An OpenWhisk Extension for Topology-Aware Allocation Priority Policies

Giuseppe De Palma[1,2(✉)], Saverio Giallorenzo[1,2], Jacopo Mauro[3], Matteo Trentin[1,2,3], and Gianluigi Zavattaro[1,2]

[1] Università di Bologna, Bologna, Italy
{giuseppe.depalma2,saverio.giallorenzo2,matteo.trentin2,
gianluigi.zavattaro}@unibo.it
[2] OLAS Research Team, INRIA, Sophia Antipolis, France
[3] University of Southern Denmark, Odense, Denmark
mauro@imada.sdu.dk

Abstract. The Topology-aware Allocation Priority Policies (tAPP) language allows users of serverless platforms to orient the scheduling of their functions w.r.t. the topological properties of the available computation nodes. A tAPP-based platform can support multiple scheduling policies, which one would usually enforce via (brittle) ad-hoc multi-instance platform deployments.

In this paper, we present an extension of the Apache OpenWhisk serverless platform that supports tAPP-based scripts. We show that our extension does not negatively impact the performance of generic, non-topology-bound serverless scenarios, while it increases the performance of topology-bound ones.

1 Introduction

Function-as-a-Service (FaaS) is a serverless cloud computing model where users deploy architectures of stateless functions and a platform handles all system operations [14].

While the FaaS model abstracts away infrastructural details, informing the scheduler on important infrastructural traits can improve the performance of serverless architectures. Indeed, function execution performance can depend on which computing resource, also called *worker*, the function runs. Effects like *data locality* [12]—related to data-access latencies—or *session locality* [12]—due to the overhead of establishing connections to other services—can negatively impact the run time of functions.

This work has been partially supported by the research project FREEDA (CUP: I53D23003550006) funded by the framework PRIN 2022 (MUR, Italy), RTM&R (CUP: J33C22001170001) funded by the MUR National Recovery and Resilience Plan (European Union - NextGenerationEU) and the French ANR project SmartCloud ANR-23-CE25-0012.

Published by Springer Nature Switzerland AG 2024
I. Castellani and F. Tiezzi (Eds.): COORDINATION 2024, LNCS 14676, pp. 201–218, 2024.
https://doi.org/10.1007/978-3-031-62697-5_11

The tAPP language [9] (briefly introduced in Sect. 3) allows users to declaratively express minimal infrastructural constraints to orient function scheduling.

In previous work [9] we presented a serverless platform that supports tAPP-specified scheduling policies. In this tool paper, we describe and evaluate the performance of our platform, which builds upon the widely adopted open-source serverless platform Apache OpenWhisk, extended to support tAPP—c.f. Section 4. In particular, the extension regarded the introduction of new components—e.g., a *watcher* service, which informs the gateway and the controllers on the current status of the nodes of the platform—and the extension of existing ones with new functionalities—e.g., to capture topological information at the level of workers and controllers, to enable live-reloading of tAPP policies, to let controllers and gateways follow tAPP policies depending on topological zones, etc.

The main contribution is in Sect. 5, where we validate our implementation through a set of benchmarks that include both generic and data-locality-bound serverless architectures, comparing the performance of vanilla OpenWhisk and our prototype. In particular, we collected a set of representative serverless test applications, divided into ad-hoc and real-world ones. Ad-hoc tests stress specific issues of serverless platforms. Real-world tests are functions taken from publicly available, open-source repositories of serverless applications used in production and selected from the Wonderless [10] serverless benchmark dataset. We show that our prototype does not exert a noticeable overhead over generic benchmarks while it substantially improves the performance of locality-bound ones (paired with dedicated tAPP scripts). A video that showcases our platform is available at https://vimeo.com/915098870.

2 Background

We dedicate this section to explaining background knowledge for readers unfamiliar with serverless and the OpenWhisk FaaS platform in particular.

Serverless Function Scheduling. The serverless development cycle is divided in two main parts: *a*) the writing of a function using a programming language supported by the platform (e.g., JavaScript, Python, C#) and *b*) the definition of an event that should trigger the execution of the function. For example, an event is a request to store some data, which triggers a process managing the selection, instantiation, scaling, deployment, fault tolerance, monitoring, and logging of the functions linked to that event. A Serverless provider schedules functions on its workers, controlling the scaling of the architecture by adjusting its available resources and billing its users on a per-execution basis. When instantiating a function, the provider has to create the appropriate execution environment for the function. Containers [8] and Virtual Machines (VM) [6] are the main technologies used to implement isolated execution environments for functions. If the provider allocates a new container/VM for every request, the initialisation overhead of the container would negatively affect both the performance of the single

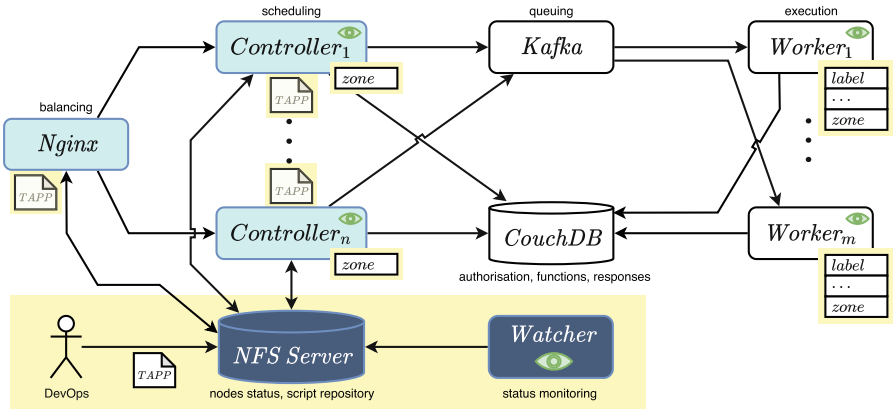

Fig. 1. Architectural view of our OpenWhisk extension. The existing OpenWhisk components we modified are in light blue while the new ones are in yellow (Color figure online).

