Affinity-aware Serverless Function Scheduling

Giuseppe De Palma
Università di Bologna
Italy
INRIA
France

Saverio Giallorenzo
Università di Bologna
Italy
INRIA
France

Jacopo Mauro
University of Southern Denmark
Denmark

Matteo Trentin
Università di Bologna
Italy
INRIA
France
University of Southern Denmark
Denmark

Gianluigi Zavattaro
Università di Bologna
Italy
INRIA
France

Abstract

Functions-as-a-Service (FaaS) is a Serverless Cloud paradigm where a platform manages the scheduling (e.g., resource allocation, runtime environments) of stateless functions. Recent work proposed using domain-specific languages to express per-function policies, e.g., policies that enforce the allocation on nodes that enjoy lower latencies to databases and services used by the function. Here, we focus on *affinity-aware* scenarios, i.e., where, for performance and functional requirements, the allocation of a function depends on the presence/absence of other functions on nodes.

We present aAPP, an extension of a declarative, platform-agnostic language that captures affinity-aware scheduling at the FaaS level. We implement an aAPP-based prototype on Apache OpenWhisk. Besides proving that a FaaS platform can capture affinity awareness using aAPP and improve performance in affinity-aware scenarios, we use our prototype to show that aAPP imposes no noticeable overhead in scenarios without affinity constraints.

ACM Reference Format:
Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. 2025. Affinity-aware Serverless Function Scheduling. In *Proceedings of International Conference on Software Architecture 2025 (ICSA '25).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Functions-as-a-Service (FaaS) is a programming paradigm supported by the Serverless Cloud execution model [23]. In FaaS, developers implement a distributed architecture from the composition of stateless functions and delegate concerns like execution runtimes and resource allocation to the serverless platform, thus focusing on writing code that implements business logic rather than worrying about infrastructure management. The main cloud providers offer FaaS [4, 12, 28] and open-source alternatives exist too [20, 21, 31, 33].

A common denominator of these platforms is that they manage the allocation of functions over the available computing resources, also called *workers*, following opinionated policies that favour some performance principle. Indeed, effects like *code locality* [21]—due to latencies in loading function code and runtimes—or *session locality* [21]—due to the need to authenticate and open new sessions to interact with other services—can substantially increase the run time of functions. The breadth of the design space of serverless scheduling policies is witnessed by the growing literature focused on techniques that mix one or more of these locality principles to increase the performance of function execution, assuming some locality-bound traits of functions [9, 11, 22, 24, 25, 41]. Besides performance, functions can have functional requirements that the scheduler shall consider. For example, users might want to ward off allocating their functions alongside "untrusted" ones—common threat vectors in serverless are limited function isolation and the ability of functions to (surreptitiously) gather weaponisable information on the runtime, the infrastructure, and the other tenants [2, 8, 13, 45].

Although one can mix different principles to expand the profile coverage of a given platform-wide scheduler policy, the latter hardly suits all kinds of scenarios. This shortcoming motivated De Palma et al. [14, 16] to introduce a domain-specific, platform-agnostic, declarative language, called *Allocation Priority Policies* (APP) to specify custom function allocation policies. Thanks to APP, the same platform can support different scheduling policies, each tailored to meet the specific needs of a set of related functions. De Palma et al. validated their approach by implementing an APP-based serverless platform as an extension of the open-source Apache OpenWhisk project.

Our contributions originate from the observation that, at lower levels of the cloud stack, popular Infrastructure-as-a-Service (IaaS) platforms (e.g., OpenStack [32]) and Container-as-a-Service (CaaS) systems (e.g., Kubernetes [26]) allow users to express affinity and anti-affinity constraints about the allocation of VM/containers—e.g., anti-affinity constraints, to reduce overhead by shortening data paths via co-location, to increase reliability by evenly distributing VM/containers among different nodes, and for security, such as preventing the co-location of VM/containers belonging to different trust tiers.

ICSA '25, March 31–April 4, 2025, Odense, Denmark
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

On the contrary, FaaS platforms do not natively support the possibility to express affinity-aware scheduling, where function allocation depends on the presence (affinity) or absence (anti-affinity) at scheduling time of other functions in execution on the available workers.

Contribution. Recognising the potential of FaaS-level affinity-aware scheduling policies, we propose a language-based solution, obtained by extending APP into an affinity-aware function scheduling language called aAPP. In Sec. 2, we present an example of affinity-aware scheduling at the FaaS level that we use to informally introduce aAPP. We formalise our proposal in Sec. 3, where we present the aAPP syntax and discuss the increment of expressiveness w.r.t. APP. In Sec. 4, we concretise our proposal by presenting a prototype implementation of an aAPP-based serverless platform as an extension of Apache OpenWhisk able to enforce aAPP-defined FaaS (anti-)affinity scheduling constraints. In Sec. 5, we experimentally show that the usage of (anti-)affinity constraints are beneficial by considering an implementation of the affinity-aware scenario introduced in Sec. 2. In Sec. 6 we compare the performance of our aAPP-based prototype and vanilla OpenWhisk with 7 benchmarks to show that aAPP imposes negligible overhead. We discuss related work and draw concluding remarks in Sec. 7.

2 Example of an Affinity-aware FaaS Scenario

We have a *divide-et-impera* data-crunching serverless application implemented through two companion functions. The first, invoked by the users, is called *divide*. Its task is to split some data into chunks, store them in a database, and invoke instances of the second function. The second function, which the *divide* invokes for each stored chunk, is called *impera*. Its task is to retrieve a chunk of data from the database and process it.

We run the above functions on the FaaS infrastructure depicted on the left of Fig. 1. The infrastructure includes two zones (e.g., separate regions of a cloud provider) and it has a *Gateway* that decides on which worker to allocate the execution of the functions. The infrastructure also includes three workers: w_1 and w_2 in $Zone_1$ and w_3 in $Zone_2$. Each zone hosts an instance of an eventually-consistent distributed database [44], used by the functions running in that zone—eventually-consistent systems are typical for (FaaS) scenarios like ours, where one favours throughput and availability w.r.t. e.g., overall data consistency [7].

In Fig. 1, we represent function allocation requests with labelled document icons sent to the *Gateway*. Note that the users (the laptop icons in Fig. 1) launch the *divide* function (e.g., d_3) while the running *divide* (e.g., d_2 requesting i_2 and i'_2) invoke the *impera* functions.

Our FaaS infrastructure executes other functions besides the one above. In Fig. 1, we represent these requests with the labels h_1, h_2, and h_3 which are compute-intensive functions—called *heavy*—that use a high amount of computational resources of the worker running them.

