A Declarative Approach to Topology-Aware Serverless Function-Execution Scheduling

(Article begins on next page)

24 April 2024

A Declarative Approach to Topology-Aware Serverless Function-Execution Scheduling

Giuseppe De Palma
Università di Bologna, Italy
giuseppe.depalma2@unibo.it

Saverio Giallorenzo
Univ. di Bologna, Italy and INRIA, France
saverio.giallorenzo2@unibo.it

Jacopo Mauro
University of Southern Denmark, Denmark
mauro@imada.sdu.dk

Matteo Trentin
Università di Bologna, Italy
matteo.trentin@studio.unibo.it

Gianluigi Zavattaro
Università di Bologna, Italy and INRIA, France
gianluigi.zavattaro@unibo.it

Abstract—State-of-the-art serverless platforms use hard-coded scheduling policies that are unaware of the possible topological constraints of functions. Considering these constraints when scheduling functions leads to sensible performance improvements, e.g., minimising loading times or data-access latencies. This issue becomes more pressing when considered in the emerging multi-cloud and edge-cloud-continuum systems, where only specific nodes can access specialised, local resources. To address this problem, we present a declarative language for defining serverless scheduling policies to express constraints on topologies of schedulers and execution nodes. We implement our approach as an extension of the OpenWhisk platform.

Figure 1: Representation of the case study.

I. Introduction

Serverless is a cloud service that lets users deploy architectures as compositions of stateless functions, delegating all system administration tasks to the serverless platform [1]. This has two benefits for users. First, they save time by delegating resource allocation, maintenance, and scaling to the platform. Second, they pay only for the resources that perform actual work, and eschew the costs of running idle servers.

For example, Amazon AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions[1] are managed serverless offers by popular cloud providers, while OpenWhisk, OpenFaaS, OpenLambda, and Fission[2] are open-source alternatives, used also in private deployments.

In all these cases, the platform manages the allocation of function executions over the available computing resources, also called *workers*. However, not all workers are equal when allocating functions. Indeed, effects like *data locality* [2]—due to high latencies to access data—or *session locality* [2]—due to the need to authenticate and open new sessions to interact with other services—can sensibly increase the run time of functions. These issues become more prominent when considered in multi-cloud and edge-cloud-continuum systems, where only specific workers can access some local resources.
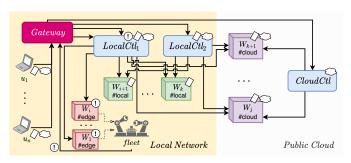
To tackle this problem, we present a solution that lets users define *topology-aware* scheduling policies able to mitigate and/or rule out inefficient function allocations.

More specifically, with "topology" of a serverless platform we mean: 1) the deployment of the platform over different *zones*, i.e., sets of resources geographically located in the same area, 2) the presence in such zones of several *controllers*, i.e., the components that manage the scheduling of functions, and 3) the availability of different *workers*, each one located in one zone but potentially reachable by all the controllers.

Motivating example: We clarify the concepts above with a case study from a company among our industry partners. We deem the case useful to clarify the motivation behind our work and help understand our contribution.

The case concerns an edge-cloud-continuum system to control and perform both predictive maintenance and anomaly detection over a fleet of robots in the production line. The system runs three categories of computational tasks: i) predictions of critical events, performed by analysing data produced by the robots, ii) non-critical predictions and generic control activities, and iii) machine learning tasks. Tasks i) follow a closed-control loop between the fleet that generates data and issues these tasks and the workers that run these tasks and can act on the fleet. Since tasks i) can avert potential risks, they must execute with the lowest latency and their control signals must reach the fleet urgently. The users of the system launch the other categories of tasks. These are not time-constrained,

---

but tasks iii) have resource-heavy requirements.