function and heavily increase the load on the worker. A solution to tackle this problem is to maintain a "warm" pool of already-allocated containers. This matter is usually referred to as *code locality* [12]. Resource allocation also includes I/O operations that need to be properly considered and can avoid bad allocations over I/O-bound devices following the principle of *session locality* [12], i.e., taking advantage of already established user connections to workers. Another important aspect to consider to schedule functions is that of *data locality*, which comes into play when functions need to intensively access (connection- or payload-wise) some data storage (e.g., databases or message queues). Intuitively, a function that needs to access some data storage and that runs on a worker with high-latency access to that storage (e.g., due to physical distance or thin bandwidth) is more likely to undergo heavier latencies than if run on a worker "closer" to it.

Apache OpenWhisk. To build our tAPP prototype we concentrate on the widely-adopted open-source project Apache OpenWhisk. While we focus on the architecture of OpenWhisk, serverless platforms share common architectural patterns [11], making this contribution useful also as a guideline for alternative serverless platforms.

The upper part of Fig. 1 above the yellow box reports the architectural view of OpenWhisk.[1] From the left, *Nginx* is the entry point and load balancer of the system and distributes the incoming requests to the *Controller*s. Controllers then decide on which of the available computation nodes, called *Worker*s (or "invokers" in OpenWhisk's parlance) to schedule the execution of a given function. *Controller*s allocate functions on Workers following a hard-coded policy that

[1] For space reason, Fig. 1 shows also the elements we modified and added to support tAPP—these will be detailed in Sect. 4.

allocates requests to the same function on the same list of *Worker*s. The principle behind this policy is caching functions on workers to reduce cold starts—the downtime due to fetching the code and loading the runtime of functions. Finally, we have Apache *Kafka* [16] and *CouchDB* [5]; the first handles the routing and queueing of requests, while CouchDB manages authorisation and the storage of functions and their responses.

3 The tAPP Language

We can now introduce the tAPP language, presenting its syntax and semantics.

We report the syntax of tAPP in Fig. 2, which is compliant with YAML [18]. The basic entities of the language are *a*) scheduling policies, defined by a *policy tag* identifier to which users can associate their functions—the policy-function association is a one-to-many relation—and *b*) workers, identified by a *worker label*—where a label identifies a collection of computation nodes. All identifiers are strings. Given a tag, the corresponding policy includes a list of blocks, possibly closed with `strategy` and `followup` options. A block includes four parameters: an optional `controller` selector, a collection of `workers`, a possible scheduling `strategy`, and an `invalidate` condition. The outer `strategy` defines the policy we must follow to select among the blocks of the tag, while the inner `strategy` defines how to select workers from the items specified within a chosen `workers` block. The `controller` defines the identifier of a specific controller we want the gateway to redirect the invocation request to. When used, it is possible to define a `topology_tolerance` option to further refine how tAPP handles failures (of controllers). The collection of `worker`s can be either a list of labels pointing to specific workers (`wrk`), or a worker `set`. In lists, the user can specify the `invalidate` condition of each single worker, while in sets, the `invalidate` condition applies to all the workers included in the set. When users specify an `invalidate` condition at block level, this is directly applied to all `workers` items (`wrk` and `set`) that do not define one. In `set`s the user can also specify a `strategy` followed to choose workers within the set. Finally, the `followup` value defines the behaviour to take in case no specified controller or worker in a tag is available to handle the invocation request.

We discuss the tAPP semantics, and the possible parameters, by commenting on the comprehensive script shown in Fig. 3. The tAPP script starts with the tag `default`, which is a special tag used to specify the policy for non-tagged functions, or to be adopted when a tagged policy has all its members invalidated, and the `followup` option is `default`. In Fig. 3, the `default` tag describes the default behaviour of the serverless platform running tAPP. In this case, we use a `workers set` to select workers, with no value specified for `set` which represents all worker labels. The `strategy` selected is the `platform` default. In our prototype in Sect. 4 the `platform` strategy corresponds to a selection algorithm that mediates load balancing and code locality by associating a function to a numeric hash and a step size—a number that is co-prime of and smaller than the number of workers. The `invalidate` strategy considers a worker non-usable when it is `overload`ed, i.e., it does not have enough resources to run the function.

```
app        ::= - tag
tag        ::= policy_tag : - controller? workers strategy? invalidate? strategy? followup?
controller ::= controller : label ( topology_tolerance : ( all — same — none ) )?
workers    ::= workers: - wrk : label invalidate?
             | workers: - set : label? strategy? invalidate?
strategy   ::= strategy : ( random | platform | best_first )
invalidate ::= invalidate : ( capacity_used n% | max_concurrent_invocations n | overload )
followup   ::= followup : ( default | fail )
```

Fig. 2. The syntax of tAPP.

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

```
- couchdb_query:
  - workers:
    - wrk: DB_worker1
    - wrk: DB_worker2
    strategy: random
    invalidate: capacity_used 50%
```

```
- workers:
  - wrk: near_DB_worker1
  - wrk: near_DB_worker2
  strategy: best_first
  invalidate: ↴
    max_concurrent_invocations 100
followup: fail
```

Fig. 3. Example of a tAPP script.

