Custom Serverless Function Scheduling Policies: An APP Tutorial

Giuseppe De Palma[1]
Università di Bologna, Italy

Saverio Giallorenzo
Università di Bologna, Italy
INRIA, Sophia Antipolis, France

Jacopo Mauro
University of Southern Denmark, Odense, Denmark

Matteo Trentin
Università di Bologna, Italy
University of Southern Denmark, Denmark

Gianluigi Zavattaro
Università di Bologna, Italy
INRIA, Sophia Antipolis, France

Abstract

State-of-the-art serverless platforms use hard-coded scheduling policies that hardly accommodate users in implementing functional or performance-related scheduling logic of their functions, e.g., preserving the execution of critical functions within some geographical boundaries or minimising data-access latencies. We addressed this problem by introducing APP: a declarative language for defining per-function scheduling policies which we also implemented as an extension of the open-source OpenWhisk serverless platform. Here, we present a gentle introduction to APP through an illustrative application developed over several incremental steps.

2012 ACM Subject Classification Software and its engineering → Scheduling; Computer systems organization → Cloud computing

Keywords and phrases Serverless, Function Scheduling, Declarative Languages, Tutorial

Digital Object Identifier 10.4230/OASIcs.Microservices.2020-2022.5

Supplementary Material *Software (Source Code)*: `https://github.com/giusdp/openwhisk/tree/old_app`, archived at `swh:1:dir:ba4498167bdd7b5c4a2cbc676dcc04e8df6e8354`

Funding Research partly supported by the H2020-MSCA-RISE project ID 778233 "Behavioural Application Program Interfaces (BEHAPI)".

1 Introduction

Serverless is a cloud computing model that has become increasingly popular in recent years. In serverless, a provider manages the dynamic allocation of resources needed to satisfy some inbound requests (such as HTTP requests, database updates, or scheduled events), removing from the shoulders of programmers the burden of managing and scaling both the infrastructure and runtime platforms.

[1] Corresponding author

Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022).
Editors: Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmerman; Article No. 5; pp. 5:1–5:16
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Besides easing software deployment, serverless adopts a per-usage cost model, where costs correspond to the resources used to answer requests, which ditch the expenses of running idle servers when no requests reach the system. Examples of serverless platforms include AWS Lambda [9], Microsoft Azure Functions [10], and Google Cloud Functions [24].

A common trait of these platforms is that they manage the allocation of functions over the available computing nodes, also called *workers*, following opinionated policies that favour some performance principle. Indeed, effects like *code locality* [28] – due to latencies in loading function code and runtimes – or *session locality* [28] – due to the need to authenticate and open new sessions to interact with other services – can sensibly increase the run time of functions. As a consequence, there exists ever-growing literature on serverless scheduling techniques that mix one or more of these locality principles to increase the performance of function execution [41, 37, 35, 1, 42, 50, 36, 32, 47, 44, 45, 48, 43, 16, 33, 13, 15, 30, 34, 46]. Besides performance, functions can have functional requirements that constrain the choices of the scheduler. For example, users might want to avoid allocating their functions alongside "untrusted" ones for security purposes [11, 51, 4, 18].

Albeit a FaaS platform can mix one or more principles to expand the profile coverage of function execution, platform-wide policies hardly suit all kinds of performance profiles. For this reason, in [20, 19] we started a new line of research dedicated to the introduction of a modular approach to FaaS scheduling. The approach hinges on the definition of custom, per-function policies through a domain-specific declarative language, called *Allocation Priority Policies* (APP). Thanks to APP, users can define co-existing scheduling policies, each best suited for its function (or sets thereof). In [20, 19], we validated our approach by extending the open-source Apache OpenWhisk serverless platform to support APP scripts and by showing that our extensions outperform vanilla OpenWhisk in locality-bound scenarios [20, 19].

In this paper, we present a gentle introduction to APP, by means of a tutorial. Even though APP is a platform-independent language, to exemplify its usage, we choose to describe it in the context of the OpenWhisk platform, which is both one of the most studied serverless platforms in the literature [26] and actively developed among the open-source alternatives.

We start our presentation in Section 2 by discussing the OpenWhisk serverless platform and introduce the APP language to control the scheduling of functions. To keep this tutorial at an introductory level, we consider the first, minimal version of APP, as appeared in [20]. We illustrate the language through realistic, incremental use cases in Section 3. We assume the reader to have basic knowledge of software programming, computer architectures and networks, and of cloud concepts like virtual machines and containers. The interested reader can deploy the extended OpenWhisk platform used in this tutorial by using Terraform and Ansible scripts provided at [40].

In Section 4, we discuss both language primitives found in an extended version of APP published in [19] and ongoing work on future extensions. We conclude in Section 5 by discussing related work and drawing final remarks.

2 Serverless Computing and the APP Language

We start by introducing in more concrete terms FaaS and the details of serverless platforms (relevant to function scheduling) by describing the architecture and execution model of Apache OpenWhisk. We deem these preliminaries useful to understand the constructs found in the APP language, presented afterwards.