Given this context, an initial example of an affinity-aware scheduling policy is to avoid the co-occurrence of the *divide* and *impera* functions with the *heavy* ones. In this way, we can improve the performance of *divide* and *impera* by avoiding resource contention with the *heavy* functions. Another improvement regards the interaction with the database. The eventual-consistency behaviour of the database entails possible delays to synchronise the instances. Waiting for synchronisation is necessary only when the functions accessing the database connect to different database instances. Moreover, to further reduce delay, we can exploit the principle of *session locality* and let functions running on the same worker share the same connection with the database. This affinity-aware scheduling policy places *impera* functions only on workers that already host *divide* functions and avoid the overhead of re-establishing new connections.

These constraints can be encoded in aAPP as shown in the script in Fig. 1. This code has three top-level items: `d`, `i`, and `h`. These are tags that identify policies, each describing the scheduling logic of a set of related functions. In the example, the tag `d` describes the logic for the *divide* functions while `i` and `h` target respectively the *impera* and *heavy* ones. The line `workers: *` found under all tags indicates that their related functions can use any of the available workers. From the top, under tag `d`, we use the `affinity` clause, introduced by aAPP, to specify that `d`-tagged functions should *not* be scheduled on a worker that currently hosts *heavy* functions (`!h`). Specifically, this is an example of *anti-affinity*, where we prevent the allocation of the tagged functions (e.g., `d`) on a worker that already hosts any anti-affine function (e.g., tagged `h`). Tag `i` declares the same anti-affinity for *heavy* functions, but it also indicates that `i`-tagged functions are *affine* with `d`-tagged ones. Affinity means that we can schedule a function on a candidate worker only if it currently hosts the former's affine functions. In the example, we use affinity to have *impera* functions run in the same worker of *divide* functions. Finally, we use tag `h` to complement the anti-affinity relation expressed in the previous tags, i.e., the *heavy* functions are anti-affine with both `d` and `i` functions and shall not be scheduled in workers that already host any of the latter.

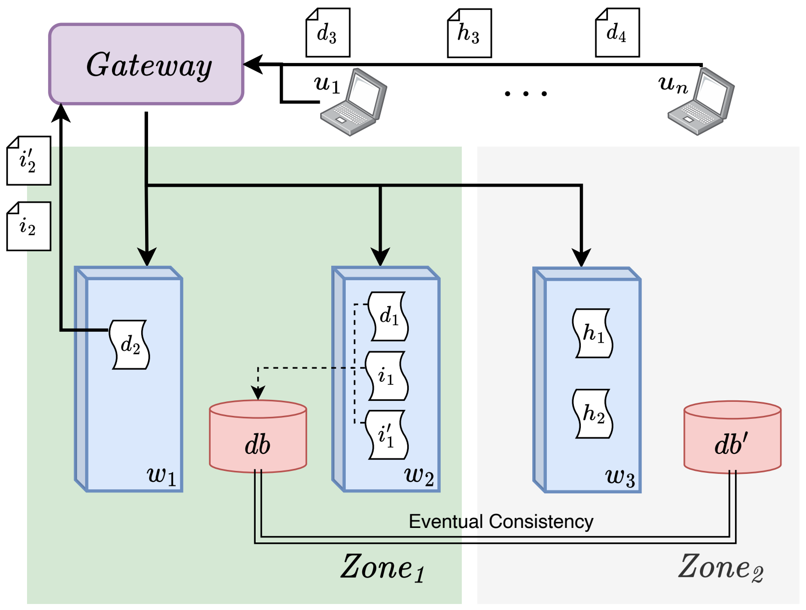
Notably, we purposefully do not identify who writes the aAPP script in the example, e.g., the developer of the functions or the administrator of the platform. Indeed, aAPP (in general, APP and all its extensions) caters to different cloud stakeholders for scheduling policy definition. For instance, if we contextualise our example in a local private cloud setup, then users can directly write their own aAPP scripts because they have direct knowledge of the infrastructure nodes. Contrarily, if we are in a managed cloud environment, the cloud provider would use aAPP to implement and enforce scheduling requirements specified by their clients based on their workflows—e.g., synthesising aAPP scripts from function and workflow code [35, 36].

3 The aAPP Language

We now present aAPP, our extension of the FaaS function scheduling language APP [14, 16] with affinity and anti-affinity constraints.

We report in Fig. 2 the syntax of aAPP. From here on, we indicate syntactic units in *italics*, optional fragments in *grey*, terminals in `monospace`, and lists with $\overline{bars}$. The syntax of aAPP draws inspiration from YAML [46], a renowned data-serialisation language for configuration files—e.g., many modern cloud tools, like Kubernetes and Ansible, use this format.[1] In aAPP, functions have associated a tag that identifies some scheduling policies. An aAPP script represents: *i)* named scheduling policies identified by a *tag* and *ii)* policy *block*s that indicate either some collection of workers, each identified by a worker *id*, or the universal `*`. To schedule a function, we use its tag to retrieve the scheduling policy that includes one or more blocks of possible workers. To select the worker, we iterate top-to-bottom

[1]While aAPP scripts are YAML-compliant, for presentation, we stylise the syntax to increase readability. For instance, we omit quotes around strings, e.g., `*` instead of `"*"`.



```
- d:
  - workers: *
    affinity:
      - !h
- i:
  - workers: *
    affinity:
      - !h
      - d
- h:
  - workers: *
    affinity:
      - !d
      - !i
```

Figure 1: Examples of a FaaS infrastructure (left) and an aAPP script (right).

$id \in Identifiers \qquad n \in \mathbb{N}$

app	$::=$	$\overline{-tag}$
tag	$::=$	$id : \overline{-\ block}$ `followup` $: f\_opt$
$block$	$::=$	`workers` $: w\_opt$ `strategy` $: s\_opt$
		`invalidate` $: \overline{-\ i\_opt}$ `affinity` $: \overline{-\ a\_opt}$
w_opt	$::=$	`*` $\mid \overline{-\ id}$
s_opt	$::=$	`any` $\mid$ `best_first`
i_opt	$::=$	`capacity_used` n`%` $\mid$ `max_concurrent_invocations` n
a_opt	$::=$	$id \mid !id$
f_opt	$::=$	`default` $\mid$ `fail`

Figure 2: aAPP syntax.

```
- f_tag:
  - workers:
    - local_w1
    - local_w2
    strategy: best_first
    invalidate:
    - capacity_used 80%
    affinity: g_tag,!h_tag
  - workers:
    - public_w1
  followup: fail
```

Figure 3: Example aAPP script.

on the blocks. We stop at the first block that has a non-empty list of valid workers and then select one of those workers according to the strategy defined by the block (described later).