Besides the `default` tag, the `couchdb_query` tag is used for those functions that access the database. The scheduler considers worker blocks in order of appearance from top to bottom. As mentioned above, in the first block (associated to `DB_worker1` and `DB_worker2`) the scheduler randomly picks one of the two worker labels and considers the corresponding worker invalid when it reaches the `50%` of capacity. Here, the notion of capacity depends on the implementation (e.g., our OpenWhisk-based tAPP implementation in Sect. 4 uses information on the CPU usage to determine the load of invokers). When both worker labels are invalid, the scheduler goes to the next `workers` block, with `near_DB_worker1` and `near_DB_worker2`, chosen following a `best_first` strategy—where the scheduler considers the ordering of the list of `workers`, sending invocations to the first until it becomes invalid, to then pass to the next ones in order. The `invalidate` strategy of the block (applied to the single `wrk`) regards the maximal number of concurrent invocations over the labelled worker—`max_concurrent_invocations`, which is set to `100`. If all the worker labels are invalid, the scheduler applies the `followup` behaviour, which is to `fail`. Users can define subsets of workers by specifying a label associated with the workers, e.g., `local` selects only those workers associated to the *local* label. The scheduling on worker-`set`s follows the same logic of block-level worker selection: it exhausts all workers before deeming the item invalid. Since worker-set selection/invalidation policies are distinct from block-level ones, we let users define the `strategy` and `invalidate` policies to select the worker in the set. For example, we can pair the above selection with a `strategy` and an `invalidate` options, e.g.,

```
– workers: - set: local strategy: random invalidate: capacity_used 50%
```

which tells the scheduler to adopt the `random` selection strategy and the `capacity_used` invalidation policy when selecting the workers in the *local* set. When worker-sets omit the definition of the selection `strategy` we consider the default one. When the invalidation option is omitted, we either apply that of the enclosing block or, if that is also missing, the default one. Summarising, given a policy tag, the scheduler follows the `strategy` (option) to select the corresponding blocks. A block includes three clauses.

The `workers` clause either contains a non-empty list of worker (`wrk`) labels, each paired with an optional invalidation condition, or a worker-`set` label (possibly blank, to select all workers) to range over sets of workers; workers `set`s optionally define the `strategy` and `invalidate` options to select workers within the set and declare them invalid.

The `strategy` clause defines the policy of item selection at the levels of *policy_tag*, *workers* block, and workers `set`s. tAPP supports three strategies: `random`, which selects items in a fair random manner; `best_first`, which selects items following their order of appearance; and `platform`, which selects items following the default strategy of the serverless platform—in our prototype, this corresponds to a co-prime-based selection.

The `invalidate` clause specifies when a worker (label) cannot host the execution of a function. When all labels in a block are invalid, we follow the defined `strategy` to select the next block until we either find a valid worker or exhaust all blocks. In the latter case, we apply the `followup` behaviour. Current `invalidate` options are: `overload`, where the worker lacks enough computational resources to run the function; `capacity_used`, where the worker reached a threshold percentage of CPU load; and `max_concurrent_invocations`, where the worker reached a threshold number of concurrent invocations. All invalidate options include the non-reachability of a worker.

Within a policy, the `followup` clause specifies what to do when all the blocks in a policy tag are invalid; either: `fail`, which drops the scheduling of the function; and `default`, which applies the `default` policy. Since the `default` block is the only possible "backup" tag used when all workers of a custom tag cannot execute a function (because they are all invalid), the `followup` value of the `default` tag is always set to `fail`.

To further detail the topological constraints of function execution scheduling, we have the *controller*. This is an optional, block-level clause that identifies which `controller`s in the current deployment we want to target to execute the scheduling policy of the current tag. Similarly to workers, we identify controllers with a label.

Users can label controllers and workers with the topological *zone* where they belong. When the designated `controller` is unavailable, tAPP can use this topological information to try to satisfy the scheduling request by forwarding it to some alternative controller. Indeed, a *controller* can have a `topology_tolerance` parameter, which specifies what workers an alternative controller can use. Specifically, `all` is the default and most permissive option and does not restrict the topology zone of workers; `same` constrains the function to run on workers in

the same zone of the faulty controller (e.g., for data locality); `none` forbids the forward to other controllers.

```
- couchdb_query:
  - controller: DBZoneCtl
    workers:
     - set: local
       strategy: random
    topology_tolerance: same
  followup: default
```

As an example, we could use the topology zones and rewrite the previous tAPP script from Fig. 3 for the `couchdb_query` tag as shown on the left. In this way, we guarantee that the function will be executed always on the workers in the same zone of the database. Lastly, tAPP lets users express a selection strategy for policy blocks; as represented by the optional *strategy* fragment of the *tag* rule in tAPP's syntax. By default, when we omit to define a *strategy* policy for blocks, tAPP allocates functions following the blocks from top to bottom—i.e., `best_first` is the default policy. Here, for example, setting the `strategy` to `random` captures the simple load-balancing strategy of using randomness to uniformly distributing requests among the available controllers.

4 Supporting tAPP in OpenWhisk

We now discuss how we modified and extended OpenWhisk to support tAPP policies. In the following, we pair OpenWhisk with the popular and widely supported container orchestrator Kubernetes to orchestrate the deployment of the components.

Figure 1 depicts the architecture of our OpenWhisk extension, where we reuse the *Workers* and the *Kafka* components, we modify *Nginx* and the *Controllers* (light blue in the picture), and we introduce two new services: the *Watcher* and the *NFS Server* (in the highlighted area of Fig. 1). The modifications mainly regard letting Nginx and Controllers retrieve and interpret both tAPP scripts and data on the status of nodes, to forward requests to the selected controllers and workers. Concerning the new services, the Watcher monitors the topology of the Kubernetes cluster and collects its current status into the NFS Server, which provides access to tAPP scripts and the collected data to the other components. Below, we present the two new services, the changes to the existing OpenWhisk components, and how the proposed system supports live-reloading of tAPP configurations. We conclude with a description of the deployment procedure of the resulting prototype.

