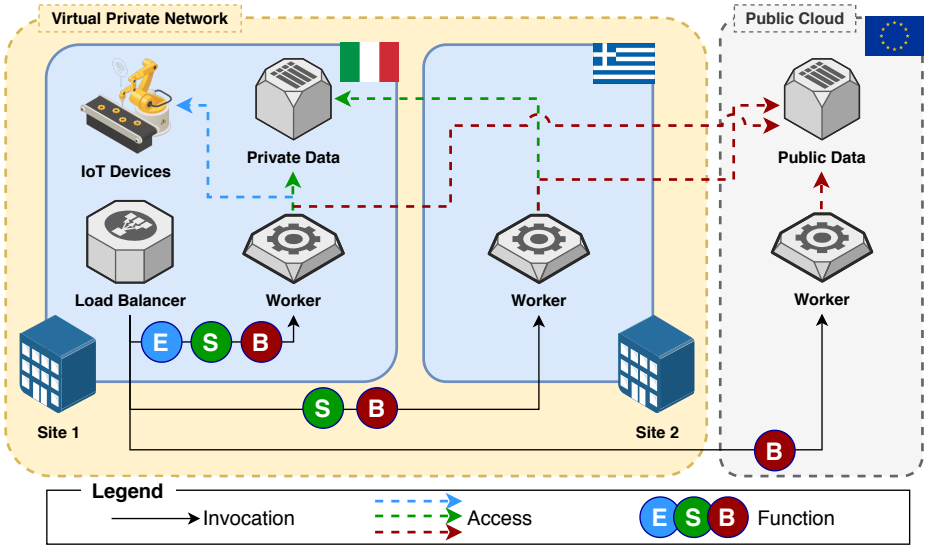

Fig. 4. Use case architecture representation.

4 Implementation in Apache OpenWhisk

We have implemented a serverless platform in which load balancing policies can be customised using the APP language. This implementation (available at `https://github.com/giusdp/openwhisk`) was obtained by modifying the OpenWhisk code base. Namely, we have replaced the load balancer module in the OpenWhisk controller, with a new one that reads an APP script, parses it, and follows the specified load balancing policies when OpenWhisk invokers should be selected[2].

To test our implementation, we used the Serverless use case depicted in Fig. 4 encompassing three Serverless domains: *i*) a private cloud with a low-power edge-device **Worker** at a first location, called **Site 1**; *ii*) a private cloud with the **Worker** at **Site 1** and a mid-tier server **Worker** at a second location, called **Site 2**; *iii*) a hybrid cloud with the two **Worker**s at **Site 1** and **Site 2** and a third mid-tier server from a **Public Cloud**. **Site 1** and **Site 2** are respectively located in Italy and Greece while the **Public Cloud** is located in northern Europe.

Site 1 is the main branch of a company and it runs both a data storage of **Private Data** and the **IoT Devices** used in their local line of production. **Site 1** also hosts the scheduler of functions, called the **Load Balancer**. The **Worker** at **Site 1** can access all resources within its site. **Site 2** hosts a **Worker** which, belonging to the company virtual private network (VPN), can access the **Private Data** at **Site 1**. The company also controls a **Worker** in a **Public Cloud** and a data storage with **Public Data** accessible by all **Worker**s.

[2] In this paper we chose to associate one worker label with one invoker. Future developments can use labels to identify pools of resources, following, e.g., recent proposals to change OpenWhisk invokers with Cluster Managers https://bit.ly/3cxYnTB).

In the use case, three different function deployments need to co-exist in the same infrastructure, marked as E , S , and B . Function E (edge) manages the **IoT Devices** at **Site 1** and it can only execute on the edge **Worker** at the same location, which has access to those devices. Function S (small) is a light-weight computation that accesses the **Private Data** storage at **Site 1**, within the company VPN. Function B (big) performs heavy-load queries on the **Public Data** storage in the **Public Cloud**. As mentioned, here data locality plays an important part in determining the performance of Serverless function execution:

- the **Worker** at **Site 1** can execute all functions. It is the only worker that can execute E and it is the worker with the fastest access to the co-located **Private Data** for S . It can execute B undergoing some latency due to the physical distance with the **Public Data** storage;
- the **Worker** at **Site 2** can execute functions S and B , undergoing some latency on both functions due to its distance from both data storages;
- the **Worker** at the **Public Cloud** can execute B , enjoying the fastest access to the related **Public Data** source.