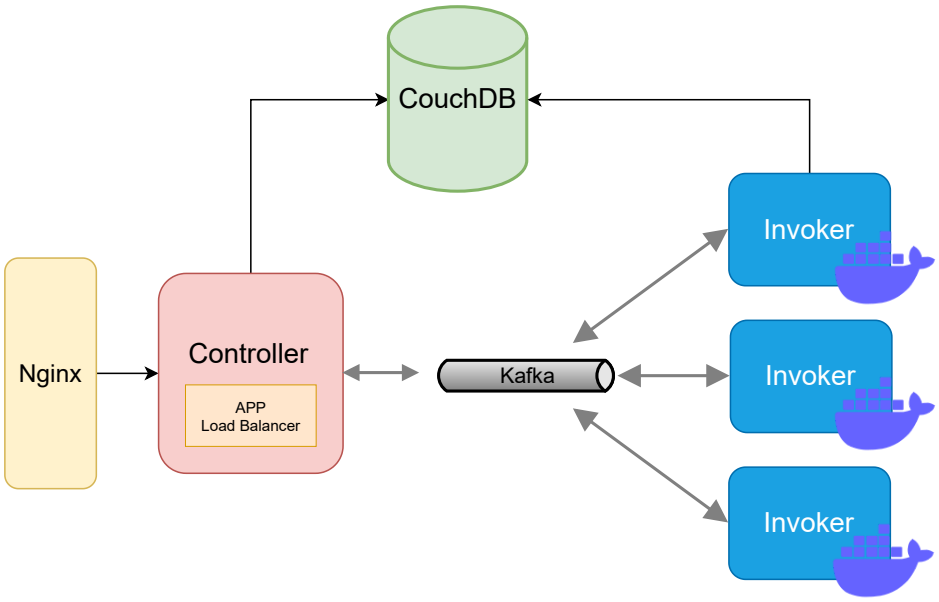

Figure 1 Diagram of the OpenWhisk Architecture with the Controller component adapted to host APP-based custom scheduling policies.

2.1 Apache OpenWhisk

Apache OpenWhisk [38] is an open-source serverless computing platform, widely-used in research and adopted by some cloud providers (IBM Cloud Functions [17] and Digital Ocean Functions [21]). In the OpenWhisk programming model, developers write "actions" which are stateless functions that run on the platform. These functions can be written in any supported programming language (e.g., Go, Java, JavaScript, PHP, Python, Ruby, Swift).

Functions can be executed by defining rules associating a trigger (e.g., an event like HTTP requests) with a certain function. When an event is received, OpenWhisk identifies the appropriate function to execute based on the event type and the function's declared trigger. Each function is executed within a container that is created and managed by OpenWhisk. The platform scales the number of containers up or down based on the workload, so developers do not have to worry about provisioning or managing infrastructure.

OpenWhisk consists of five components (see Figure 1). The entry point for requests is an Nginx reverse proxy that redirects requests to (one or more) Controllers. Controllers are the characterising component of the architecture. They manage the overall state of the system, including request handling, user authentication, and function deployment. Since they are the component that chooses which worker shall execute a given function, Controllers work also as a Load Balancers. Note that, in Figure 1, we label workers as Invoker(s), which is the nomenclature used in the OpenWhisk project to dub workers. The message broker used to communicate between Controllers and workers is Apache Kafka [6]. Workers are responsible for creating Docker containers, initialising them with functions' code, and running the functions. A CouchDB database is used to store the state of the system, including information about users, functions, triggers, rules, invocation requests, and invocation results.

Controllers use a hardcoded scheduling policy dubbed "co-prime scheduling". The co-prime logic allocates functions on workers by associating a function to a hash and a step size. The hash finds the primary worker, called the "home" worker. The step size finds a

list of workers used in succession when the preceding ones become unavailable, e.g., due to overload. This routine allows OpenWhisk's scheduler to route invocations of the same function to the same worker(s), resulting in a high probability of cached container reuse. This practice minimises the occurrence of "cold starts", i.e., the increased latency in function execution due to the overhead of fetching the code of the function to put it in execution.

2.2 The APP Language

Similarly to OpenWhisk, the other FaaS platforms adopted in the industry come with hardcoded scheduling policies which may not always meet the specific needs of developers and providers. For example, OpenWhisk maximises container reuse to avoid cold starts, but it does not take into account other factors that can impact the latency and improve performance (least of all satisfy individual functional requirements on scheduling). As an example of performance improvement, let us have a function that accesses a database. Running that function on a pool of machines close to the database would reduce network traffic and latency.

The motivation behind APP is to allow a FaaS platform to follow specific scheduling policies depending on which function must be scheduled for execution. APP helps users optimise their serverless architectures by allowing them to define customised FaaS scheduling policies that instruct the platform on which workers are best suited to run each function. From the architectural point of view, letting serverless platforms support APP is straightforward and one mainly needs (as we did in our extensions [20, 19]) to modify the Controller (see Figure 1) so that it follows the scheduling policies defined in a given APP script.