OpenWhisk Controller. To let the original OpenWhisk controller execute tAPP scripts, we extended the existing codebase of OpenWhisk. The component is written in Scala and it consists of a base `LoadBalancer` class which the vanilla OpenWhisk load balancer extends. To let OpenWhisk support tAPP scheduling policies, we introduced a new class that also extends the base one, called `ConfigurableLoadBalancer`. This class implements a parser and an engine that interprets tAPP scripts.

Watcher and NFS Server Services. We introduce the *Watcher* service to map tAPP-level information, such as zones and controllers/workers labels, to deployment-specific information, e.g., the name Kubernetes uses to identify computation nodes.

To realise the Watcher, we rely on the APIs provided by Kubernetes, which we use to deploy our OpenWhisk variant. In Kubernetes, applications are collections of services deployed as "pods", i.e., a group of one or more containers that must be placed on the same node and share network and storage resources. Kubernetes automates the deployment, management, and scaling of pods on a distributed cluster and one can use its API to monitor and manipulate the state of the cluster.

Our Watcher polls the Kubernetes API, asking for pod names and the respective *label*s and *zone*s of the nodes (cf. Fig. 1), and stores the mapping into the *NFS Server*.

As shown in Fig. 1, Nginx uses the output of the Watcher to forward requests to controllers, allowing tAPP scripts to target controllers through their label rather than their specific pod identifier. Besides abstracting away deployment details, this feature supports dynamic changes to the deployment topology, e.g., when Kubernetes decides to move a controller pod at runtime on another node.

Moreover, the *NFS Server* works as the main injection point for tAPP scripts, both right after deployment and during the execution of the platform. When a new script is available on the *NFS Server*, the Controllers and Nginx obtain a copy, avoiding possible latencies due to fetching at function invocation. Future refinements can include the implementation of a dedicated API for the injection of scripts, removing the need for the *NFS Server*.

Nginx, OpenWhisk's Entry Point. Nginx forwards requests to all available controllers, following a hard-coded round-robin policy. To support tAPP, we change how Nginx processes incoming requests of function execution. We used *njs* (a subset of the JavaScript language that Nginx provides to extend its functionalities) for this integration.

Namely, we wrote an njs plug-in to analyse all requests passing through Nginx. The plug-in extracts any tag from the request parameters and compares it against the tAPP scripts. If the extracted tag matches a policy-tag, we interpret the associated policy, resolve its constraints, and find the related node label. The last step is translating the label into a pod name, done using the label-pod mapping produced by the Watcher service. Since Nginx manages all inbound traffic, we strived to keep the footprint of the plug-in small, e.g., we only interpret tAPP scripts and load the mappings when requests carry some tags and we use caching to limit retrieval downtimes from the NFS Server. From the user's point of view, the only visible change regards the tagging of requests. When tags are absent, Nginx follows the `default` policy or, when no tAPP script is provided, it falls back to the built-in round-robin.

Topology-Based Worker Distribution. We associate labels with pods via the topology labels provided by Kubernetes. These labels are names assigned to

nodes that can describe the structure of the cluster by annotating their zones and attributes. In Fig. 1, we draw labels as boxes on the side of the controllers and workers.

Since OpenWhisk does not have a notion of topology, all controllers can schedule all functions on any available worker. Our extension unlocks a new design space that administrators can use to fine-tune how controllers access workers, based on their topology. At deployment, DevOps define the access policy used by all controllers. Our investigation led us to identify four topological-deployment access policies.

The *default* policy is the original one of OpenWhisk, where controllers have access to a fraction of all workers' resources. This policy has two drawbacks. First, it tends to *overload* workers, since controllers race to access workers in an uncoordinated manner. Second, it gives way to a form of *resource grabbing*, since controllers can access workers outside their zone, effectively taking resources away from "local" controllers.

The *min_memory* policy is a refinement of the *default* policy and it mitigates overload and resource-grabbing by assigning only a minimal fraction of the worker' resources to "foreign" controllers. For example, in OpenWhisk the resources regard the available memory for one invocation (in OpenWhisk, 256 MB). When workers have no controller in their topological zone or no topological zone at all, we follow the default policy. The *min_memory* policy has a drawback too: it can lead to scenarios where smaller zones quickly become saturated and unable to handle requests.

The *isolated* policy lets controllers access only co-located workers. This reduces overloading and resource grabbing but accentuates small-zone saturation effects.

The *shared* policy accesses primarily local workers and lets them access foreign ones after having exhausted the locals. This policy mediates between partitioning resources and efficient usage, but it can suffer from resource grabbing by remote controllers.

Controllers follow the policies declared in the available tAPP scripts and access topological information and tAPP scripts in the same way as described for Nginx. If no tAPP script is available, controllers resort to their original, hard-coded logic but prioritise scheduling functions on co-located workers.

Since the cluster's topology, its attributes, and the related tAPP scripts can change (e.g., to include a new node), we designed our prototype to dynamically support such changes, avoiding stop-and-restart downtimes. We implement this feature by storing a single global copy of the policies in the NFS Server, while we keep multiple, local copies in Nginx and each controller instance. When we update the reference copy, we notify Nginx and the controllers and let them handle cache invalidation and retrieval.

Deploying tAPP *-based OpenWhisk* The standard way to deploy OpenWhisk is by using the Docker images available for each component of the architecture—this lets developers choose the configuration that suits their deployment scenario, spanning single-machine deployments, where all the components run on the same

node, and clustered (e.g., via Kubernetes) deployments, e.g., assigning a different node to each component. Since we modified the Controller component of the architecture, we built a new, dedicated Docker image so that it is generally available to be used in place of the vanilla controller.