Finally, besides data locality, the scheduler should also take into account how heavily the functions impact on the load of each **Worker**, considering that the **Worker** in the **Public Cloud** is as powerful as the one at **Site 2**, followed by the **Worker** at **Site 1**, which is a low-power edge device.

Experimental Results We compare the differences on the architecture and performance of the use case above as implemented using our APP-based OpenWhisk prototype against a naïve implementation using the vanilla OpenWhisk.

Specifically, we implement the use case using a Kubernetes cluster composed of a low-power device—with an Intel Core i7-4510U CPU with 8GB of RAM—in Italy for **Site 1**, a Virtual Machine—comparable to an Amazon EC2 a1.large instance—from the Okeanos Cloud (`https://okeanos.grnet.gr`) located in Greece for **Site 2**, and a Virtual Machine—comparable to an Amazon EC2 a1.large instance—from the **Public Cloud** of Microsoft Azure located in Northern Europe.

Following the requirements of the use case, we define the APP deployment plan for the use case as follows (we put the three tags in column for compactness):

```
Function_E:
  - workers:
    - worker_site1
  followup: fail
```

```
Function_S:
- workers:
    - worker_site2
    - worker_site1
    strategy: random
  followup: fail
```

```
Function_B:
  - workers:
    - worker_public_cloud
    - worker_site2
    - worker_site1
    strategy: best_first
    followup: fail
```

Commenting the code above, we have function E represented by `Function_E`, where the only invoker available is the one at **Site 1** (`worker_site1`). Since we do

not allow other invokers to handle (E), we set the `followup` value to `fail`. For (S) we have `Function_S`, where the invokers available are the ones at **Site 1** and **Site 2** (`worker_site2`). We let the two invokers split evenly the load of invocations, assigning `random` as routing `strategy`. Also here we let the invocation `fail` since we do not have other invokers able to access the **Private Data** storage within the company VPN. Finally, the policy for (B) (`Funcion_B`) includes all workers (hence also `worker_public_cloud` besides the ones at **Site 1** and **Site 2**) selected according to the `best_first` strategy. As for (S), also here we let the invocation `fail` since no other invokers are available.

For the APP-based deployment, we locate the Load Balancer at **Site 1** registering to it the three **Worker**s/invokers from **Site 1**, **Site 2** and the **Public Cloud**. For the naïve implementation, we use the same cluster but we install three separate but co-existing vanilla OpenWhisk instances. The three separate instances are needed to implement the functional requirements of limiting the execution of function (E) only on the Italian **Worker**, of (S) only on the Italian and Greek **Workers**, and of (B) on all **Worker**s.

To implement the databases (both **Private** and **Public** ones) we used a CouchDB instance deployed at **Site 1** and another in the **Public Cloud**. To simulate the access to IoT devices at **Site 1** (function (E)) we implemented a JavaScript function that, queried, returns some readings after a one-second delay. We followed a similar strategy for (S) and (B), where two JavaScript functions perform a (respectively lighter and heavier) query for JSON documents.

Architectural Evaluation An evident problem that arises with the triple-deployment combination is the increased consumption of computational and memory resources to host 3 copies of all the components, most importantly the Controller and the Invoker. A partial solution to this is to deploy separately the Kafka, Redis, and CouchDB components used by OpenWhisk, configuring them to be used by the three different installations simultaneously. However, we did not perform such optimisation to minimise the differences between the two tested architectures.

Quantitative Evaluation To have statistically relevant figures to compare the two setups (the APP-based and the vanilla one), we fired a sequence of 1000 requests for each function in each setup. We report the results of the tests of the APP-based implementation in Table 1 and those of the vanilla one in Table 2. In both tables, the first column on the left reports the tested function. The three following columns report the number of requests served by the respective **Worker**s at **Site 1**, **Site 2**, and in the **Public Cloud**. The last two columns report the time passed from sending a request to the reception of its response: the second-to-last column reports the average time (in ms) and the last one reports the average time (in ms) for the fastest 95^{th} percentile of request-responses.