Syntax-wise, APP adopts the current trend of configuration files in popular DevOps and Cloud tools (e.g., Kubernetes) by supporting a YAML [39]-compatible syntax, where users define scheduling policies in a terse, declarative way. To define function-specific policies, we assume to associate each function with a tag; then, in APP we name each policy with a tag, so that, at runtime, the Controller can pair each function with its APP policy and follow the latter's scheduling logic.[2]

We report the syntax of APP in Figure 2, as found in [20], of which we provide a brief overview in this section. In Section 3, we delve into more details with concrete examples.

The second assumption we make about the environment to run APP scripts is a 1-to-1 association so that each worker has a unique, identifying label. Indeed, the main, mandatory component of any policy (identified by a *policy_tag*) are the `workers` therein, i.e., a collection of labels that identify on which workers the scheduler can allocate the function. Specifically, each policy has a list of one or more *block*s, each including other two optional parameters besides the `worker` clause: the scheduling `strategy`, followed to select one of the workers of the block, and an `invalidate` condition, which determines when a worker cannot host a function. When a selected worker is invalid, the scheduler tries to allocate the function on the rest of the available workers in the block. If none of the workers of a block is available, then the next block is tried. The last clause, `followup`, encompasses a whole policy and defines what to do when no *block*s of the policy managed to allocate the function. When set to `fail`, the scheduling of the function fails; when set to `default`, the scheduling continues by following the (special) `default` policy.

We close by overviewing the options for both the `strategy` and `invalidate` parameters.

[2] The pairing of functions and policies is an orthogonal issue w.r.t. to APP. Indeed, one can obtain the same result with a 1-to-1 coupling between function identifiers and policies. However, we prefer the tag-based decoupling presented here because it is more flexible; e.g., it allows users to apply the same policy to multiple functions, as long as they are associated with the same tag.

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

Figure 2 The APP syntax.

The `strategy` parameter has 3 alternatives:

- `platform` indicates the usage of the platform-specific scheduling logic for the workers of the block. In practice, it delegates the scheduling decision to the platform's default scheduler (e.g., in OpenWhisk, it would use the original "co-prime" scheduling logic, cf. Section 2.1). This strategy is the default, when `strategy` is omitted, i.e., when there is no particular need to change the scheduler behaviour for a given function.
- `random` allocates functions stochastically among the workers of the block, following a uniform distribution. This strategy is useful in scenarios where balancing the distribution of functions over workers is more important than avoiding cold starts (e.g., when the system usually receives bursts of invocations).
- `best-first` allocates functions on workers based on their top-down order of appearance in the block. This strategy can decrease functions' run time, e.g., when workers have different performances and the user orders them by their best-to-worst performance ranking.

The `invalidate` parameter allows users to define when a worker becomes invalid to execute a given function. For instance, the hardcoded policy in OpenWhisk for worker invalidation is that it either exhausted its memory capacity (where each function consumes 256 MB) or reached the limit of 30 concurrent function invocations.

APP supports 3 `invalidate` options:

- `overload` (the standard one, when `invalidate` is omitted) captures the hardcoded invalidation strategy of the platform (e.g., in OpenWhisk, it translates to the exhaustion of the memory capacity or 30 concurrent invocations);
- `capacity_used`, associated with a percentage value, determines the threshold of memory capacity that declares a worker invalid;
- `max_concurrent_invocations`, associated with an integer value, sets the upper limit of concurrent invocations allowed on a worker.

Note that both `max_concurrent_invocations` and `capacity_used` are refinements of the `overload` option, thus their effect is to reduce respectively the maximum number of concurrent invocations or the memory capacity of the `overload` option.

3 APP Running Example

We show how we can use APP to control the scheduling of functions through an incremental, running example. We start with separate functions and their scheduling policy cases of increasing complexity and conclude by tying these functions together into a single FaaS application and related APP script.

3.1 Step 0: A Simple, Pure Function

The lowest level of complexity for our application is a single, lightweight, pure function, i.e., a function whose output depends only on its input, and requires no interaction with external services; examples of this category are functions performing mathematical operations, pre-processing of data, etc. Since we assume no interactions with external APIs or databases nor other requirements on workers that could impact the scheduling, we can fall back to the vanilla, hardcoded scheduling policy of the host serverless platform. When there is no need for specific APP-based policies, the standard `default` policy captures the basic behaviour of the underlying platform.

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

If no function has a specific tag, they fall under the `default` tag policy. This tag only has one block, which targets all workers (`*` match all the possible works labels) and adopts the `platform` `strategy` and the `overload` `invalidate` condition.

3.2 Step 1: Handling Locality when Querying External Databases

For the next step in complexity, let us assume we have some functions that use an external data source. Serverless functions indeed usually have limits on the size of their input payloads (e.g., AWS Lambda limits the payload size to 256KB and 6MB for asynchronous and synchronous requests respectively)[3] and they normally interact with data storage services, like databases, to retrieve (and store) the data they work on.