5 Evaluation

We evaluate our prototype by comparing the performance of vanilla OpenWhisk and our tAPP-based variant under different benchmarks. We show that the overhead of running tAPP scheduling policies is negligible and that, in locality-bound scenarios, custom scheduling policies reduce function run times.

To obtain our empirical results, we devise two kinds of benchmarks. The first kind of benchmarks measures the overhead of the advanced features introduced by our prototype against the performance of vanilla OpenWhisk. The purpose of these "overhead" tests is to empirically quantify the impact of performance of the advanced, dynamic features introduced by tAPP in our prototype w.r.t. vanilla OpenWhisk. Since we want to focus on the performance of the platform, rather than the execution of the functions, we avoid tests that can introduce biases generated by data locality effects, i.e., those coming from the vanilla OpenWhisk accidentally choosing workers with a high-latency access to some data sources. The second kind of benchmarks focuses on "data-locality" effects and benchmarks the performance gain of topology-aware policies. The idea of these tests is to evaluate the performance gains that tAPP-based policies can provide, compared against the possible suboptimal scheduling of the vanilla version.

In what follows, we mark (**O**) overhead tests and (**D**) data-locality ones.

To perform a comprehensive comparison, we collected a set of representative serverless test applications, divided into ad-hoc and real-world ones. Ad-hoc tests stress specific issues of serverless platforms. Real-world tests are functions taken from publicly available, open-source repositories of serverless applications used in production and selected from the Wonderless [10] serverless benchmark dataset.

Ad-hoc Tests. Each ad-hoc test focuses on specific a trait: **hellojs** (**O**) implements a "Hello World" application and it provides an indication of the performance functions with a simple behaviour which parses and evaluates some parameters and returns a string; **sleep** (**O**) waits 3 s and benchmarks the handling of multiple functions running for several seconds and the management of their queueing process; **matrixMult** (**O**) multiplies two 100×100 matrices and returns the result to the caller, to measure the performance of handling functions performing some meaningful computation; **cold-start** (**O**) is a parameterless variant of **hellojs** that loads a heavy set of dependencies (42.8 MB) required and instantiated when the function starts; **mongoDB** (**D**) stresses the effect of data locality by executing a query requiring 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. This test focuses on the

performance of accessing delocalized data; **data-locality** (**D**) encompasses both a memory- and bandwidth-heavy data-query function. It requests a large document (124.38 MB) from a MongoDB database and extracts a property from the returned JSON. This test witnesses both the impact of data locality w.r.t latency and bandwidth occupation.

All tests use Node.js 10 except those using MongoDB (v5), which use Node.js 12.

Real-World Tests. We draw our real-world tests from Wonderless [10]; a peer-reviewed dataset with almost 2000 projects automatically scraped from GitHub. The projects target serverless platforms like AWS, Azure, Cloudflare, Google, and OpenWhisk.

In a way, Wonderless reflects the current situation of serverless industrial adoption. The distribution of its projects is heavily skewed towards AWS-specific applications. Indeed, out of the 1877 repositories in the dataset, 97.8% are AWS-specific. Since we need the projects to work on OpenWhisk, we exclude most of them, leaving us with 66 projects which, unfortunately, sometimes carry limited information on their purpose and usage, they implement "Hello Word" applications, and have deployment problems. Thus, to select our real-world tests, we followed these exclusion criteria: *a*) the project must have a `README.md` file written in English with at least a simple description of the project's purpose. This filters out repositories that contain no explanation on their inner workings or a description of the project; *b*)the project works as-is, i.e., no compilation or execution errors; are thrown when deployed and the only modifications allowed for its execution regard configuration and environment files (i.e., API keys, credentials, and certificates). The reason for this rule concerns both the validity and reliability of the dataset, since fixing execution bugs could introduce biases from the researchers and skew the representativeness of the sample; *c*) the project must not use paid services (e.g., storage on AWS S3 or deployment dependent on Google Cloud Functions), which guarantees that the tests are generally available and easily reproducible; *d*) the project must represent a realistic use case. These exclude "Hello World" examples and boilerplate setups. The project must implement at least a function accepting input and producing an output as a result of either an internal transformation (such as code formatting or the calculation of a complex mathematical expression) or the interaction with an external service. This rule filters out all projects which do not represent concrete use cases.

The filtering led to the selection of three real-world tests:[2] **slackpost** (**O**), from bespinian/k8s-faas-comparison, is a project written in Javascript, run on Node.js 12, and available for different platforms. It consists of a function that sends a message through the Slack API. While not complex, it is a common example of a serverless application that acts as the endpoint for a Slack Bot; **pycatj** (**O**), from hellt/pycatj-web, is a project written in Python, run on Python 3.7, and it requires pre-packaged code to work. It consists of a formatter that takes

[2] For reproducibility, we provide the list of the rejection criteria applied to all 63 non-AWS discarded projects at [1].

an incoming JSON string and returns a plain-text one, where key-value pairings are translated in Python-compatible dictionary assignments. As a sporadically invoked web-based function, it represents an ordinary use case for serverless; **terrain** (**D**), from terraindata/terrain, is a project written in Javascript and run on Node.js 12. The repository contains a serverless application that stress-tests a deployed backend. The backend is a traditional, non-serverless application deployed on a separate machine from the test cluster, which works as the target for this stress test. This is a common example of a serverless use case: monitoring and benchmarking external systems.

5.1 Test Environment

We used Apache JMeter to test and record the latency of our benchmarks, i.e., the time between the delivery of the request and the reception of the first response.