We depict the solution that we have designed for the deployment of the system in Figure 1. We consider three kinds of functions, one for each category of tasks: critical functions ⚠ (in Figure 1), generic functions ◇, and machine learning functions ⬭. To guarantee low-latency and the possibility to immediately act on the robots, we execute *critical* functions ⚠ on edge devices (workers $W_1,...,W_i$ in Figure 1) directly connected to the robots. Since machine-learning algorithms require a considerable amount of resources that the company prefers to provision on-demand, we execute the machine-learning functions ⬭ on a public cloud, outside the company's perimeter ($W_{k+1},...,W_j$ in Figure 1). The generic functions ◇ do not have specific, resource-heavy requirements. Hence, we schedule these preferably on the local cluster ($W_{i+1},...,W_k$ in Figure 1) and use on-demand public-cloud workers when the local ones are at full capacity.

For performance and reliability, our solution considers two function-scheduling controllers for the internal workers, i.e., the controllers $LocalCtl_1$ and $LocalCtl_2$, and one for cloud workers, i.e., the controller $CloudCtl$. One local controller, namely $LocalCtl_1$, has a dedicated low-latency connection with the edge devices able to act on the fleet.

Finally, a $Gateway$ balances the load among the controllers. To follow the requirements of the company, instead of adopting a generic round-robin policy, we need to instruct the $Gateway$ to forward critical functions ⚠ to $LocalCtl_1$, the generic functions ◇ to one between $LocalCtl_1$ and $LocalCtl_2$, and the cloud functions ⬭ to $CloudCtl$ (or to any other controller when the latter is not available).

Contribution of the paper: The case above presents a scenario with a multi-zone serverless platform deployment (local network and public cloud) and where the function-execution scheduling policy depends on a topology of different clusters (edge-devices, local cluster, and cloud cluster). The scheduling policies influence the behaviour of both the gateway and the controllers, which need to know the current status of the workers (e.g., to execute generic functions in the cloud when the local cluster is overloaded).

Instead of hard-coding the desired behaviour in the deployment of the serverless platform, the approach that we propose in this paper is based on a new declarative language, called TAPP (Topology-aware Allocation Priority Policies), used to write *configuration* files describing topology-aware function-execution scheduling policies. In this way, following the Infrastructure-as-Code philosophy, users (typically DevOps) can keep all relevant scheduling information in a single repository (in one or more TAPP files) which they can version, change, and run without incurring in downtimes due to system restarts to load new configurations.

We also implement a serverless platform that supports TAPP-specified scheduling policies. We avoid starting from scratch and we build upon the serverless platform
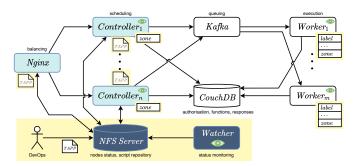

Figure 2: Architectural scheme of our extension of OpenWhisk. Modified components from OpenWhisk are in light blue and new components are highlighted in yellow .

presented in [3], which is an extension of OpenWhisk where worker-selection happens via tags associated with functions.

II. PRELIMINARIES

The main proprietary serverless platforms are AWS Lambda, Google Cloud Functions, and Azure Functions. Despite their large user base, these solutions do not provide source code. Also open-source alternatives are gaining traction in the serverless market. The most popular ones are Apache OpenWhisk, OpenFaaS, and OpenLambda. Since these platforms have similar architectures [4] we focus on the most popular, open-source one: Apache OpenWhisk.

Apache OpenWhisk is an open-source, serverless platform initially developed by IBM and donated to the Apache Software Foundation. We report in Figure 2 a scheme of the architecture of OpenWhisk. For compactness, we include in Figure 2 the modified and new elements introduced by our extension, which we describe in Section IV. Here, we focus only on the original components and functionalities of OpenWhisk.

In Figure 2, from left to right, we first find *Nginx*, which acts as the gateway and load balancer to distribute the incoming requests. *Nginx* forwards each request to one of the *Controller*s in the current deployment. The *Controller*s are the components that decide on which of the available computation nodes, called *Worker*s, to schedule the execution of a given function. *Controller*s and *Worker*s do not interact directly but use Apache *Kafka* [5] and *CouchDB* [6] to respectively handle the routing and queuing of execution requests and to manage the authorisations, the storage of functions and of their outputs/responses. *Worker*s execute functions using Docker containers. To schedule executions, *Controller*s follow a hard-coded policy that mediates load balancing and caching. This works by trying to allocate requests to the same functions on the same *Worker*s[3], hence saving time by skipping the retrieval of the function from *CouchDB* and the instantiation of the container already cached in the memory of the *Worker*.