Each tag can define a `followup` clause, which specifies what to do if the policy of the tag did not lead to the scheduling of the function; either `fail`, to terminate the scheduling, or `default` to apply the special `default`-tagged policy. Each block can define a `strategy` for worker selection (`any` selects non-deterministically one of the available workers in the list; `best_first` selects the first available worker in the list), a list of constraints that `invalidate`s a worker for the allocation (`capacity_used` invalidates a worker if its resource occupation reaches the set threshold; `max_concurrent_invocations` invalidates a worker if it hosts more than the specified number of functions), and an `affinity` clause that carries a list containing affine tag identifiers *id* and anti-affine tags, represented by negated tag identifiers *!id*. aAPP is a minimal extension of APP where we add the `affinity` clause to capture (anti-)affinity constrains. Similar to the notion of affinity introduced by Microsoft in its IaaS offering [29], in aAPP, the relation of (anti-)affinity is "directional": we do not impose any properties like symmetry or anti-symmetry on affinity or anti-affinity to capture as many useful scenarios as possible and avoid imposing well-formedness properties that limit its expressiveness.[2]

We show an example two-block aAPP policy for functions tagged `f_tag` in Fig. 3. The first block restricts allocations on the workers labelled `local_w1` and `local_w2` and the latter on `public_w1`. The first block specifies as invalid (i.e., which cannot host the function under scheduling) the workers that reach a memory consumption above 80%. Since the strategy is `best_first`, we allocate the function on the first valid worker; if none are valid, we proceed with the

[2]If we had symmetric anti-affinity, we would not capture scenarios where, e.g., a function `init` is the seeding function for a database and function `query` manipulates that data. The function `init` should always run before `query` but never where `query` is already running, while function `query` should run where `init` is present. To obtain this behaviour, we need `init` anti-affine with `query` but `query` affine with `init`.

next block. The function has affinity with g_tag and anti-affinity with h_tag. Hence, a valid worker requires the presence of at least a function with tag g_tag and no functions with tag h_tag. If both the first and second blocks do not find a valid worker, the scheduling of the function fails (instead of continuing with the default tag).

Notably, the addition of (anti-)affinity constraints strictly increases the expressiveness of APP [37]. Indeed, APP can capture anti-affinity constraints only by severely limiting the flexibility of resource allocation, e.g., either through a partition of the workers and a static assignment of anti-affine functions to distinct partitions, or by limiting artificially the capacity that can be used or the number of functions we can allocate on a worker. This approach conflicts with the cloud principle of resource sharing and optimization. The situation for affinity is even poorer: APP cannot capture these constraints because it does not keep track of the functions currently allocated on the workers.

4 aAPP-based Apache OpenWhisk

We have implemented and validated an aAPP-based FaaS platform, obtained by extending the APP prototype of Apache OpenWhisk—an open-source, serverless platform initially developed by IBM and donated to the Apache Software Foundation.

In Fig. 4 we represent the usual flow followed by function invocations in OpenWhisk. The *Entrypoint* for function execution requests is an Nginx reverse proxy which passes the requests to a *Controller* responsible for forwarding them, via a Kafka *Message Broker*, to *Invoker* components—the workers, in OpenWhisk's lingo.

The figure also depicts the main intervention we performed to make the existing APP extension of OpenWhisk aAPP-compliant. The extension concerns two parts: the APP parser and the *ConfigurableLoadBalancer*, both absent in vanilla OpenWhisk and originally introduced in APP-based OpenWhisk [16]. The parser was extended to add compatibility for aAPP scripts, while the *ConfigurableLoadBalancer* was extended to keep track of the functions allocated to all the workers. We introduced two lookup tables, called *activeFunctions* and *activeTagActivations*, to implement the tracking functionality. The first table associates the allocated functions (and their tags) to their host worker and allows the load balancer to verify affinity and anti-affinity constraints. The *activeTagActivations* table keeps tracks of the state of the different function instances (possibly of the same function definition, so we cannot use their identifiers) by pairing their *activation ids* with their function identifiers; when we observe the termination of an active function, we look its function identifier up and remove that instance from the *activeFunctions* table—we detect instance terminations thanks to the messages workers send to notify the load balancer of their completion. Finally, we adjusted the internal *schedule* function to integrate these new changes and select workers based on (anti-)affinity requirements.

The logic that implements the scheduling semantics of aAPP scripts is straightforward. We present it in (Python-like) pseudocode in Listing 1. In Listing 1, the schedule function requires the name of the function under scheduling (f), a map that represents the current infrastructure configuration (workers and functions running therein, explained later) (conf), an aAPP script encoded as a Python dictionary of objects (aapp), and a registry that maps each function to a tag and its memory occupancy (reg). The infrastructure configuration maps, for each worker, the list of functions scheduled on it (fs), the memory allocated for those functions (memory_used), and the total amount of memory of the worker (max_memory).

Given these inputs, schedule gets the tag associated with f (Line 2) and then extracts the blocks associated with this tag in the aapp script (Line 3). If the followup option is different from "fail", we append the blocks associated with the default tag to the list of f's blocks (Line 5). Then, we obtain the list of valid workers for every block in order of appearance (Line 9). When the workers clause uses *, we consider all the workers present in the configuration (Line 8). If the list of valid workers is non-empty, we choose the first one when the strategy is best_first (Line 12) and a random one otherwise (Line 14). If the list is empty, the scheduling fails (Line 15).

The schedule function uses the valid function to check when a worker is valid, i.e., it is available, it has enough capacity to host the function (Lines 18–19), and that allocating on it the function satisfies all the constraints of capacity_used, max_concurrent_invocations (Lines 21–26), and affinity (Lines 27–34).

To implement our prototype, we have modified the Scala codebase of the OpenWhisk project; specifically, we have modified the scheduling algorithm to implement the logic of Listing 1 and manage the workers-functions status, on a fork of the OpenWhisk repository [3]. The entire system is easily deployable using Terraform and Ansible scripts.

5 Performance Improvements via Affinity-awareness

To validate our platform and show that the usage of (anti-)affinity constraints for affinity-aware scenarios are beneficial, we use the example presented in Sec. 2 as a benchmark. We show that, by enforcing (anti-)affinity constraints, we can reduce average execution times and tail latency.

Recalling the example, we consider a simple *divide-et-impera* serverless architecture running in a realistic co-tenancy context. Users invoke *divide* functions, requesting the solution of a problem. At invocation, *divide* splits the problem into sub-problems and invokes instances of the second function, *impera*. The *impera* instances solve their relative sub-problems and store their solution fragments on a persistent storage service. After the *imperas* terminated, *divide* retrieves the partial solutions, assembles them, and returns the response to the user. We consider a multi-zone execution context where each zone hosts an instance of an eventually-consistent distributed database. The workers in one zone access the local instance of the database. Another function, called *heavy*, represents possible interferences of serverless co-tenancy.