Configuration. The basic configuration for JMeter to run the ad-hoc tests uses 4 parallel threads (users), with a 10-s ramp-up time, i.e., the time needed to reach the total number of threads, and 200 requests per user. For some ad-hoc tests, we considered more appropriate a slight modification of the basic configuration. For the **sleep** test we use 25 requests per user since we deem it not necessary to have a larger sample size as the function has a predictable behaviour. The **cold-start** is meant to deliberately disregards the best practices of serverless development to showcase how the platform handles the cold start of "heavy" functions. For this reason, we throttle the invocations of these functions one every 11 min to let caches timeout—OpenWhisk's default cache timeout is 10 min. – and we use only 1 user performing 3 requests; this is enough to witness the effect on cache invalidation and initialisation times. Finally, due to the fact that the **data-locality** test is resource-heavy, we use only 50 repetitions for each of the 4 users; this is enough to witness data-locality effects.

We have a different configuration for each Wonderless test: **slackpost** has 1 user, 100 repetitions, and a 1-s pause, to account for Slack API's rate limits; **pycatj** has 4 parallel users, 200 repetitions, and a 10-s ramp-up time, akin to the default for ad-hoc tests; **terrain** has 1 user, 5 repetitions, and a 20-s pause, since the task is already a stress test and the amount of parallel computation on the node is high.

For each test with the exclusion of the previously mentioned **cold-start**, we execute 10 runs, removing and re-deploying the whole platform every 2 repetitions to avoid benchmarking specific configurations, e.g., bad, random configurations where vanilla OpenWhisk elects as primary a high-latency worker.

Cluster. For both reproducibility and reliability, we automatised all the levels of the deployment steps: the provisioning of the virtual machines (VMs) and both the deployment of Kubernetes and of (our extended version of) OpenWhisk on Google Cloud Platform via a Terraform and Ansible scripts. Once the Kubernetes cluster is up and running, we use the Helm package from

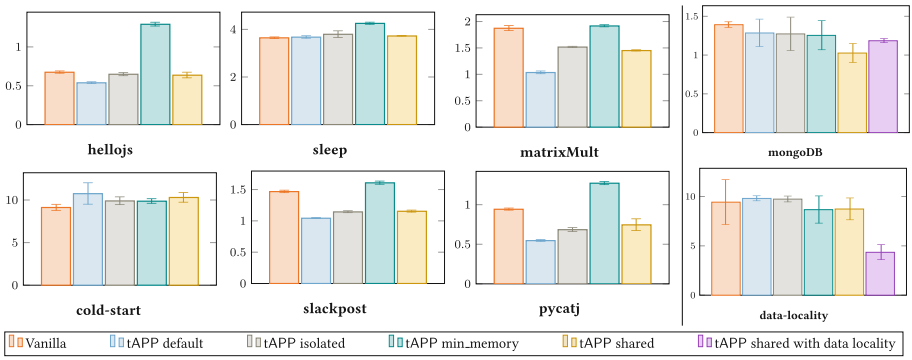

Fig. 4. Left: overhead tests (no data locality effects), average latency (bars) and standard deviation (barred lines) in seconds. Right: data-locality tests, average latency (bars) and standard deviation (barred lines) in seconds.

`openwhisk-deploy-kube` [2], that we forked to implement a tAPP-specific package for the installation with our custom controller image. This automatically deploys every component on a Kubernetes cluster and allows the user to parameterize the configuration of the deployment; specifically, we configure the deployment to select our tAPP-based controller image. The vanilla version os OpenWhisk is instead the one from OpenWhisk's official repository at https://github.com/apache/openwhisk-deploy-kube, commit 18960f.

We deployed both the vanilla and extended versions of OpenWhisk on a cluster of six virtual machines distributed across two regions (corresponding to two zone labels used in the deployment). We used a Kubernetes master node (not used as a computation node by OpenWhisk), along with one controller and one worker in the first region: *France Central*. The other controller and its two associated workers were in the second region: *East US*. All workers are *Standard_DS1_v2* Azure virtual machines, while the Kubernetes master node and both controllers are *Standard_B2s* Azure virtual machines. For the test, we also deployed two machines in the AWS region (*us-east*): a *t2.micro* EC2 instance for MongoDB and a *t2.medium* EC2 instance for the **terrain** backend. All machines (both on Azure and AWS) ran Ubuntu 20.04. To identify the best target for the data-locality tests, we measured the latency between the five (excluding the Kubernetes master node) cluster nodes and the two EC2 instances, which averages at 2 ms for machines located in *East US*, and 80ms for machines located in *France Central*. This identified the *East US* nodes as the optimal targets. The code used to deploy and run the tests is available at [2].

5.2 Results

We now present the results of running our tests on vanilla OpenWhisk and our prototype. In particular, we test our extension under all four topology-based

worker distribution policies: *default*, *isolated*, *min_memory*, and *shared* (cf. Sect. 4).

An initial comment regards **terrain**. While we could deploy this project, at runtime we observed up to 60% of timeouts and request errors (in comparison, the other tests report 0% failure rate). This test is a real-world one and, according to our testing methodology (cf. Sect. 5), we use its code as-is. Since this error rate is too high for valid tests, we discard it in this section (its raw data is in [1], for completeness).

We first present the results of the overhead tests and then the data-locality ones.

Overhead Tests. To better compare the overhead of our extension w.r.t. the vanilla OpenWhisk one, we run the **hellojs**, **sleep**, **matrixMult**, **cold-start**, **slackpost**, and **pycatj** without a tAPP script. As a consequence, we also do not tag test functions, since there would be no policies to run against. As specified in Section 4, this makes our platform resort to the original scheduling logic of OpenWhisk, although it prioritises (and undergoes the overhead of) scheduling functions on co-located workers. These tests are therefore useful to evaluate the impact on performance of our four zone-based worker distribution policies, in comparison with the topology-agnostic policy hard-coded in OpenWhisk (cf. Section 2).