We comment on the results starting from (E) (first row from the header in both tables). As expected, all requests for (E) are executed at **Site 1**. The slight difference in the two averages (APP ca. 5.6% faster than vanilla) and the two

	Site 1	Site 2	Public Cloud	Average (ms)	95% Average (ms)
E	1000	0	0	1096.53	1019.03
S	466	534	0	149.18	90.86
B	0	90	910	105.18	64.62

Table 1. 1000 invocation for each function in the APP-based OpenWhisk deployment.

	Site 1	Site 2	Public Cloud	Average (ms)	95% Average (ms)
E	1000	0	0	1159.90	1025.52
S	19	981	0	385.30	302.08
B	185	815	0	265.69	215.793

Table 2. 1000 invocations for each function in the vanilla OpenWhisk deployment.

fastest 95$^{\text{th}}$ percentile (APP ca. 0.6% faster than vanilla) come from the heavier resource consumption of the vanilla deployment.

As expected, the impact of data locality and the performance increase provided by the data-locality-aware policies in APP become visible for S and B . In the case of S , the Load Balancer of the vanilla deployment elected **Site 2** as the location of the main invoker (passing to it 98.1% of the invocations). We remind that S accesses a **Private Data** storage located at **Site 1**. The impact of data locality is visible on the execution of S in the vanilla deployment, being 88.35% slower than the APP-based deployment on average and 107.5% slower for the fastest 95$^{\text{th}}$ percentile. On the contrary, the APP-based scheduler evenly divided the invocations between **Site 1** (46.6%) and **Site 2** (53.4%) with a slight preference for the latter, thanks to its greater availability of resources. In the case of B , the Load Balancer of the vanilla deployment elected again **Site 2** as the location of the main invoker (passing to it 81.5% of all the invocations) and **Site 1** as the second-best (passing the remaining 18.5%). Although available to handle computations, the invoker in the **Public Cloud** is never used. Since B accesses a **Public Data** storage located in the **Public Cloud**, also in this case the effect of data locality is strikingly visible, marking a heavy toll on the execution of B in the vanilla deployment, which is 86.5% slower than the APP-based deployment on average and 107.8% slower for the fastest 95$^{\text{th}}$ percentile. The APP-based scheduler, following the preference on the **Public Cloud**, sends the majority of invocations to the **Public Cloud** (91%) while the invocations that exceed the resource limits of the **Worker** in the **Public Cloud** are routed to **Site 2** (9%), as defined by the `Function_E` policy.

As a concluding remark over our experiment, we note that these results do not prove that the vanilla implementation of OpenWhisk is generally worse (performance-wise) than the APP-based one. Indeed, what emerged from the experiment is the expected result that, without proper information and software infrastructure to guide the scheduling of functions with respect to some optimisation policies, the Load Balancer of OpenWhisk can perform a suboptimal

scheduling of function executions. Hence, there was a chance that the Load Balance of OpenWhisk could have performed some better scheduling strategies in our experiment, however that would have been an occasional occurrence rather than an informed decision. Contrarily, when equipped with the proper information (as it happens with our APP-based prototype) the Load Balancer can reach consistent results, which is the base for execution optimisation.

5 Related Work

While the industrial adoption of Serverless is spreading [19], it is a hot research topic due to its "untapped" potential [9,11,12,1].

Regarding the optimisation of Serverless function scheduling, Kuntsevich et al. [20] present an analysis and benchmarking approach for investigating bottlenecks and limitations of Apache OpenWhisk Serverless platform, while Shahrad et al. [21] report on the performance implications of using a Serverless architecture (over Apache OpenWhisk), showing how its workloads go against the locality-preserving architectural assumptions common in modern processors.

One of the main approaches explored in the literature to improve Serverless performance through function scheduling comes from improving the warm- vs cold-start of functions [12,1]. Those techniques mainly regard containers re-utilisation and function scheduling heuristics to avoid setting up new containers from scratch for every new invocation. However, other techniques have been recently proposed in the literature. Mohan et al. [22] present an approach focused on the pre-allocation of network resources (one of the main bottlenecks of cold starts) which are dynamically associated with new containers. Abad et al. [23] present a package-aware scheduling algorithm that tries to assign functions that require the same package to the same worker. Suresh and Gandhi [24] present a function-level scheduler designed to minimise provider resource costs while meeting customer performance requirements.