Experimental setup. To run the use case, we deploy the OpenWhisk versions of APP and aAPP on a 8-node Kubernetes cluster on the Digital Ocean platform; one node acts as the control plane (and as such, it is unavailable to OpenWhisk), one hosts the OpenWhisk core components (i.e., the Controller, the OpenWhisk internal database CouchDB, and the messaging system Kafka), and six nodes are workers. We deploy the control plane and the OpenWhisk core components on virtual machines with 2 vCPU and 2 GB RAM, while we deploy 4 workers on virtual machines with 2 vCPU and 2 GB RAM and 2 workers with 1 vCPU and 1 GB RAM. All machines run the Ubuntu Server 20.04 OS. Location-wise, we place the control plane, the OpenWhisk core components, and 3 workers in Europe and 3

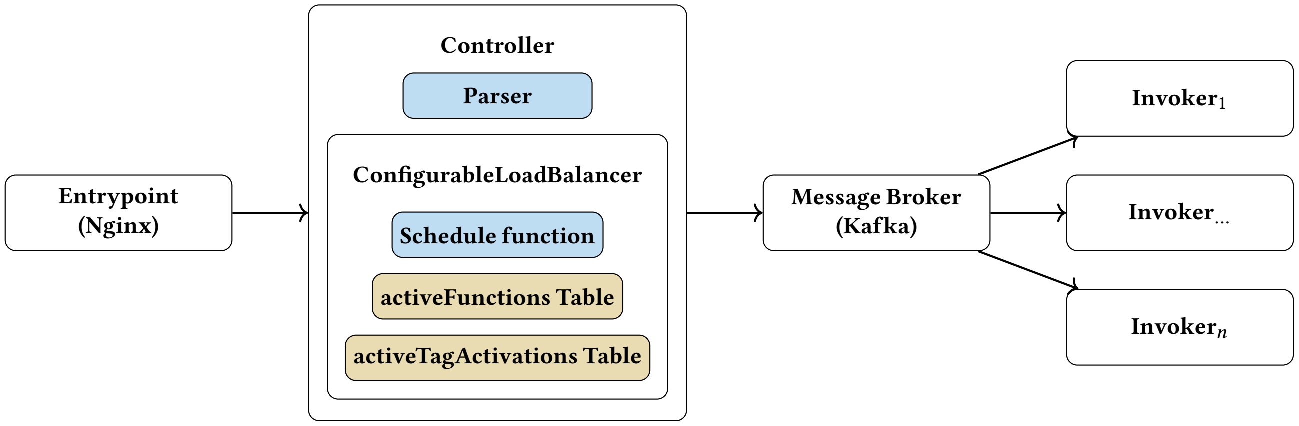


Figure 4: Extended Apache OpenWhisk for aAPP (modified modules in blue, added modules in yellow).

workers in North America (2 with the more powerful configuration and 1 with the lesser one in each zone). To implement persistent storage, we deploy a 2-node MongoDB replica set, one in Europe and one in North America, using the 6.0.2 version of the Community Server. We distribute the load generated by the *heavy* functions on the platform with two variants, *heavy_eu* and *heavy_us*, which we constrain to be resp. allocated in the Europe and the North America data centres on the less powerful workers (identified with *workereu1* and *workerus1*), to further amplify the effect of co-tenancy they exert.

All functions are in JavaScript and run on OpenWhisk NodeJS runtime *nodejs:14*. The *divide* function invokes two instances of the *impera* functions with 3 parameters: i) a freshly generated index, ii) an array populated with 100 random numbers, and iii) the initial–final indexes of the values to work on (0–49 to the first *impera* instance, 50–99 to the second one). After, it waits for the *impera* functions to terminate, and then opens a connection to the local instance of MongoDB with the aim of retrieving documents representing the results computed by the *impera* functions. The *divide* function has the ability to detect the correct documents doing a query on the MongoDB by using the freshly generated index. The *impera* functions simulate a computation on the values received from the *divide* function: it connects to the local instance of MongoDB and stores on it one document for each received value. Such document contains one of the value and the freshly generated index received upon invocation. In this way, each execution of the *impera* function stores in MongoDB 50 documents. Finally, the *heavy* function simulates computing resource consumption by performing 1 billion iterations of a computation consisting of the random generation of two numbers and the execution of their multiplication.

Notice that the *divide* function needs to retrieve from MongoDB the documents generated by the two *impera* function it invokes. Each function opens a connection to the local instance of MongoDB, and in case an *impera* function is executed in a different zone w.r.t. the *divide* function it can take some time for the two database instances to converge. The *divide* function implements an exponential back-off retry approach [34]—it tries to fetch the documents from its local storage instance; if the data is not there, starting from a 1-second delay, the function waits for a back-off time that exponentially increases at each retry.

Experiments and Results. In our experiment, we consider three APP/aAPP scripts to showcase the benefits of (anti-)affinity constraints. The first, which uses the full expressiveness of aAPP, is the one reported in Fig. 6—where *impera*s (tagged with i) are affine with *divide* (tagged with d) and they are both anti-affine with the *heavy* functions (tagged with h_eu and h_us). The affinity between *divide* and *impera* functions is used to guarantee that the database writer (the *impera* function) and reader (the *divide* function) access the same database instance. Instead, the anti-affinity with the *heavy* functions is used to avoid resource contention between the *divide*/*impera* functions and the *heavy* functions. The second script removes the affinity constraints between *impera* and *divide* from the first script (anti-affinity-only-aAPP). The third script omits the anti-affinity constraints from the second one, effectively making it an APP script, with no consideration for affinity.

Each experiment involves 5 sequential runs. Each run invokes the *heavy_eu* and *heavy_us* functions in non-blocking mode, followed by 10 calls of the *divide* function, each one waiting for the previous to complete. Upon termination of the *heavy* functions, we proceed with the remaining runs; for a total of 10 *heavy* and 50 *divide* functions per experiment. To ensure reliable results, we run the experiment 5 times, totalling 250 calls of the *divide* function for each of the three APP/aAPP script. We use Apache JMeter to simulate each request, tracking its latency, number of retries (to retrieve storage data), and outcomes (success or failure).

The results match our expectations. The mean and median latency for the divide functions in aAPP is resp. 1547ms and 883ms, while the 95^{th} tail latency is 3041ms. The corresponding figures increase for anti-affinity-only-aAPP: 2337ms (+40%), 2381ms (+91%), and 3476ms (+13%). As expected, the latency increases even more substantially for APP, with respectively (percentage increase vs aAPP) 8118ms (+135%), 2648ms (+99%), and 60157ms (+180%).