The addition of interaction with databases can cause latency problems due to data locality, i.e., the latency of the function is higher if database access is slow (e.g., due to long-distance interactions or narrow bandwidth). Scheduling a function's execution on a poorly-performing worker might incur latency overheads. Let us assume that the database is in Canada and that we have three workers in the region: `Canada_worker0`, `Canada_worker1`, `Canada_worker2`. Let us also assume that the query functions are tagged with the string `queries`. To schedule the functions only on those workers we can use the following script.

```
- queries:
  - workers:
    - Canada_worker0
    - Canada_worker1
    - Canada_worker2
    strategy: platform
  followup: default
```

[3] `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html`.

We add the `queries` tag to label our new locality-bound functions, and require them to be scheduled only on workers that are located in Canada. We still keep the `platform` `strategy`, as we are not interested in giving any additional priority to one of the workers.

Note that we set the `followup` to `default`. This captures the fact that, while this kind of functions should be scheduled on workers that are close to the database (Canada), there is no strict functional requirement for it. Therefore, if neither of the preferred workers is available at the time of scheduling, it is acceptable for the function to be scheduled on some different workers, following the `default` policy.

3.3 Step 2: Prioritising Workers and Customising Invalidation

Building on the previous step, let us assume that the workers in Canada are not equivalent in terms of performance and we have a powerful worker labelled `Canada_worker_Large`, a less powerful worker labelled `Canada_worker_Small`, and a powerful worker with limited bandwidth labelled `Canada_worker_Large_Narrow_Bandwidth`.

We expect that the query function would perform the best on the `Canada_worker_Large` worker since it is both powerful and has wide bandwidth. When this worker is invalid, we have to choose between `Canada_worker_Large_Narrow_Bandwidth` and `Canada_worker_Small`. In this example, we prioritise the latter, as the computational limitations should come into play only when tasked with a relatively high number of invocations, while the former's reduced bandwidth might have a stronger effect on overall latency.

Moreover, while we want our workers to be prioritised according to their expected performance, we want to limit the resources that our functions are going to take to avoid scheduling too many functions in one worker, thus risking slowing down their run time. We set a limit of 75% memory load threshold for invalidation so that we never allocate more than $^3/_4$ of a given worker's capacity.

To satisfy all new requirements, we update the configuration script as follows.

```
- queries:
  - workers:
    - Canada_worker_Large
    - Canada_worker_Small
    - Canada_worker_Large_Narrow_Bandwidth
    strategy: best_first
    invalidate:
      capacity_used: 75%
  followup: default
```

Here, we list sequentially the workers in order of priority, changing the strategy to `best_first` and setting the `invalidate` condition as discussed above.

3.3.1 Step 2.5: Invalidation Conditions and Block Lists

In the previous script, all three workers share the same `invalidate` condition, which can be undesirable when the difference in computational power among the nodes is sensible. If more fine-grained control is needed, we could define multiple blocks for the `queries` tag. Since APP tries to schedule functions within a given policy block by block, in their top-to-bottom order of appearance, we can define a scheduling logic analogous to the `best_first` strategy seen in the previous section by distributing the workers in different blocks, each with their

specific `invalidate` options and exploiting the ordering of blocks to impose priority among them. Specifically, we set to 8 the threshold of maximal concurrent functions running on the `Large` workers, while we set it to 2 for the `Small` worker.

```
- queries:
  - workers:
    - Canada_worker_Large
    invalidate:
      max_concurrent_invocations: 8
  - workers:
    - Canada_worker_Small
    invalidate:
      max_concurrent_invocations: 2
  - workers:
    - Canada_worker_Large_Narrow_Bandwidth
    invalidate:
      max_concurrent_invocations: 8
  followup: default
```

3.4 Step 3: Enforcing Configurations and Discarding Defaults

Let us now broaden the components of our serverless application by adding, alongside our initial data-dependent functions, some computationally heavier functions. More precisely, we want also to schedule functions that specifically require the presence of a GPU on the worker, e.g., needed to execute vector-based calculations in parallel.

In the previous steps, we allowed `queries` functions to execute on any worker different from the preferred ones if all the latter are invalid, thanks to the `default followup` option. Here, we assume that the GPU-dependent functions cannot be scheduled on non-GPU hardware, either because the function can not be compiled on non-GPU hardware or because running in other nodes leads to overloading the CPUs, thus causing failures due to timeouts[4].

We capture this behaviour as follows.

```
- GPU:
  - workers:
    - GPU_worker0
    - GPU_worker1
    strategy: platform
  followup: fail
```

Unlike the previous examples, we now use the `fail` option for the `followup` keyword. This tells the scheduler to terminate the invocation with an error if the required workers are not available, without falling back to the `default` tag.

3.5 Step 4: Putting it all Together

Let us conclude this section of examples by tying all cases seen before into a consistent scenario. Indeed, this is the final step in complexity for our example application, where we combine all the features of APP seen so far.

[4] Serverless frameworks usually impose a maximal run time for functions that span a few minutes, e.g., OpenWhisk default maximal duration is 60 seconds.