---

[3]The OpenWhisk allocation policy, called "co-prime scheduling", associates a function to a hash and a step size to identify a sequence of possible workers able to run the function (assigned to the first non-overloaded one).

$$policy\_tag \in \mathit{Identifiers} \cup \{\texttt{default}\} \qquad label \in \mathit{Identifiers} \qquad n \in \mathbb{N}$$

app	::=	$\overline{tag}$
tag	::=	$policy\_tag$: - *controller?* *workers* *strategy?* *invalidate?* *strategy?* *followup?*
controller	::=	`controller` : *label* (`topology_tolerance` : (`all` | `same` | `none`))?
workers	::=	`workers`: $\overline{\text{-}\ label}$
	|	`workers`: $\overline{\text{-}\ *(label)?\ \textit{strategy?}\ \textit{invalidate?}}$
strategy	::=	`strategy` : (`random` | `platform` | `best_first`)
invalidate	::=	`invalidate` : (`capacity_used` : n% | `max_concurrent_invocations` : n | `overload`)
followup	::=	`followup` : (`default` | `fail`)

Figure 3: The syntax of TAPP (the extensions from APP are highlighted).

III. TOPOLOGY-AWARE SERVERLESS SCHEDULING

We now present our approach to topology-aware function-execution scheduling and the TAPP language.

The approach relies on *policy tag*s that associate functions to scheduling policies. A tag identifies a policy (e.g., we can use a tag "critical" to identify the scheduling behaviour of the critical functions of our case study) and it marks all those functions that shall follow the same scheduling behaviour (e.g., any kind of critical function of the study).

Topologies are part of policies and come in two facets. *Physical* topologies relate to *zones*, which can represent availability zones in public clouds and plants in multi-plant industrial settings. *Logical* topologies instead represent partitions of workers. The logical layer expresses the constraints of the *user* and identifies the pool of workers which can execute a given function (e.g., for performance). The smallest logical topology is the singleton, i.e., a worker, which we identify with a distinct label (e.g., W_1 in Figure 1). In general, policies can target lists of singletons as well as aggregate multiple workers in different sets.

The interplay between the two topological layers determines which workers a controller can use to schedule a function. For example, we can capture the scheduling behaviour of the critical functions of our case study in this way: 1) we assign $LocalCtl_1$, $LocalCtl_2$, and $W_1,...,W_k$ to the same zone, 2) we configure said workers to only accept requests from co-located controllers (this, e.g., excludes access to $CloudCtl$), and 3) we set the policy of the critical functions to only use the workers tagged with the *edge* label—#edge in Figure 1.

Besides expressing topological constraints, policies can include other directions such as the strategy followed by the controller to choose a worker within the pool of the available ones (e.g., to balance the load evenly among them) and when workers are ineligible (e.g., w.r.t. their resource quotas).

To the best of our knowledge, the serverless platform by De Palma et al [3] is the only one that supports declaratively, customised function-execution scheduling policies by using an ad-hoc language dubbed Allocation Priority Policies (APP). APP lets users associate workers to functions, but it does not include any notion of physical topology (the language has no visibility over controllers, which all behave the same) and it supports only the singleton and universal logical topologies (either single workers or all of them). Notwithstanding these limitations, we deemed it feasible to use APP as basis for our approach and propose a topology-aware extension called TAPP.

In the following, we report a summary of the main components of APP, we describe TAPP, and discuss a TAPP script that implements the semantics of the case study in Section I.

The APP language [3]: We report the syntax of (T)APP in Figure 3—the highlighted parts belong to TAPP. An APP script essentially pairs two entities: *i*) scheduling policies, identified by a *policy tag* which represents some functions, and *ii*) a list of *workers* which can execute those functions— in Figure 3, $\overline{bars}$ indicate non-empty lists. More in general, a policy tag points to a list of policy *blocks*. Each block involves a list of *workers*, each identified by a distinct *label*. Optional elements (marked with ?) modify the default semantics of the block: *strategy* and *invalidate*. The first defines the `strategy` that the controller follows to choose among the *workers* in the block (e.g., `random` randomly selects one worker of the block, `best_first` selects workers following their descending order of appearance in the block). The `invalidate` option defines when one worker is invalid (e.g., `capacity_used` when it reaches a maximal % of CPU, `max_concurrent_invocations` when it reaches a maximal number of buffered concurrent invocations). When a selected worker is invalid, we try to schedule the function on the other (valid) workers of the block following the `strategy`, until exhaustion. If all workers in the policy tag are invalid, the policy fails and we execute the (optionally) specified `followup` rule: the `default` option tries to execute the function via the (special) `default` tag (if any); the `fail` option skips the `default` block and aborts the scheduling of the function.