To further analyse the differences, in Fig. 5, we report the scatter plots where we sort the latencies of the *divide* functions from the shortest to the longest (x-axis). We focus on this measure because it offers a comprehensive overview of the performance of the architecture. In particular, it includes the latencies of the related *impera* functions and its latencies are concretely the ones experienced by the users interacting with the system. The first striking observation

```
def schedule(f, conf, aapp, reg):
 (memory, tag) = reg[f]
 blocks = aapp[tag].blocks # get the blocks
 if aapp[tag].followup != 'fail':
  blocks += aapp['default'].blocks # add default tag blocks
 for block in blocks:
  if '*' in block['workers']:
   block['workers'] = conf.keys
  workers = [ for worker in block['workers'] if valid(f,worker,conf,reg,block)]
  if len(workers) > 0: # if at least one valid worker is found
   if block['strategy'] == 'best_first':
    return workers[0]
   elif block['strategy'] == 'any':
    return random.choice(workers)
 raise Exception('Function not schedulable')

def valid(f, w, conf, reg, block):
 (memory, tag) = reg[f]
 if (w not in conf) or (conf[w]['memory_used'] + memory > conf[w]['max_memory']):
  return False
 if 'invalidate' in block:
  if ('capacity_used' in block['invalidate']) and
     (block['invalidate']['capacity_used'] <= conf[w]['memory_used']):
    return False
  if ('max_concurrent_invocations' in block['invalidate']) and
    (block['invalidate']['max_concurrent_invocations'] <= len(conf[w]['fs'])):
   return False
 if 'affinity' in block:
  affine_tags = set([t for t in block['affinity'] if not t.startswith('!')])
  anti_affine_tags = set([t[1:] for t in block['affinity'] if t.startswith('!')])
  w_tags = set([t for (_, t) in [reg(f) for f in conf[w]['fs']]])
  for t in affine_tags:
   if t not in w_tags: return False
  for t in anti_affine_tags:
   if t in w_tags: return False
 return True
```

Listing 1: The pseudo-code of the schedule and valid functions.

is that the distribution of the aAPP data points is interrupted (there are almost no instances) between the 1000ms and the 2400ms mark. We attribute this behaviour to having OpenWhisk core components installed in one region, which exert some overhead on the workers of the other region when they interact with the platform (e.g., to fetch functions and receive/send requests/notifications). We see similar intervals, although less apparent, for APP and anti-affinity-only-aAPP.

In the 200–1000ms interval, aAPP provides consistent, fast performance, while APP and anti-affinity-only-aAPP show only a few well-performing cases—the rest, on the same performance bracket, are shifted to the right, achieving slower results. We can characterise the "fast" invocations as those where the *divide* and its two *impera* functions appear on a "free" node, i.e., without the *heavy* function, in Europe. Specifically, when using APP, each invocation has a $2/6$ probability of appearing on a free node in Europe, i.e., the probability of fast invocations is $(2/6)^3 \approx 3.7\%$; using anti-affinity-only-aAPP the figure becomes $(1/2)^3 = 12.5\%$ (each invocation has a $1/2$ chance of appearing on a European free node). Finally, using aAPP makes the probability raise to 50%, as all three functions go on the same node (either in the US or in the EU).

Overall, already introducing anti-affinities improves performance (mean, median, tail latency improve resp. of 110%, 10%, and 178%), which shows the impact of sharing a worker with *heavy* functions—APP shows a long tail of invocations after the ca. 3000ms mark in Fig. 5. Looking at worst cases, using aAPP does not result in a considerable performance increase. This is visible from the plot by noticing how the tail high-percentage instances of anti-affinity-only-aAPP and aAPP almost overlap, resulting in a small (+13%) improvement

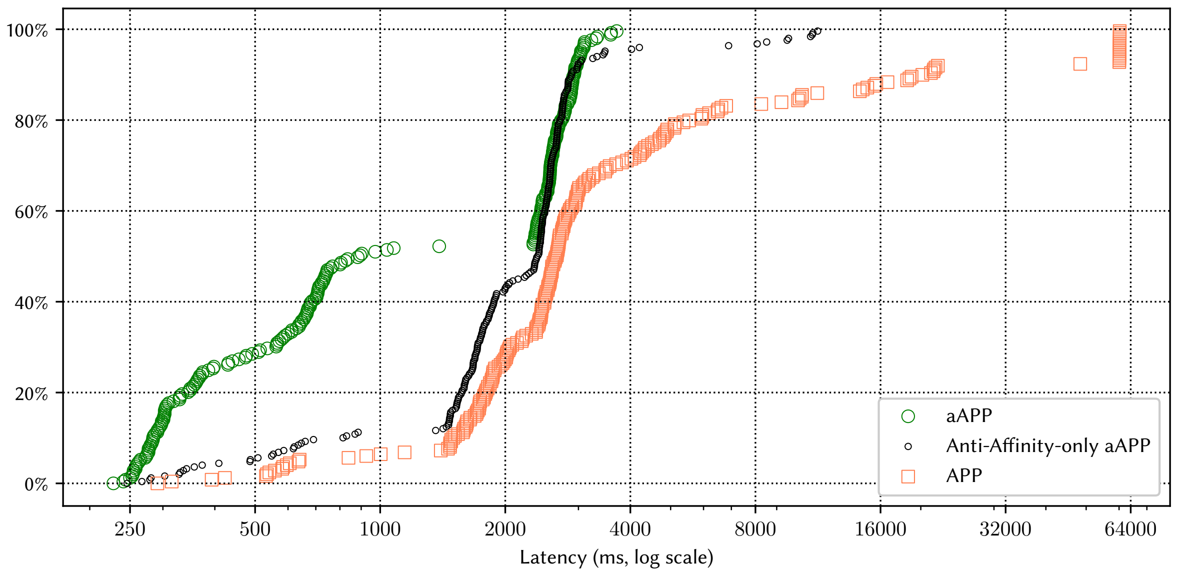
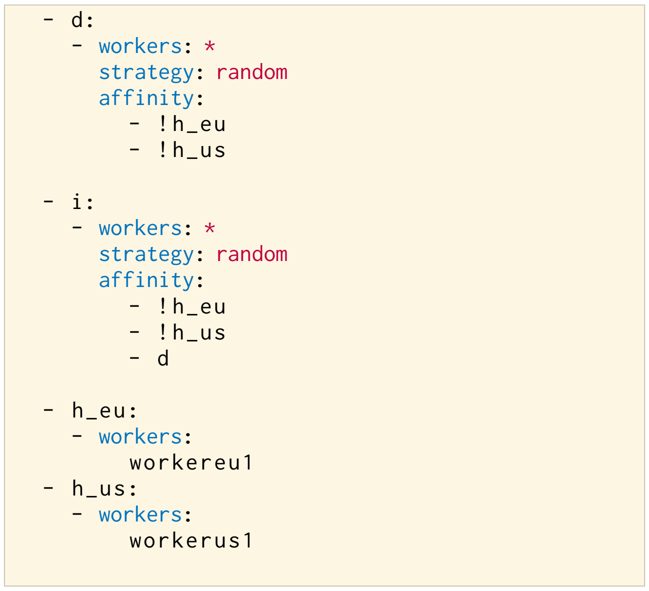


Figure 5: Sorted scatter plot of *divide* functions; x is the latency (ms) of the $y^{\text{th}}\%$ fastest invocation.



```
  - d:
    - workers: *
      strategy: random
      affinity:
        - !h_eu
        - !h_us

  - i:
    - workers: *
      strategy: random
      affinity:
        - !h_eu
        - !h_us
        - d

  - h_eu:
    - workers:
        workereu1
  - h_us:
    - workers:
        workerus1

```

Figure 6: The aAPP script used for the tests.

in tail latency. The differences in mean (+40%) and median (+91%) latency between having affinities or not emerges in the 250–1000ms bracket, where not having affinity causes to have only a few data points w.r.t. to the higher number of fast instances of aAPP. Practically, the figures and distribution show how strongly North American allocations impact latency vs the benefit of co-location. Besides increasing performance, aAPP succeeds in eliminating database access retries, contrarily to anti-affinity-only-aAPP (i.e., 42 requests suffer at least one retry in APP, 23 in anti-affinity-only-aAPP, and 0 in aAPP).