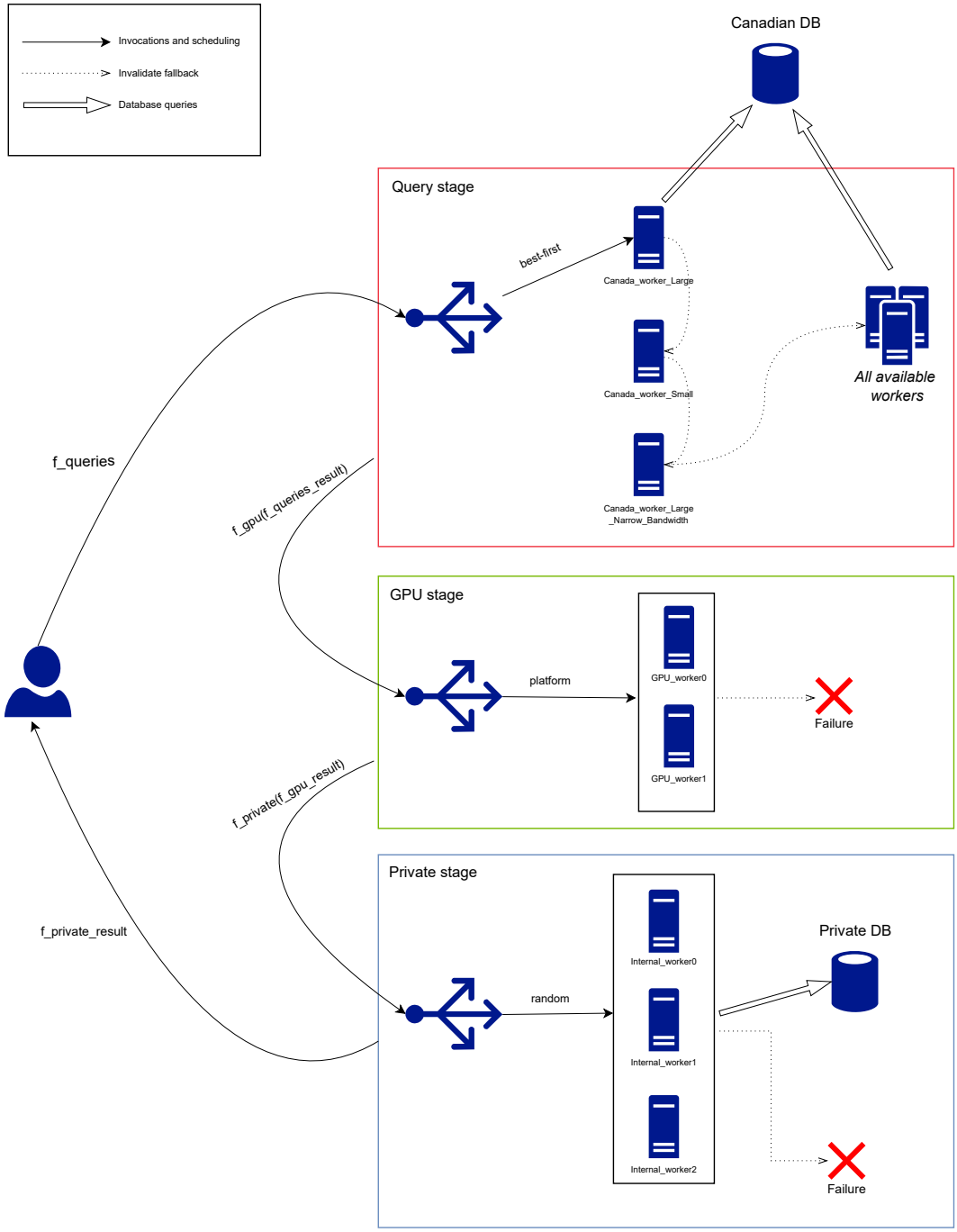

Figure 3 Architectural diagram of our example serverless application. Note how the `queries`-tagged functions fall back to select from all available workers when none of the preferred ones is available, by using the `default` policy.

Here, we assume that we are working in a hybrid cloud context, that is, we have both a private and a public part of our deployment: namely, besides the `Canada_worker`s and the `GPU_worker`s, previously presented and deployed on public cloud, we consider additional `Internal_worker`s deployed on-premises. Let us also assume that our private section has access to data inaccessible from the outside that can be used only by local workers, i.e. the additional `Internal_worker`s, for security reasons.

The application collects data from an external database, and subsequently pre-processes that data, using functions with the `queries` tag (discussed in Section 3.3.1). The output of these functions is then used to feed the `GPU` functions, which infer additional information from our data (discussed in Section 3.4). In the final step, the private database is queried and the received information is combined to enrich the information previously inferred. As such, we require the new functions implementing this final step to specifically target private workers.

