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
jacopo.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 and show relevant scenarios where our extension is on par with or outperforms vanilla OpenWhisk.

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.

Fig. 1: Representation of the case study.

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 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

\(^1\)Resp. https://aws.amazon.com/lambda/, https://cloud.google.com/functions/, https://azure.microsoft.com/
\(^2\)Resp. https://openwhisk.apache.org/, https://www.openfaas.com/, https://github.com/open-lambda/open-lambda, https://fission.io/
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 $W_1, \ldots, W_i$ 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 ($W_{k+1}, \ldots, W_j$ in Figure 1). The generic functions 🔄 do not have specific, resource-heavy requirements. Hence, we schedule these preferably on the local cluster ($W_i, \ldots, W_k$ 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 acts as load balancer among the controllers. However, 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 where we need to deploy the serverless platform over at least a couple of zones (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).

One can obtain a deployment of the case by modifying the source code of all the involved components and by hard-coding their desired behaviour. However, this solution requires a deep knowledge of the internals of the components and is a fragile solution, difficult to maintain and evolve.

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.

Besides presenting TAPP itself, we validate our approach by implementing a serverless platform that supports TAPP-specified scheduling policies. In doing so, we avoid starting from scratch and we build upon the serverless platform presented in [3], which provides an extension of OpenWhisk where worker-selection happens via tags associated with functions. Although the version of OpenWhisk from [3] is close to our idea, we had to introduce new components and improve the existing ones 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.. Moreover, we had to extend the original OpenWhisk architecture with a watcher service, which informs the gateway and the controllers on the current status of the nodes of the platform.

Beyond proving our idea realisable, our TAPP-based OpenWhisk version allows us to evaluate the performance of topology-aware scheduling policies. To do that, we compare our version against vanilla OpenWhisk considering several deployments of the platforms and several use-cases. In all cases our version is on par with or outperforms vanilla OpenWhisk.

**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 one could build upon and offer limited documentation on their inner workings. Fortunately, customisable and evolvable 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 describe their main components via 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 Controllers in the current deployment.

The Controllers are the components that decide on which of the available computation nodes, called Workers\(^3\), to schedule the execution of a given function. Controllers and Workers 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.

Workers execute functions using Docker containers. To schedule executions, Controllers 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 Workers\(^4\), 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.

### III. Topology-aware Serverless Scheduling

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

Our approach relies on policy tags 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 \(LocalCt1_1, LocalCt1_2\), 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, \(#\text{edge}\) 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., their resource quotas).

\(^3\)OpenWhisk’s documentation uses the more specific term “invokers”.

\(^4\)More precisely, the OpenWhisk allocation policy, called “co-prime scheduling”, associates a function to a hash and a step size. The hash finds the primary worker. The step size finds a list of workers used in succession when the preceding ones become overloaded.

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). Nonetheless, these limitations, inspired by the simplicity of the APP language, we deemed it feasible to use it as basis for our approach and propose a topology-aware extension of the APP language, 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, 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 controllers 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
We define controller strategy functions. We define controller workers: randomly we tell the scheduler to adopt the strategy controller to define an alternative controller can use. Specifically, all which only specify co-location constraints, i.e., requests to consider workers infrastructure to implement the TAPP constraints.

The scheduling on same -induced worker-sets follows the same logic of block-level worker selection: it exhausts all workers before deeming the block invalid. Since worker-set selection/invalidation 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 selection 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 local.

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.

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

In the script, at lines 1–6, we define the tag associated to critical (⋄) functions: only LocalCtl_1 can manage their 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 randomly 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 randomly (lines 17 and 19).

IV. TAPP in OpenWhisk

We modified OpenWhisk to support TAPP-based scheduling. In particular, to manage the deployment of components, we pair OpenWhisk with the popular and widely-supported container orchestrator Kubernetes 6.

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 labels and zones 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 hardcoded round-robin policy. To support TAPP, we intervened on how Nginx processes incoming request of function execution.