6 aAPP's Overhead is Negligible

While aAPP allows us to exploit (anti-)affinity constraints that would not be expressible otherwise, it is crucial to assess the overhead introduced by the additional functionalities of aAPP. In this section, we show that the added functionalities (to track the state of functions on workers) of our aAPP-based prototype have negligible impact on the platform's performance.

For the experiments, we decided to use the benchmark suite used by De Palma et al. [15] to benchmark their APP-based OpenWhisk implementation. Note that, in our settings, we are not interested in the data locality capabilities of APP but only in checking the scheduling performances of aAPP. Thus, we deploy the platforms in only one cloud zone and use 2000 invocations for each scenario, to simplify as much as possible the testing environment and have enough invocations to draw meaningful comparisons. The benchmarks are:

- *hello-world* implements a simple echo application, and indicates the baseline performance of the platform;
- *long-running* waits for 3 seconds before responding to benchmark the handling of multiple functions running for several seconds and the management of their queueing process;
- *compute-intensive* multiplies two 10^2 square matrices and returns the result to the caller, measuring both the performance of handling functions performing some meaningful computation and of handling large invocation payloads;
- *DB-access (light)* executes a query for a document from a remote MongoDB database. The requested document is lightweight, corresponding to a JSON document of 106 bytes, with little impact on computation. De Palma et al. used the case to measure the impact of data locality on the overall latency. Since we have all workers in the same cloud zone, we use it to measure the overhead of scheduling functions that fetch small payloads from a local database;

	OpenWhisk		APP		aAPP	
	avg	st dev	avg	st dev	avg	st dev
hello-world	0.68	1.16	0.73	1.25	0.8	1.27
long-running	0.48	0.53	0.69	0.92	0.71	1.01
compute-intens.	11.57	11.92	10.17	11.67	10.01	9.66
DB-acc., light	0.65	1.31	0.85	1.62	0.83	1.31
DB-acc., heavy	0.44	0.69	0.91	1.25	1.04	1.7
external service	1.28	2.08	1.95	3.33	1.49	2.5
code dependen.	0.64	1.06	1.0	2.27	0.86	1.8

Figure 7: Comparison of scheduling times between vanilla, APP-, and aAPP-based OpenWhisk (avg and st dev are in ms).

- *DB-access (heavy)* regards both a memory- and bandwidth-heavy data-query function. The function fetches a large document (124.38 MB) from a MongoDB database and extracts a property from the returned JSON. Similarly to the previous function, we use this one to evaluate the overhead of scheduling functions that fetch large payloads from a local database;
- *External service* tracks the performance of invoking an external API (Slack). De Palma et al. drew the function from the Wonderless dataset [19];
- *Code dependencies* is a formatter that takes a JSON string and returns a plain-text one, translating the key-value pairings into Python-compatible dictionary assignments. De Palma et al. drews also this case from the Wonderless dataset [19].

For completeness, we note that we omitted the *cold-start* case from De Palma et al. [15], which is an echo application with sizable, unused dependencies. The peculiarity of the case is its 10-minute invocation pattern, used to track cold-start times (so that the platform evicts cached copies of the function, requiring costly fetch-and-startup times at any subsequent invocation). We omit this test since we can observe its effects with the *hello-world* and *code-dependency* cases.

We run the benchmarks on a one-zone Google Cloud cluster with four Ubuntu 20.04 virtual machines with 4 GB RAM each, one with 2 vCPU for the OpenWhisk controller and three with 1 vCPU, resp. for two workers and a MongoDB instance for the *DB-access* cases. We run 2000 function invocations for each case in batches of 4 parallel requests (500 per thread), recording both the scheduling time (the time between the arrival of a request at the controller and the issuing of the allocation) and the execution latencies. We compare aAPP, APP, and vanilla OpenWhisk. For a fair comparison with vanilla OpenWhisk, we set the APP/aAPP configurations with a `default` policy that falls back to the vanilla scheduler.

For all cases and platforms, we report on Fig. 7 in tabular form the average (avg) and standard deviation (st dev) of the scheduling time. On average, all platforms allocate functions in less than 2ms, except for the *compute-intensive* case, which takes less than 12ms (likely due to the large request payloads that the controller needs to forward to workers). As expected, OpenWhisk vanilla is the fastest, closely (under one millisecond) followed by APP and aAPP—except for the *compute-intensive* case, where APP and aAPP perform better and OpenWhisk is slower by less than 2ms. The differences between APP and aAPP are even smaller, with APP being generally slightly (sub-millisecond) faster than aAPP. To better characterise the comparison, in Fig. 8, we show the plot-line distribution of the scheduling times. The curves exhibit the typical tail distribution pattern [17] of cloud workloads (which accounts for the high standard deviation reported in Fig. 7) and confirm our observations; excluding the tails, they almost overlap with negligible sub-millisecond differences.

7 Related Work and Conclusion

To the best of our knowledge, aAPP is the first language that allows developers to state affinity constraints to better control the scheduling of the functions in FaaS platforms. By extending OpenWhisk, we demonstrate the effectiveness of using (anti-)affinity constraint of aAPP in reducing latency and tail latency. Furthermore, we benchmark that the overhead of supporting aAPP-based affinity constraints is minimal compared to vanilla OpenWhisk and its APP-based variant.

Broadening our scope, the works we see the closest to ours come from the neighbouring area of microservices [18]—the state-of-the-art style alternative to serverless for cloud architectures. Proposals in this direction are by Baarzi and Kesidis [6], who present a framework for the deployment of microservices that infers and assigns affinity and anti-affinity traits to microservices to orient the distribution of resources and microservices replicas on the available machines; Sampaio et al. [39], who introduce an adaptation mechanism for microservice deployment based on microservice affinities (e.g., the more messages microservices exchange the more affine they are) and resource usage; Sheoran et al. [40], who propose an approach that computes procedural affinity of communication among microservices to make placement decisions. Looking at the industry, Azure Service Fabric [27] provides a notion of *service affinity* that ensures that the replicas of a service are placed on the same nodes as those of another, affine service. Another example is Kubernetes, which has a notion of *node affinity* and *inter-pod (anti-)affinity* to express advanced scheduling logic for the optimal distribution of pods [26].