To recapitulate, besides presenting the new `private` policy, we report the other policies seen in the previous sections, which make up the final APP script used to customise the scheduling of the functions in our example application.

```
- queries:
  - workers:
    - Canada_worker_Large
    invalidate:
      max_concurrent_invocations: 8
  - workers:
    - Canada_worker_Small
    invalidate:
      max_concurrent_invocations: 2
  - workers:
    - Canada_worker_Large_Narrow_Bandwidth
    invalidate:
      max_concurrent_invocations: 8
  followup: default
- GPU:
  - workers:
    - GPU_worker0
    - GPU_worker1
    strategy: platform
  followup: fail
- private:
  - workers:
    - Internal_worker0
    - Internal_worker1
    - Internal_worker2
    strategy: random
    invalidate:
      capacity_used :80%
  followup: fail
- default:
  - workers : *
    strategy: platform
    invalidate: overload
```

The final architecture of the application, which we depict in Figure 3, consists of a pipeline with three stages. In the first stage, `queries`-tagged functions retrieve data from a Canadian database and perform some pre-processing over it, and then they invoke `GPU`-tagged functions for machine learning tasks (second stage). In the third stage, the completed `GPU` functions invoke `private`-tagged functions which combine the processed data with private data accessible only to the `Internal_workers` (see the `worker` clause under the `private` policy in the code above). Note that the `default` tag (at the end of the reported snippet) can be used by the `queries`-tagged functions, but not by the `GPU` and `private` functions, which all `fail` in case all their workers are invalid. When the `private`-tagged functions are invoked, only the private part of the system (the one containing the `Internal` workers and private database) is involved since the `private` tag lists only the `Internal` workers. In this example, we set the `strategy` to `random` to uniformly distribute the functions over the three workers. The `followup` set to `fail` naturally captures the fact that it would be pointless to run the `private` functions outside the private part of the system, since those workers would not be able to reach the private database and the `private` functions would fail.

4 APP Extensions

The original version of APP, presented in the previous sections, inspired extensions that introduced new language primitives. In this section, we discuss a couple of primitives respectively found in an extended version of APP published in [19] and from ongoing work, motivating them with concrete examples.

4.1 Targeting sets of workers

In cloud settings, the addition or removal of computing nodes is often done unbeknownst to the applications running on them. As a consequence, if the function scheduling language allows for referring to the computing nodes only individually, it may be impossible to define scheduling policies that exploit dynamic scenarios where nodes can change at runtime. An extension of APP to handle these more dynamic scenarios is to drop the 1-to-1 relation imposed between labels and workers. Specifically, we can allow the same worker to have multiple labels (e.g., a unique one to identify it and multiple, shared ones) and then add an APP primitive to express when we intend that a label induces a collection of workers.

As an example, we change the scenario in Section 3.3.1 to have three *families* of workers (instead of three workers): the `Large`, `Small`, and `Large_with_Narrow_Bandwidth` tiers.

In [19], we extended the APP language with the possibility to use expressions to match various tags of workers. Here, we present a refinement of that idea, introducing the `set` keyword to indicate that we intend worker labels as shared among a collection of workers, which the scheduler can choose among. As an example, the following APP script adds the `set` keyword before the workers' label found in the script from Section 3.3.1 to indicate that APP shall interpret it as the collection of workers associated with that label.

```
- queries:
  - workers:
    - set: Canada_worker_Large
    invalidate:
      max_concurrent_invocations: 8
  - workers:
    - set: Canada_worker_Small
```

```
    invalidate:
      max_concurrent_invocations: 2
  - workers:
    - set: Canada_worker_Large_Narrow_Bandwidth
    invalidate:
      max_concurrent_invocations: 8
  followup: default
```

Thus, the `set` keyword allows users to capture dynamic scenarios where workers change at runtime without requiring them to update the script when the underlying infrastructure change. Moreover, the new keyword allows us to handle large amounts of workers in a cleaner way, making scripts more easily writable and readable.

4.2 Function Anti-affinity

A second primitive, subject of ongoing work, is an `anti-affinity` option for the `invalidate` condition. This option allows policies to invalidate a worker when it hosts one or more functions that are deemed by the user anti-affine with the one under scheduling. Anti-affinity is important, e.g., for security reasons since there could be functional requirements that require avoiding running critical functions on a worker with other, unknown functions, which could exploit possible limitations of the runtime isolation to surreptitiously gather information from the former. Anti-affinity constraints may also play a role in the optimisation of the application performance. For example, let us consider a refinement of the scenario presented in Section 3.4 where the `GPU_Workers` have only 1 GPU each. In this context, it would be ideal to avoid scheduling another GPU-intensive function on the same worker, as this could impact the performance of both functions.

Adding an `anti-affinity` `invalidate` option allows us to capture this scenario with minimal modifications from the script presented in Section 3.4.

```
- GPU:
  - workers:
    - set: GPU_Worker
      invalidate:
        anti-affinity: GPU
  followup: fail
```

In the code above, the scheduler avoids placing `GPU` functions on a `GPU_Worker` if it hosts already a function associated with the same tag. In the example, in particular, we declared any function with tag `GPU` anti-affine with any other function with the same tag. This allows the execution of at most one function with tag `GPU` in all the workers with tag `GPU_Worker`.