To do this, we used njs 7: 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 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 strive 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’s 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.

When scheduling functions, 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. In case 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. TESTING METHODOLOGY

We now describe the comparison of the vanilla version of OpenWhisk against our extension: the benchmarks we selected for the comparison and the relevant implementation details for running the tests and collecting the logs.

We devise two kinds of tests. Overhead tests measure the overhead between vanilla OpenWhisk and our prototype; these tests have no data locality issues. Data-locality tests benchmark the effect of data locality and measure the performance gain of topology-aware policies. In the following description we mark (O) overhead tests and (D) data-locality ones.

A. Benchmarks

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 [7] serverless benchmark dataset.

Ad-hoc Tests: Ad-hoc tests focus on specific traits:
- hellojs (O) implements a “Hello World” application. 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 seconds. This test benchmarks the handling of multiple functions running for several seconds and the management of their queueing process;
- matrixMult (O) multiplies two 100x100 matrices and returns the result to the caller. This test measures the performance of handling functions performing some meaningful computation;
- cold-start (O) is a parameter-less variant of hellojs that loads a heavy set of dependencies (42.8 MB) required and instantiated when the function starts. Moreover, we throttle the invocations of this function to one every 11 minutes to let caches timeout8. This function deliberately disregards serverless development best practices and showcases how the platform handles the cold start of “heavy” functions, i.e., the delay occurring when a function’s code has to be initialised on the worker;
- 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 ad-hoc tests use Javascript and run on Node.js 10. The tests that use MongoDB (version 5) run on Node.js 12.9

Real-world Tests: We draw our real-world tests from Wonderless [7]: a recent, peer-reviewed dataset that contains almost 2000 projects automatically scraped from GitHub. The projects target serverless platforms like AWS, Azure, Cloudflare, Google, Kubebench, and OpenWhisk.

Wonderless somehow reflects the current situation of serverless industrial adoption: the distribution of projects in Wonderless is heavily skewed towards AWS-specific applications. Indeed, out of the 1877 repositories in the dataset, 97.8% are AWS-specific. Unfortunately, we needed to exclude from our analysis these projects, since they often rely on using AWS-specific APIs and would require

8 The default OpenWhisk cache timeout is 10 minutes.
9 https://github.com/apache/openwhisk-runtime-nodejs.
sensible refactoring to run on other platforms—one such effort could be subject of future work but escapes the scope of the current one. This left 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:

- 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;
- 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;
- 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;
- 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 these real-world tests:

- **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 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.

### B. Test Environment

To perform load testing and collect benchmarks we used a well-known and stable tool: Apache JMeter\(^{11}\). We used JMeter without a graphical interface to dedicate all resources to the tests and maximise the number of concurrent requests. We consider as metric the latency, 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-second 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 test is long-running, so we use only 1 user performing 3 requests; this is enough to witness the effect on cache invalidation and initialisation times. Since 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-second pause, to account for Slack API’s rate limits; pycatj has 4 parallel users, 200 repetitions, and a 10-second ramp-up time, akin to the default for ad-hoc tests; terrain has 1 user, 5 repetitions, and a 20-second pause, since the task is already a stress test and the amount of parallel computation on the node is high. For each test, 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. The unique exception is cold-start, which is a long-running test (see the discussion above), for which we considered 3 runs; these are sufficient to evaluate cache invalidation and the corresponding initialisation times, which are independent of decisions taken at deployment time.

**Cluster:** We deployed both the vanilla and extended versions of OpenWhisk on a cluster composed of six virtual machines distributed across two different regions (corresponding to two zone labels used in the deployment). The vanilla version is the one from OpenWhisk’s official repository\(^{12}\) at commit 18960f.

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 were Standard_DS1_v2 Azure virtual machines, while the Kubernetes master node and both controllers were Standard_B2s Azure virtual machines.