Overall, the mentioned work proves the usefulness of affinity-aware deployments at lower layers than FaaS (e.g., VMs, containers, microservices) and compels a discussion on the interplay between aAPP and IaaS/CaaS-level affinity, which we detail under two main directions. On the one hand, one could realise a version of aAPP for the Infrastructure and/or the Container layers. We argue it is more interesting to focus on FaaS. Indeed, there are mainstream IaaS and

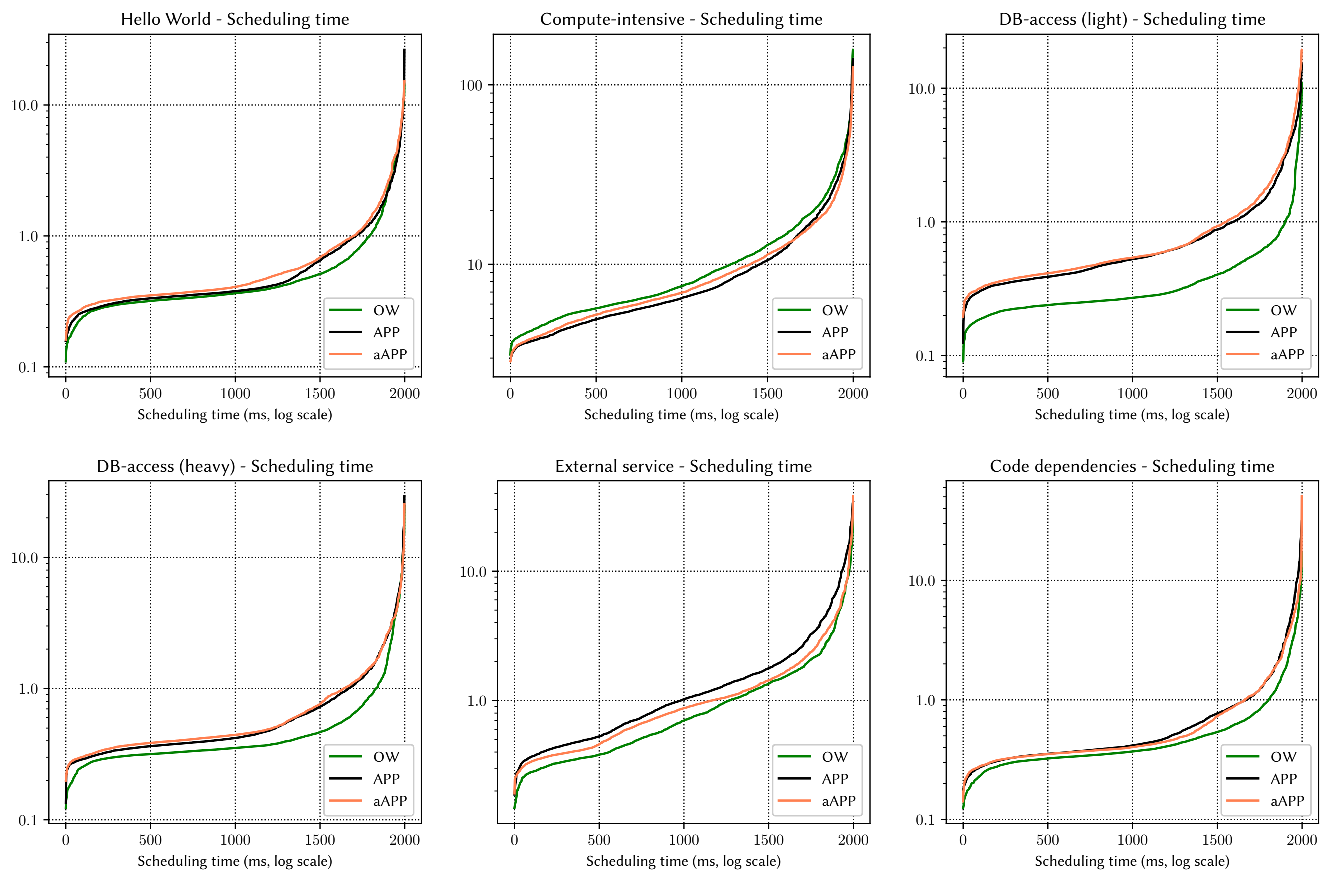


Figure 8: Distribution of scheduling times for vanilla, APP-, and aAPP-based OpenWhisk (x-axis represents instances sorted from the quickest scheduling time to the slowest).

CaaS platforms that allow users to program directly ad-hoc schedulers (e.g., Kubernetes exposes APIs for creating scheduler plugins that define its scheduling policies). Since these layers afford a higher level of customisation than aAPP—at the expense of more technical involvement on the part of the users—a variant of aAPP for the IaaS/CaaS-levels seems less useful. On the other hand, one can use IaaS and CaaS platforms that support affinity constraints to implement affinity-aware FaaS platforms. We see two main problems with pursuing this path. The first regards performance. To implement FaaS-level affinity using IaaS/CaaS affinity constraints, we need to impose a 1:1 relation between a function instance and the VM/container running it (if we let the same VM/container run parallel copies of the same function we cannot guarantee e.g., self anti-affinity). A consequence of such an implementation is precluding the platform from exploiting the ubiquitous serverless optimisation technique of VM/container reuse to avoid cold starts [30, 42, 43]. The second problem regards abstraction leakage, where letting FaaS users access the underlying IaaS/CaaS layers leaks details and control of the infrastructural components and breaks FaaS' paradigmatic abstractions.

A recent trend of FaaS is the definition/handling of the composition/workflows of functions, like AWS step-functions [5] and Azure Durable functions [10]. The main idea behind these works is to allow users to define workflows as the composition of functions with their branching logic, parallel execution, and error handling. The orchestrator/controller of the platform then uses the workflow to manage function executions and handle retries, timeouts, and errors. Our proposal is orthogonal to these works. Indeed, assuming the workflow is available, the orchestrator developed for handling serverless workflows should be extensible with an aAPP-like script to specify where to schedule the functions within a given workflow. Future work on this integration would support the enforcement of even more expressive policies than aAPP, like preventing function instances of the same workflow from sharing nodes.

Another interesting proposal, Palette [1], uses optional opaque parameters in function invocations to inform the load balancer of Azure Functions on the affinity with previous invocations and the data they produced. While Palette does not support (anti-)affinity constraints, it allows users to express which invocations benefit from running on the same node. We deem an interesting future work extending aAPP to support a notion of (anti-)affinity that considers the history of scheduled functions.

Regarding the constructs we have proposed for expressing affinity-aware policies in aAPP, we observe that an alternative approach could be to let the user directly declare the properties to enforce, leaving to the platform the task to realise them at run time. The scheduling runtime of this APP variant would allocate a function only if the allocation satisfies the formula or fail otherwise. The limitation of this approach lies in its scalability. Verifying the satisfiability of a property could require assessing multiple interacting constraints, possibly leading to an exponential time complexity with respect to the formula's size, the number of workers, and the number of functions involved.[3] Contrarily, the aAPP scheduler checks whether it can allocate a function on a worker in linear time on the size of the workers and aAPP script length.