We highlight that the introduction of anti-affinity constraints may be problematic for serverless applications of considerable size and introduce interesting research challenges. Indeed, to sustain high-traffic situations, serverless platforms (like OpenWhisk) consider the presence of multiple controllers, so that they can share the load of the inbound function invocations and avoid creating architectural bottlenecks. However, having multiple controllers can introduce scheduling races, so that multiple controllers asynchronously schedule functions on the same worker (this effect is generally not a problem since, at worst, the worker would non-deterministically reject allocations that exceed its capacity limits). In the context of anti-affinity, this issue becomes more relevant. Indeed, to guarantee the respect of anti-affinity constraints one might need to sensibly alter the serverless platform architecture. For example,

one can use global locks – which might determine sensible performance degradation – or require the partitioning of workers among the available controllers – which would thwart the principle of resource sharing of cloud computing.

5 Discussion and Conclusion

We presented a tutorial on APP, an innovative approach to providing fine-grained control over serverless function scheduling to users. The interested reader can retrieve an OpenWhisk extension that supports APP-based scripts at [7].

To the best of our knowledge, APP is the first platform-agnostic configuration language for serverless scheduling. As serverless computing is gaining wider adoption [11, 27], many proposals tackled the problem of improving serverless function scheduling under different application contexts (and locality principles). In particular, there are several works focused on optimising serverless function scheduling focusing on improving the cold-start problem. These include techniques focused on container re-utilization and function scheduling heuristics and dedicated balancing algorithms [31, 27, 36, 35, 42, 1, 49, 3, 41]. Other research directions in the field regard the programming of compositions of serverless functions and the application of the Serverless paradigm to contexts such as Fog/Edge and IoT Computing. Examples of the first direction are proposals of calculi to formally reason on serverless functions and their implementations [22, 29] as well as proposals of the elements underpinning runtime support for compositions-as-functions [12]. The second direction sees studies on the emergence of real-time and data-intensive applications for edge computing and proposed a serverless platform designed for it, as well as frameworks for supporting cloud-to-edge serverless computing [14, 8, 25, 23].

Regarding APP's evolution, we want to explore the advantages offered by APP from both the language and implementation perspectives. On the latter, we plan to extend the number of platforms that support APP, besides our initial prototype based on OpenWhisk. As a prerequisite to support APP, we need nodes to be labelled. Looking at the architectures of alternative open-source serverless platforms, like OpenFaaS and Knative, adding support for one such prerequisite would follow the implementation we developed for OpenWhisk. Although we do not have precise knowledge of the internals of closed-sourced alternatives, e.g., AWS Lambda and Azure Functions, they likely do not label nodes or expose this information to the scheduler (since they do not support scheduling policies based on the identification of single nodes or groups thereof). However, since FaaS platforms tend to all share the same architecture, we deem it plausible to use our development on OpenWhisk as a guideline to add support for node labelling also in closed-source platforms.

From the language perspective, we propose to extend APP, focusing on the exploration of locality principles (e.g., code and session locality) and in providing users with constructs able to support custom definitions of `strategy` and `invalidate` options directly in the source APP configuration (also via shareable and importable modules). These extensions would enable greater flexibility and customisation of scheduling policies. We are also interested in studying heuristics that, based on the monitoring of existing serverless applications can suggest optimising scheduling policies. We deem configurator optimisers [2, 5] a good starting point for this activity, which can be extended to automatically generate policies based on developer's requirements. Finally, we propose to investigate the separation of concerns between developers and providers, to minimise the information that providers have to share to allow developers to schedule functions efficiently. This balancing would involve hiding the complexity of providers' dynamically changing infrastructure, while also allowing developers to customise their scheduling policies to meet their needs.

References

1 Cristina L. Abad, Edwin F. Boza, and Erwin Van Eyk. Package-aware scheduling of faas functions. In *Proc. of ACM/SPEC ICPE*, pages 101–106. ACM, 2018. doi:10.1145/3185768.3186294.

2 Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro. Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016, Beijing, China, November 9-11, 2016, Proceedings*, volume 9984 of *Lecture Notes in Computer Science*, pages 229–245, 2016. doi:10.1007/978-3-319-47677-3_15.

3 Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *Proc. of USENIX/ATC*, pages 923–935, 2018. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

4 Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, 2018. doi:10.1145/3276488.

5 Amazon Web Services. AWS Compute Optimizer. https://aws.amazon.com/compute-optimizer/, aug 2023.

6 Apache kafka. https://kafka.apache.org/, aug 2023.

7 APP-based openwhisk extension. https://github.com/giusdp/openwhisk, aug 2023.

8 Austin Aske and Xinghui Zhao. Supporting multi-provider serverless computing on the edge. In *ICPP, Workshop Proceedings*, pages 20:1–20:6. ACM, 2018. doi:10.1145/3229710.3229742.

9 Aws lambda. https://aws.amazon.com/lambda/, aug 2023.

10 Microsoft azure functions. https://azure.microsoft.com/, aug 2023.

11 Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017. doi:10.1007/978-981-10-5026-8_1.

12 Ioana Baldini, Perry Cheng, Stephen J Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. In *ACM Onward! 2017*, pages 89–103, 2017. doi:10.1145/3133850.3133855.

13 Ali Banaei and Mohsen Sharifi. Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform. *The Journal of Supercomputing*, sep 2021. doi:10.1007/S11227-021-04057-Z.