Besides resource re-utilisation, other approaches tackle the problem of optimising function scheduling with new balancing algorithms. Steint [25] and Akkus et al. [26] proposed new algorithms for Serverless scheduling, respectively using a non-cooperative game-theoretic load balancing approach for response-time minimisation and a combination of application-level sandboxing with a hierarchical message bus. Sampé et al. [27] present a technique to move computation tasks to storage workers with the aim to exploit data locality with small, stateless functions that intercept and operate on data flows.

Baldini et al. [19] focus on the programming of compositions of Serverless functions. In particular, they demonstrate that Serverless function composition requires a careful evaluation of trade-offs, identifying three competing constraints that form the "Serverless trilemma", i.e., that without specific run-time support, compositions-as-functions must violate at least one of the three constraints. To solve the trilemma, they present a reactive core of OpenWhisk that enables the sequential composition of functions.

Other works explored how to apply the Serverless paradigm to contexts like Fog/Edge and IoT Computing. The work presented in [28] studies the emergence of real-time and data-intensive applications for Edge Computing and proposes a Serverless platform designed for it. The work in [29] introduces instead a framework for supporting Multi-Provider Serverless Edge Computing to schedule executions across different providers.

Hall et al. [30] show how containers introduce an overhead unsuitable for Edge applications (requiring low-latency response or with hardware limitations), proposing a Serverless platform based on WebAssembly as a lighter environment to run Serverless applications in Edge scenarios. In [31] the authors present a variant of Edge Computing called "Deviceless" Edge Computing, where a prototypical architecture supports the distributed pooling and scheduling of geographically sparse devices with a high tolerance to network disruption and location-aware scheduling of functions.

Besides optimising Serverless scheduling, a common denominator of the works described above is that many extend or experiment with Apache OpenWhisk, which is also the technology we used to implement our prototype. Indeed, a line of future work on APP can test its expressiveness by capturing and implementing the policies presented in those works, so that users can choose to use them in their function deployments. In this context, APP is an encompassing solution *i*) able to let Serverless providers offer those scheduling strategies as options to their users, who can then choose which of them best suit their needs and *ii*) able to let different scheduling policies coexist in the same platform, while now researchers and implementors provide them as ad-hoc, incompatible implementations.

Recent work tackled the problem of formally reasoning on Serverless architectures. Gabbrielli et al. [32] present a core calculus for Serverless, combining ideas from both the λ-calculus (for functions, equipped with futures) and the π-calculus (for communication), paired with a repository of function definitions. On a similar research direction, Jangda et al. [33] present a formal model for Serverless architectures, also inspired by the λ-calculus, equipped with two semantics: a more involved one that captures the low-level details of function implementations and a simpler one that omits low-lever details of computation to ease reasoning on the interactions among Serverless functions. These two works offer formalisms that can be used to automatically reason on the properties of APP-defined function deployments. Future works can explore new policies that, through static analyses, capture details of function execution able to optimise their scheduling.

6 Conclusion

We addressed the problem of *function-execution scheduling optimisation*, proposing a methodology that provides developers with a declarative language called APP to express scheduling policies for functions. We extended the scheduler of OpenWhisk to use APP-defined policies in the scheduling of Serverless functions and empirically tested our extension on a use case that combines IoT, Edge, and Cloud Computing, contrasting our implementation with a naïve one using the

vanilla OpenWhisk stack to achieve the same functional requirements. We believe that APP can be seamlessly integrated in other Serveless platforms.

Besides the future investigations centred around the exploration of locality principles (e.g., code and session locality) as outlined in Section 5, an interesting line of work is to evolve APP to be able to express the definition of in-policy elements—such as scheduling strategies (`strategy`) and invalidation rules (`invalidate`)—directly in the source APP configuration, next to the ones given as "primitives" by the scheduler (e.g., `platform` or `best_first` strategies).

We are also interested in studying heuristics that, based on the monitoring of existing serverless applications, can suggest to its developer optimising scheduling policies. A starting point for this are configurator optimisers such as [34] that can be extended to automatically generate policies based on developer requirements.