The TAPP language: The syntax of TAPP (Figure 3) results from extending APP with (the highlighted) parts, which capture topology-aware function scheduling policies. First, we introduce the *controller* . This is an optional, block-level parameter that identifies which of the possible, available `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.

A *controller* clause can have `topology_tolerance` as optional parameter. This additional parameter allows users to further refine how TAPP handles failures (of controllers). Indeed, when deploying controllers and workers, users can label them with the topological *zone* they belong in[4]. Hence, 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.

The `topology_tolerance` option specifies what workers an alternative controller can use: `all` is the default and most permissive option and imposes no restriction on 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.

TAPP expands the expressiveness of the universal selector `*` found in APP—corresponding to the second, highlighted `workers:...` fragment in Figure 3). In APP, assigning `*` to the `workers` parameter indicates that the policy encompasses all available workers. In TAPP, users can restrict the scope of `*` by suffixing it with a label, e.g., `workers:*local` selects only those workers associated to the *local* label.

The scheduling on `*`-induced worker-sets follows the same logic of block-level worker selection: it exhausts all workers before deeming the block invalid. Since worker-set policies could differ from block-level ones, we allow users to define the `strategy` and `invalidate` policies to select the worker in the set. For example, if we pair the above selection with a *strategy* and an *invalidate* options, e.g.,

`workers:*local  strategy:random  invalidate:capacity_used:50%`

we tell the scheduler to adopt the `random` strategy and the `capacity_used` invalidation policy when selecting the workers in the *local* set. When worker-sets omit strategies or invalidation options, they follow those of their enclosing block.

Lastly, TAPP extends APP by letting users express a selection strategy for policy blocks. This is represented by the highlighted, optional *strategy* fragment of the *tag* rule. The extension is backwards compatible, i.e., when we omit to define a *strategy* policy for blocks, TAPP has the same semantics as APP, trying to allocate 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 uniformly distributing requests among the available controllers.

A. Case Study

We exemplify TAPP by showing and commenting on the salient parts of a TAPP script—reported in Figure 4—that captures the scheduling semantics of the case in Figure 1.

In the script, at lines 1–6, we define the tag associated to `critical` (①) functions: only `LocalCtl_1` can manage their

[4]TAPP neglects zone labels of controllers and workers, which is infrastructure-level information, and it only specifies co-location constraints.

```
1  critical:
2    - controller: LocalCtl_1
3      workers:
4        - *edge
5      strategy: random
6    followup: fail
7  machine_learning:
8    - controller: CloudCtl
9      workers:
10       - *cloud
11     topology_tolerance: same
12   followup: default

13 default:
14   - controller: LocalCtl_1
15     workers:
16       - *internal
17       strategy: random
18     - *cloud
19       strategy: random
20     strategy: best_first
21   - controller: LocalCtl_2
22     workers: # same as above
23     strategy: best_first
24   strategy: random
```

Figure 4: A TAPP script that implements the scheduling semantics of the case study in Section I (Figure 1).

scheduling, they can only execute on `#edge`/`*edge` workers ($W_1,...,W_i$ in Figure 1), and no other policy can manage them (`followup:fail`). At line 5 we specify to evenly distribute the load among all `*edge` workers with `strategy:random`.