14 Luciano Baresi and Danilo Filgueira Mendonça. Towards a serverless platform for edge computing. In *2019 IEEE ICFC*, pages 1–10. IEEE, 2019. doi:10.1109/ICFC.2019.00008.

15 Luciano Baresi and Giovanni Quattrocchi. Paps: A serverless platform for edge computing infrastructures. *Frontiers in Sustainable Cities*, 3:690660, 2021.

16 Giuliano Casale, Matej Artač, W-J Van Den Heuvel, André van Hoorn, Pelle Jakovits, Frank Leymann, Mike Long, Vasilis Papanikolaou, Domenico Presenza, Alessandra Russo, et al. Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):77–87, 2020. doi:10.1007/S00450-019-00413-W.

17 IBM Cloud. Ibm cloud functions. https://cloud.ibm.com/functions/, aug 2023.

18 Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. In *Proceedings of The Web Conference 2020*, pages 939–950, 2020. doi:10.1145/3366423.3380173.

19 Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. A declarative approach to topology-aware serverless function-execution scheduling.

In *2022 IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 11–15, 2022*. IEEE, 2022. doi:10.1109/ICWS55610.2022.00056.

20 Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. Allocation priority policies for serverless function-execution scheduling optimisation. In *Proc. of ICSOC*, volume 12571 of *LNCS*, pages 416–430. Springer, 2020. doi:10.1007/978-3-030-65310-1_29.

21 DigitalOcean. Digitalocean functions. https://www.digitalocean.com/products/functions/, aug 2023.

22 Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Pio Zingaro. No more, no less - A formal model for serverless computing. In *Proc. of COORDINATION*, volume 11533 of *LNCS*, pages 148–157. Springer, 2019. doi:10.1007/978-3-030-22397-7_9.

23 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, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3078468.3078497.

24 Google cloud functions. https://cloud.google.com/functions/, aug 2023.

25 Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, pages 225–236, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3302505.3310084.

26 Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. Survey on serverless computing. *Journal of Cloud Computing*, 10(1):1–29, 2021. doi:10.1186/S13677-021-00253-7.

27 Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *CIDR*. www.cidrdb.org, 2019. URL: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

28 Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *Proc. of USENIX HotCloud*, 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

29 Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. of ACM on Prog. Lang.*, 3(OOPSLA):1–26, 2019. doi:10.1145/3360575.

30 Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proc. of ACM SIGOPS SOSP*, pages 691–707, New York, NY, USA, 2021. ACM. doi:10.1145/3477132.3483541.

31 Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019. URL: http://arxiv.org/abs/1902.03383.

32 Stefan Kehrer, Jochen Scheffold, and Wolfgang Blochinger. Serverless skeletons for elastic parallel processing. In *2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM). IEEE*, pages 185–192, 2019.

33 Daniel Kelly, Frank Glavin, and Enda Barrett. Serverless computing: Behind the scenes of major platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 304–312. IEEE, 2020. doi:10.1109/CLOUD49709.2020.00050.

34 Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *Proc. of USENIX ATC*, pages 805–820. USENIX Association, 2021. URL: https://www.usenix.org/conference/atc21/presentation/kotni.

35 Aleksandr Kuntsevich, Pezhman Nasirifard, and Hans-Arno Jacobsen. A distributed analysis and benchmarking framework for apache openwhisk serverless platform. In *Proc. of Middleware (Posters)*, pages 3–4, 2018. doi:10.1145/3284014.3284016.

36 Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *Proc. of HotCloud 19*, Renton, WA, jul 2019. USENIX Association. URL: https://www.usenix.org/conference/hotcloud19/presentation/mohan.

37 Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

38 Apache openwhisk. https://openwhisk.apache.org/, aug 2023.

39 Ingy döt Net Oren Ben-Kiki, Clark Evans. Yaml ain't markup language (yaml™) version 1.2. https://yaml.org/spec/1.2.2/, 2021.

40 APP-based openwhisk deployment. https://github.com/giusdp/ow-gcp, aug 2023.

41 Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. Data-driven serverless functions for object storage. In *Middleware*, Middleware '17, pages 121–133. ACM, 2017. doi:10.1145/3135974.3135980.

42 Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proc. of MICRO*, pages 1063–1075, 2019. doi:10.1145/3352460.3358296.

43 Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. of USENIX ATC*, pages 205–218, 2020. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

44 Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proc. of USENIX ATC*, pages 419–433. USENIX Association, 2020. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

45 Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proc. of Middleware*, Middleware '20, pages 1–13, New York, NY, USA, 2020. ACM. doi:10.1145/3423211.3425682.

46 Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*, pages 17–25. IEEE, 2022. doi:10.1109/ICFEC54809.2022.00010.

47 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)*, pages 144–153, 2020. doi:10.1109/IC2E48712.2020.00022.

48 Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, jul 2020. URL: http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf.

49 Manuel Stein. The serverless scheduling problem and noah. *CoRR*, abs/1809.06100, 2018. arXiv:1809.06100.

50 Amoghavarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proc. of WOSC@Middleware*, pages 19–24. ACM, 2019. doi:10.1145/3366623.3368136.

51 Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.