Implementation-wise, OpenWhisk supports scenarios where multiple controllers share the pool of available workers (e.g., for redundancy and load balancing) and take scheduling decisions without coordination. In our aAPP-based implementation, such multi-controller configurations present a problem since we need to prevent scheduling races among controllers—e.g., imagine two controllers that select an available, empty worker and, at the same time, allocate mutually anti-affine functions on it. Supporting multi-controller deployments is important, but we deem dealing with it is outside the scope of this paper and an interesting subject for future work.

Finally, while our evaluation demonstrates that affinity and anti-affinity constraints can enhance serverless application performance, a standardised benchmark of real-world examples would enable a more comprehensive analysis. Unfortunately, to the best of our knowledge, no such benchmark currently exists, even across other cloud abstraction layers (e.g., Kubernetes applications with affinity and anti-affinity configurations). As a direction for future work, we plan to collaborate with the community to collect instances of real-world applications and their affinity constraints, creating a benchmark to compare scheduler performances in serverless platforms.

References

[1] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose M. Faleiro, Gohar Irfan Chaudhry, Iñigo Goiri, Ricardo Bianchini, Daniel S. Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *EuroSys*. ACM, 365–380. https://doi.org/10.1145/3552326.3567496

[2] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 118:1–118:26. https://doi.org/10.1145/3276488

[3] Anonymous. 2024. Link omitted for blind review purposes.

[4] AWS. 2024. AWS Lambda. https://aws.amazon.com/lambda/.

[5] AWS. 2024. AWS Step Functions. https://aws.amazon.com/step-functions/.

[6] Ataollah Fatahi Baarzi and George Kesidis. 2021. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 427–441.

[7] Peter Bailis and Ali Ghodsi. 2013. Eventual consistency today: Limitations, extensions, and beyond. *Commun. ACM* 56, 5 (2013), 55–63.

[8] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*. Springer, 1–20.

[9] Ali Banaei and Mohsen Sharifi. 2022. ETAS: predictive scheduling of functions on worker nodes of Apache OpenWhisk platform. *J. Supercomput.* 78, 4 (2022), 5358–5393. https://doi.org/10.1007/S11227-021-04057-Z

[10] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. 2021. Durable functions: semantics for stateful serverless. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[3]For example, if the language allows encoding properties such as "schedule the function f only if it does not prevent scheduling higher priority functions $g_1,...,g_n$" then, due to the NP-hardness of bin-packing [38], the problem of scheduling a function becomes an NP-hard problem.

[11] 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. 2020. Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems* 35, 1 (2020), 77–87.

[12] Google Cloud. 2024. Google Cloud Functions. https://cloud.google.com/functions/.

[13] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. 2020. Valve: Securing function workflows on serverless computing platforms. In *Proceedings of The Web Conference 2020*. 939–950.

[14] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. 2022. A Declarative Approach to Topology-Aware Serverless Function-Execution Scheduling. In *IEEE International Conference on Web Services, ICWS*. IEEE, 337–342. https://doi.org/10.1109/ICWS55610.2022.00056

[15] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. 2024. An OpenWhisk Extension for Topology-Aware Allocation Priority Policies. In *Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14676)*, Ilaria Castellani and Francesco Tiezzi (Eds.). Springer, 201–218. https://doi.org/10.1007/978-3-031-62697-5_11

[16] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. 2020. Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation. In *ICSOC (Lecture Notes in Computer Science, Vol. 12571)*. Springer, 416–430. https://doi.org/10.1007/978-3-030-65310-1_29

[17] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[18] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216. https://doi.org/10.1007/978-3-319-67425-4_12

[19] Nafise Eskandani and Guido Salvaneschi. 2021. The Wonderless Dataset for Serverless Computing. In *MSR*. IEEE, 565–569. https://doi.org/10.1109/MSR52588.2021.00075

[20] Fission. 2024. Fission. https://fission.io/.

[21] Scott Hendrickson, Stephen Sturdevant, Edward Oakes, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. *login Usenix Mag.* 41, 4. https://www.usenix.org/publications/login/winter2016/hendrickson

[22] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *SOSP*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 691–707. https://doi.org/10.1145/3477132.3483541

[23] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. University of California.

[24] Daniel Kelly, Frank Glavin, and Enda Barrett. 2020. Serverless computing: Behind the scenes of major platforms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 304–312.

[25] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *Proc. of USENIX ATC*. USENIX Association, 805–820.

[26] Kubernetes. 2024. Node Affinity. https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/.

[27] Microsoft. 2024. Azure Service Fabric. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-overview.

[28] Microsoft. 2024. Microsoft Azure Functions. https://azure.microsoft.com/.

[29] Microsoft. 2024. Service affinity in Service Fabric. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-cluster-resource-manager-advanced-placement-rules-affinity.

[30] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *HotCloud*. USENIX Association. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[31] OpenFaaS. 2024. OpenFaaS. https://www.openfaas.com/.

[32] OpenStack. 2024. Documentation. https://docs.openstack.org/project-deploy-guide/openstack-ansible/ocata/app-advanced-config-affinity.html.

[33] Apache OpenWhisk. 2024. Apache OpenWhisk. https://openwhisk.apache.org/.

[34] Paul Osman. 2018. *Microservices Development Cookbook: Design and build independently deployable, modular services*. Packt Publishing.

[35] Giuseppe De Palma et al. 2023. Serverless Scheduling Policies based on Cost Analysis. In *TiCSA@ETAPS 2023 (EPTCS, Vol. 392)*. 40–52. https://doi.org/10.4204/EPTCS.392.3

[36] Giuseppe De Palma et al. 2024. Towards a Function-as-a-Service Choreographic Programming Language: Examples and Applications. arXiv:2406.09099 [cs.PL] https://arxiv.org/abs/2406.09099

[37] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. 2024. On the Complexity of Reachability Properties in Serverless Function Scheduling. *CoRR* abs/2407.14159 (2024). https://doi.org/10.48550/ARXIV.2407.14159 arXiv:2407.14159

[38] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

[39] Adalberto R Sampaio, Julia Rubin, Ivan Beschastnikh, and Nelson S Rosa. 2019. Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications* 10, 1 (2019), 1–30.

[40] Amit Sheoran, Sonia Fahmy, Puneet Sharma, and Navin Modi. 2021. Invenio: Communication Affinity Computation for Low-Latency Microservices. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems.* 88–101.

[41] Christopher Peter Smith, Anshul Jindal, Mohak Chadha, Michael Gerndt, and Shajulin Benedict. 2022. Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In *ICFEC.* IEEE, 17–25.

[42] Khondokar Solaiman and Muhammad Abdullah Adnan. 2020. WLEC: A Not So Cold Architecture to Mitigate Cold Start Problem in Serverless Computing. In *IC2E.* IEEE, 144–153. https://doi.org/10.1109/IC2E48712.2020.00022

[43] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[44] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.

[45] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18).* 133–146.

[46] YAML. 2022. YAML Specification. https://yaml.org/spec/.