We report on the left of Fig. 4, in seconds, the average (bars) and the variance (barred lines) of the latency of the performed tests. For reference, we report in [1] all the experimental data. Since the standard deviation in the results is generally small, we concentrate on commenting on the results of the averages.

In the results, Vanilla OpenWhisk has better performance w.r.t. all our variants in the **sleep** and the **cold-start** cases, where all tested policies have similar performance. The latency in these tests does not depend on the adopted scheduling policies, but on other factors: the three-second sleep in **sleep**, the long load times in **cold-start**. While we expected a sensible overhead in both cases, we found encouraging results: the overhead of topology-based worker selection strategies is negligible—particularly in the **sleep**, where the *shared* policy almost matches the performance of vanilla OpenWhisk.

In the other four tests (**hellojs**, **matrixMult**, **slackpost**, and **pycatj**), the *default* worker distribution policy outperforms both vanilla OpenWhisk and the other policies. This policy combines the standard way in which OpenWhisk allocates resources (where each worker reserves the same amount of resources for each controller) and our topology-based scheduling approach (where each controller selects workers in the same zone and uses remote workers only when the local ones are overloaded). These results confirm that the latency reduction from topology-based scheduling compensates (and even overcomes) its overhead—in some cases, the performance gain is significant, e.g., **matrixMult** shows a latency drop of 44%.

We deem the good performance of our extension in these tests (spanning simple and more meaningful computation and real-world applications) a positive result. Indeed, we expected topology-based scheduling to mainly allay data local-

ity issues, but we have experimentally observed significant performance improvements also in tests free from this effect.

We also note that the *min_memory* policy tends to perform the worst. To explain this fact, we draw attention to also the results of the *isolated* policy: both strategies can lead to saturated zones when faced with many requests, but they act differently with overloaded local workers. The *isolated* policy ignores remote workers and returns control to Nginx, which passes the invocation to a different controller. The *min_memory* policy instead tries to access remote workers with minimal resource availability, which can lead to higher latencies due to queuing and remote communications. The results of *default* and *shared* reinforce this conclusion: they increase resource sharing within the cluster and mitigate possible asymmetries (here, we had two workers in one zone and one in the other).

Data-Locality Tests. For the data-locality tests, we first run them without tagging functions and provide no tAPP script, thus comparing vanilla OpenWhisk and our extension on a common ground where the main difference between the two stands on the four distribution policies applied at deployment level and their overhead. Then, we ran the same tests (on our extension), but we tagged the functions and provided a tAPP script that favours executing functions on workers close to the data source.

We report on the right of Fig. 4, in seconds, the average (bars) and the variance (barred lines) of the latency of the data-locality tests **mongoDB** and **data-locality**—the full experimental data is in [1]. For brevity, we show, with the right-most bar on the right of Fig. 4, the results of the best-performing distribution policy (*shared*, see below) paired with the mentioned tAPP script.

As expected, in all tests our extension outperforms vanilla OpenWhisk, confirming previous evidence on data locality [12] and presenting useful applications of topology-aware scheduling policies for topology-dependent workflows.

In **mongoDB**, our extension outperforms vanilla OpenWhisk under all strategies, although it undergoes a higher variance. The small variance of vanilla OpenWhisk in this test is probably thanks to the light test query, which mitigates instances where vanilla OpenWhisk uses high-latency workers.

The results from **data-locality** confirm the observation above. There, the variance for vanilla OpenWhisk is larger—quantitatively, the variances of **mongoDB** for our extension stay below 0.5 s, while the variance of vanilla OpenWhisk in **data-locality** is 9-fold higher: 4.5 s. Here, the heavier test query strongly impacts the performance of those "bad" deployments that prioritise high-latency workers.

More precisely, the best performing strategies are *shared* for **mongoDB** and *min_memory* for **data-locality**. In the first case, since the query did not weigh too much on latency (e.g., bandwidth-wise), mixing local and remote workers favoured the *shared* policy, which, after exhausting its local resources, can freely access remote ones. In the second case, the *min_memory* policy performed slightly better than the *shared* one. We attribute this effect to constraining the

selection of workers mainly to the local zone and resorting in minimal part to remote, higher-latency workers.

Given the results above, we performed the tAPP-based tests (right-most column on the right of Fig. 4) with the *shared* policy[3].

Compared to the tag-less *shared* policy, the tagged case in **mongoDB** is a bit slower, but more stable (small variance). In **data-locality** it almost halves the run time of the tag-less case.

These tests witness the trade-off of using tAPP-based scheduling to exploit data locality and the overhead of parsing the tAPP script: due to its many lightweight requests, **mongoDB** represents the worst case for the overhead, but the test still outperforms vanilla OpenWhisk (showing that the overhead is compensated by the advantages of our worker selection strategies); in **data-locality**, the heaviness of the query and the payload favours spending a small fraction of time to route functions to the workers with lower latency to the data source.

6 Related Work and Conclusion

Related Work. Many works tackle minimising serverless function invocation latency, often trying to optimise function scheduling [17,20].

One work close to ours is by Sampé et al. [19], where the authors propose to favour data locality by allocating functions to storage workers. The main difference with our proposal is that we designed tAPP to specify scheduling constraints on topologies, where data locality may emerge; contrarily Sampé et al. frame the problem as topologies induced by data-locality issues.