Finally, we would like to investigate the separation of concerns between developers and providers, trying to minimise the information that providers has to share to allow developers to schedule functions efficiently, while, at the same time, hide the complexity of their dynamically changing infrastructure.

References

1. E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019.
2. L. Baresi and D. F. Mendonça, "Towards a serverless platform for edge computing," in *IEEE ICFC 2019*, pp. 1–10, IEEE, 2019.
3. AWS, "AWS IoT Greengrass." https://aws.amazon.com/greengrass/. Acc. 04/2020.
4. AWS, "Lambda." https://aws.amazon.com/lambda/. Acc. 04/2020.
5. "Apache openwhisk." https://openwhisk.apache.org/, 2019. Acc. 04/2020.
6. Microsoft, "Azure Functions." https://azure.microsoft.com/services/functions. Acc. 04/2020.
7. Google, "Cloud Functions." https://cloud.google.com/functions. Acc. 04/2020.
8. Iron.io, "IronFunctions." https://open.iron.io. Acc. 04/2020.
9. S. Hendrickson, S. Sturdevant, E. Oakes, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *login Usenix Mag.*, vol. 41, no. 4, 2016.
10. IBM, "Cloud Functions." https://www.ibm.com/cloud/functions. Acc. 04/2020.
11. I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, pp. 1–20, Springer, 2017.
12. J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *CIDR*, www.cidrdb.org, 2019.
13. D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
14. M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," *University of California, Berkeley, Rep. UCB/EECS*, vol. 28, no. 13, p. 2009, 2009.

15. L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX/ATC 18)*, pp. 133–146, 2018.
16. Q. Xie, M. Pundir, Y. Lu, C. L. Abad, and R. H. Campbell, "Pandas: robust locality-aware scheduling with stochastic delay optimality," *IEEE/ACM Trans. on Networking*, vol. 25, no. 2, pp. 662–675, 2016.
17. W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," *IEEE/ACM Trans. Netw.*, vol. 24, pp. 190–203, 2016.
18. O. Ben-Kiki, C. Evans, and B. Ingerson, "Yaml ain't markup language (yaml™) version 1.1," *Working Draft 2008-05*, vol. 11, 2009.
19. I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," in *ACM Onward! 2017*, pp. 89–103, 2017.
20. A. Kuntsevich, P. Nasirifard, and H.-A. Jacobsen, "A distributed analysis and benchmarking framework for apache openwhisk serverless platform," in *Middleware (Posters)*, pp. 3–4, 2018.
21. M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *MICRO'52*, pp. 1063–1075, 2019.
22. A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *HotCloud 19*, 2019.
23. C. L. Abad, E. F. Boza, and E. V. Eyk, "Package-aware scheduling of faas functions," in *ACM/SPEC ICPE*, pp. 101–106, ACM, 2018.
24. A. Suresh and A. Gandhi, "Fnsched: An efficient scheduler for serverless functions," in *WOSC@Middleware*, pp. 19–24, ACM, 2019.
25. M. Stein, "The serverless scheduling problem and noah," *arXiv preprint arXiv:1809.06100*, 2018.
26. I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX/ATC 18)*, pp. 923–935, 2018.
27. J. Sampé, M. Sánchez-Artigas, P. García-López, and G. París, "Data-driven serverless functions for object storage," Middleware '17, p. 121–133, Association for Computing Machinery, 2017.
28. L. Baresi and D. F. Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE ICFC*, pp. 1–10, IEEE, 2019.
29. A. Aske and X. Zhao, "Supporting multi-provider serverless computing on the edge," in *ICPP, Workshop Proceedings*, pp. 20:1–20:6, ACM, 2018.
30. A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," IoTDI '19, (New York, NY, USA), p. 225–236, ACM, 2019.
31. A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: Extending serverless computing to the edge of the network," SYSTOR '17, (New York, NY, USA), ACM, 2017.
32. M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, and S. P. Zingaro, "No more, no less," in *International Conference on Coordination Languages and Models*, pp. 148–157, Springer, 2019.
33. A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–26, 2019.
34. E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, and J. Mauro, "Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies," in *SETTA*, pp. 229–245, 2016.