At lines 7–12, we find the tag of the `machine_learning` (◯) functions. We define `CloudCtl` as the controller and consider all `#cloud` workers ($W_{k+1},...,W_j$ in Figure 1) as executors, i.e., any worker in the public cloud $W_{k+1},...,W_j$. Notice that at line 12 we specify to use the `default` policy as the `followup`, in case of failure. The interaction between the `followup` and the `topology_tolerance` (line 11) parameters makes for an interesting case. Since the `topology_tolerance` is (the) `same` (zone of the controller `CloudCtl`), we allow other controllers to manage the scheduling of the function (in the `default` tag) but we continue to restrict the execution of machine-learning functions only to workers within the `same` zone of `CloudCtl`, which, here, coincide with `#cloud`-tagged workers.

Lines 13–24 define the special, `default` policy tag, which is the one used with tag-less functions (here, our generic ones ◇) and with failing tags targeting it as their `followup` (as seen above, line 12). In particular, the instruction at line 24 indicates that the `default` policy shall `random`ly distribute the load on both worker blocks (lines 14–20 and 21–23), respectively controlled by `LocalCtl_1` and `LocalCtl_2`. Since the two blocks at lines 14–20 and 21–23 are the same, besides the `controller` parameter, we focus on the first one. There, we indicate two sets of valid `workers`: the `#internal` ones (line 16, $W_{i+1},...,W_k$ in Figure 1) and the `#cloud` ones (as seen above, for lines 9–10). The instruction at line 20 (`strategy:best_first`) indicates a precedence: first we try to run functions on the `#local` cluster and, in case we fail to find valid workers, we offload on the `#cloud` workers—in both cases, we distribute the load `random`ly (lines 17 and 19).

IV. TAPP IN OPENWHISK

We modified OpenWhisk to support TAPP-based scheduling (available at https://github.com/mattrent/openwhisk). In particular, to manage the deployment of components, we pair OpenWhisk with the popular and widely-supported container orchestrator Kubernetes. The extended platform

is available as an open-source project.

Figure 2 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 Figure 2).

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 detail the two new services, we discuss the changes to the existing OpenWhisk components, and we conclude by describing how the proposed system supports live-reloading of TAPP configurations.

Watcher and NFS Server Services: To support TAPP-based scheduling, we need 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. The new *Watcher* service fits this purpose: it gathers deployment-specific information and maps it to TAPP-level properties. 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. Figure 2), and stores the mapping into the *NFS Server*.

As shown in Figure 2, Nginx uses the output of the Watcher to forward function-execution requests to controllers. This allows TAPP scripts to define which controller to target without the need to specify a pod identifier, but rather use a label (e.g., `CloudCtl` in Figure 4). Besides abstracting 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.

Nginx, OpenWhisk's Entry Point: OpenWhisk's Nginx forwards requests to all available controllers, following a hard-coded round-robin policy. To support TAPP, we intervened on how Nginx processes incoming request of function execution.

To do this, we used *njs*[5]: a subset of the JavaScript language that Nginx provides to extend its functionalities. Namely, we wrote a 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

---
[5]https://nginx.org/en/docs/njs/index.html

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: To associate labels with pods, we exploit the topology labels provided by Kubernetes. These labels are names assigned to nodes and they are often used to orient pod allocation. Labels offer an intuitive way to describe the structure of the cluster, by annotating their zones and attributes. In Figure 2 we represent 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 without knowing how the other controllers are using them. 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, 256MB). When workers have no controller in their topological zone, or no topological zone at all, we follow the default policy. Also this policy has a drawback: 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 allows controllers to access primarily local workers and let them access foreign ones after having exhausted the local ones. This policy mediates between partitioning resources and the efficient usage of the available ones, although it suffers a stronger effect of resource-grabbing from remote controllers.

To schedule functions, controllers follow the policies declared in the TAPP scripts (if any), accessing both these and topological information in the same way as described for Nginx. When no TAPP script is available, controllers resort to their original, hard-coded logic (explained in Section II) but still prioritise scheduling functions on co-located workers.

Dynamic update of topologies and TAPP scripts: Since both the cluster's topology, its attributes, and the related TAPP scripts can change (e.g., to include a new node or a new policy tag), we designed our TAPP-based prototype to dynamically support such changes, avoiding stop-and-restart downtimes.

To do this, we chose to store a single global copy of the policies into 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 of the change and let them handle cache invalidation and retrieval.