Broadening our scope, we find proposals like Banaei et al. [7], who present a scheduling policy that governs the order of invocation processing, depending on the availability of the resources they use; Abad et al. [3] who propose a package-aware technique that favours re-using the same workers for the same functions to cache dependencies; Suresh and Gandhi [23], who introduced a scheduling policy oriented by resource usage of co-located functions on workers; Steint [22] and Akkus et al. [4] who respectively present a scheduler based on game-theoretic allocation and on the interaction of sandboxing of functions and hierarchical messaging. Other works rely on the state and relations among functions to determine scheduling policies. Examples include scheduling functions within a single workflow as threads within a single process of a container instance, reducing overhead by sharing state among them [15]; using state by supporting both global and local state access, aiming at performance improvements for data-intensive applications [21]; associating each function invocation with a shared log among serverless functions [13].

Drawing a comparison between the above works and ours, by using tAPP, the user expresses explicitly topologies considered at scheduling time, while topologies emerge as implicit, runtime configurations in the other proposals.

[3] In **data-locality**, *min_memory* has a slightly lower average than *shared*, but the latter has both lower variance and maximal latency.

Conclusion. We presented a tAPP-based serverless platform implementation for the specification and execution of topology-aware serverless scheduling policies. We used the presented tool to show that topology-aware scheduling can improve the performance of serverless architectures. Indeed, our benchmarks have shown that in almost all the considered test cases, the tAPP-based solution outperforms the unmodified platform, making it suitable both for generic applications, and especially for locality-sensitive functions.

Regarding future work, we consider extending the support for tAPP to other serverless platforms, like OpenLambda, OpenFAAS, and Fission. We also plan to expand our range of tests: both to include other aspects of locality (e.g., sessions) and specific components of the platform (e.g., message queues, controllers), and new benchmarks for alternative platforms, to elicit the peculiarities of each implementation. Moreover, we plan to consider cloud-edge use cases, where both local and remote machines execute functions and may benefit from topology-aware optimisations that exploit data locality. Regarding tests, we remark on the general need for more platform-agnostic and realistic suites, to obtain fairer and thorough comparisons.

References

1. tAPP-based openwhisk extension (2022). https://github.com/mattrent/openwhisk
2. Repository of rejected projects from wonderless (2022). https://github.com/mattrent/openwhisk-deploy-kube
3. Abad, C.L., Boza, E.F., Eyk, E.V.: Package-aware scheduling of faas functions. In: Proceedings of ACM/SPEC ICPE, pp. 101–106. ACM (2018). https://doi.org/10.1145/3185768.3186294
4. Akkus, I.E., et al.: SAND: towards high-performance serverless computing. In: Proceedings of USENIX/ATC, pp. 923–935 (2018)
5. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: the definitive guide: time to relax. " O'Reilly Media, Inc." (2010)
6. Armbrust, M., et al.: Above the clouds: a Berkeley view of cloud computing. University of California, Berkeley, Rep. UCB/EECS **28**(13), 2009 (2009)
7. Banaei, A., Sharifi, M.: Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform. J. Supercomput. (2021). https://doi.org/10.1007/s11227-021-04057-z
8. Bernstein, D.: Containers and cloud: from lxc to docker to kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014)
9. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: 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)
10. Eskandani, N., Salvaneschi, G.: The wonderless dataset for serverless computing. In: Proceedings of IEEE/ACM MSR, pp. 565–569 (2021). https://doi.org/10.1109/MSR52588.2021.00075
11. Hassan, H.B., Barakat, S.A., Sarhan, Q.I.: Survey on serverless computing. J. Cloud Comput. **10**(1), 1–29 (2021)
12. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with openlambda. In: Proceedings of USENIX HotCloud (2016)

13. Jia, Z., Witchel, E.: Boki: stateful serverless computing with shared logs. In: Proceedings of ACM SIGOPS SOSP, pp. 691–707. ACM, New York, NY, USA (2021). https://doi.org/10.1145/3477132.3483541
14. Jonas, E., et al.: Cloud programming simplified: a Berkeley view on serverless computing. Technical report UCB/EECS-2019-3, EECS Department, University of California, Berkeley (2019)
15. Kotni, S., Nayak, A., Ganapathy, V., Basu, A.: Faastlane: accelerating function-as-a-service workflows. In: Proceedings of USENIX ATC, pp. 805–820. USENIX Association (2021)
16. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: a distributed messaging system for log processing. In: Proceedings of NetDB, vol. 11, pp. 1–7 (2011)
17. Kuntsevich, A., Nasirifard, P., Jacobsen, H.A.: A distributed analysis and benchmarking framework for apache openwhisk serverless platform. In: Proceedings of Middleware (Posters), pp. 3–4 (2018)
18. Oren Ben-Kiki, Clark Evans, I.d.N.: Yaml ain't markup language (yaml™) version 1.2 (2021). https://yaml.org/spec/1.2.2/
19. Sampé, J., Sánchez-Artigas, M., García-López, P., París, G.: Data-driven serverless functions for object storage. In: Proceedings of Middleware, pp. 121–133. ACM (2017). https://doi.org/10.1145/3135974.3135980
20. Shahrad, M., Balkind, J., Wentzlaff, D.: Architectural implications of function-as-a-service computing. In: Proceedings of MICRO, pp. 1063–1075 (2019)
21. Shillaker, S., Pietzuch, P.: Faasm: Lightweight isolation for efficient stateful serverless computing. In: Proceedings of USENIX ATC, pp. 419–433. USENIX Association (2020)
22. Stein, M.: The serverless scheduling problem and noah. arXiv preprint arXiv:1809.06100 (2018)
23. Suresh, A., Gandhi, A.: Fnsched: an efficient scheduler for serverless functions. In: Proceedings of WOSC@Middleware, pp. 19–24. ACM (2019). https://doi.org/10.1145/3366623.3368136