V. RELATED WORK

The proposal closest to ours is [3], on which we build to implement our approach and prototype. As mentioned in Section III, the solution by De Palma et al. [3], although not explicitly stated by the authors, captures some degenerate cases of topology-aware scheduling, which TAPP generalises. Besides these commonalities, [3] lacks any other notion of topology from this work and does not distinguish among (located) controllers. Another work close to ours is by Sampé et al. [7], who present an approach that allocates functions to storage workers, favouring data locality. The main difference with our work is that the one by Sampé et al. focusses on topologies induced by data-locality issues, while we consider topologies to begin with, and we capture data locality as an application scenario.

More in general, Banaei et al. [8] introduce a scheduling policy that governs the order of invocation processing depending on the availability of the resources they use. Abad et al. [9] present a package-aware technique that favours re-using the same workers for the same functions to cache dependencies. Suresh and Gandhi [10] show a scheduling policy oriented by resource usage of co-located functions on workers. Other scheduling policies exploit the state and relation among functions. For example, Kotni et al. [11] present an approach that schedules functions within a single workflow as threads within a single process of a container instance, reducing overhead by sharing state among them.

The main difference between these works and ours is that in the former topologies (if any) emerge as implicit, runtime artefacts and scheduling do not directly reason on them. Moreover, being a general approach to scheduling, future work on TAPP can include scheduling policies proposed in these works, e.g., as strategies for worker selection.

VI. CONCLUSION

We introduced TAPP, a declarative language that provides DevOps with finer control on the scheduling of serverless functions. Being topology-aware, TAPP scripts can restrict the execution of functions within zones and help improve the performance (e.g., exploiting data or code locality properties), security, and resilience of serverless applications. To validate our approach, we presented a prototype TAPP-based serverless platform, developed on top of OpenWhisk.

Due to space limitations, we have not discussed the experimental validation of our implementation, reported for reference in the companion technical report [12]. Here we simply report the final conclusions of our experiments: our topology-aware scheduling is usually on par or outperforms hard-coded, vanilla OpenWhisk scheduling, in particular in tests that stress data locality.

Future work includes applying TAPP on different platforms, e.g., OpenLambda, OpenFAAS, and Fission and to formalise the semantics of TAPP, e.g., building on existing "serverless calculi" [13], [14].

REFERENCES

[1] J. M. Hellerstein *et al.*, *Serverless computing: One step forward, two steps back*, 2019.

[2] S. Hendrickson *et al.*, "Serverless computation with openlambda," in *Proc. of USENIX HotCloud*, 2016.

[3] G. De Palma *et al.*, "Allocation priority policies for serverless function-execution scheduling optimisation," in *Proc. of ICSOC*, ser. LNCS, Springer, 2020.

[4] H. B. Hassan *et al.*, "Survey on serverless computing," *Journal of Cloud Computing*, 2021.

[5] J. Kreps *et al.*, "Kafka: A distributed messaging system for log processing," in *Proc. of NetDB*, 2011.

[6] J. C. Anderson *et al.*, *CouchDB: the definitive guide: time to relax*. " O'Reilly Media, Inc.", 2010.

[7] J. Sampé *et al.*, "Data-driven serverless functions for object storage," in *Middleware*, ACM, 2017.

[8] A. Banaei and M. Sharifi, "Etas: Predictive scheduling of functions on worker nodes of apache openwhisk platform," *The Journal of Supercomputing*, Sep. 2021.

[9] C. L. Abad *et al.*, "Package-aware scheduling of faas functions," in *Proc. of ACM/SPEC ICPE*, ACM, 2018.

[10] A. Suresh and A. Gandhi, "Fnsched: An efficient scheduler for serverless functions," in *Proc. of WOSC@Middleware*, ACM, 2019.

[11] S. Kotni *et al.*, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. of USENIX ATC*, USENIX Association, 2021.

[12] G. De Palma *et al.*, *Topology-aware serverless function-execution scheduling*, https://arxiv.org/abs/2205.10176, 2022.

[13] M. Gabbrielli *et al.*, "No more, no less - A formal model for serverless computing," in *Proc. of COORDINATION*, ser. LNCS, Springer, 2019.

[14] A. Jangda *et al.*, "Formal foundations of serverless computing," *Proc. of ACM on Prog. Lang.*, 2019.