We also deployed two machines in the same AWS region (us-east): a t2.micro EC2 instance for MongoDB

---

\(^{10}\)For reproducibility, we provide the list detailing the rejection criteria applied to each of the 63 non-AWS projects we discarded at [8].

\(^{11}\)https://jmeter.apache.org/

\(^{12}\)https://github.com/apache/openwhisk-deploy-kube
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 2ms 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 [8].

VI. RESULTS

We now present the results obtained running our tests on vanilla OpenWhisk and our topology-aware extension. In particular, we test our extension under all four topology-based worker distribution policies: default, isolated, min_memory, and shared (cf. Section IV).

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. Section V), we use its code as-is. Since this error rate is too high to consider the test valid, we discard it in this section (its raw data is in [8], for completeness).

In the following, we first present the results of the overhead tests and then the results from the data-locality ones.

A. Overhead tests

To better compare the overhead of our extension w.r.t. the vanilla OpenWhisk one, we run the hellos, 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 IV, 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. As such, these tests are 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 II).

We report in Figure 5, in seconds, the average (bars) and the variance (barred lines) of the latency of the performed tests. For reference, we report in [8] all the experimental data. Since the standard deviation in the results is generally small, we concentrate on commenting 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 (hellos, 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 to 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) an extremely positive result. Indeed, we expected topology-based scheduling to mainly allay data locality 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).

**B. Data-locality tests**

For the data-locality tests, we first comment on the tests we ran following the same modality of the overhead tests: we do not tag functions and provide no TAPP script. This lets us compare 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 in Figure 6, 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 [8]. For brevity, we show, with the right-most bar in Figure 6, 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 [2] 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 seconds, while the variance of vanilla OpenWhisk in **data-locality** is 9-fold higher: 4.5 seconds. 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 of Figure 6) with the shared policy.

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.

### VII. Related Work

Many works on serverless focus on minimising the latency of function invocations. Several of them tackle the problem by optimising function scheduling [9], [10].

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. [11], 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. focuses 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. [12] introduce a scheduling policy that governs the order of invocation processing depending on the availability of the resources they use. Abad et al. [13] present a package-aware technique that favours re-using the same workers for the same functions to cache dependencies. Suresh and Gandhi [14] show a scheduling policy oriented by resource usage of co-located functions on workers. Steint [15] and Akkus et al. [16] respectively present a scheduler based on game-theoretic allocation and on the interaction of sandboxing of functions and hierarchical messaging. Other scheduling policies exploit the state and relation among functions. For example, Kotni et al. [17] 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. Shillaker and Pietzuch [18] use state by supporting both global and local state access, aiming at performance improvements for data-intensive applications. Similarly, Jia and Witchel [19] associate each function invocation with a shared log among serverless functions.

An alternative approach, besides improving scheduling, is reducing the number of cold starts when launching new functions [1], [20]. In this direction, Shahrad et al. [21] introduce an empirically-informed resource management policy that mediates cold starts and resource allocation. Silva et al. [22] propose a solution based on process snapshots:

---

13In **data-locality**, min_memory has a slightly lower average than shared, but the latter has both lower variance and maximal latency.
when the user deploys the function, they generate/store a snapshot of the process that runs that function and, when the user invokes the function, they load/run the related snapshot. Similarly, on network elements, Mohan et al. [23] present an approach based on pre-creating networks and connecting them to (a pool of) containers “paused” after the network-creation step. At function invocation, they skip network-initialisation start-up times by attaching to one of the containers and completing the initialisation from there.

The main difference between these works and our proposal 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.

VIII. 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, and we used it to show that topology-aware scheduling is on par or outperforms hard-coded, vanilla OpenWhisk scheduling—e.g., tests that stress data locality gain considerable latency reductions.

Future work includes applying TAPP on different platforms, e.g., OpenLambda, OpenFAAS, and Fission. We 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. Regarding tests, we remark the general need for more platform-agnostic and realistic suites, to obtain fairer and thorough comparisons.

We also intend to formalise the semantics of TAPP, e.g., building on existing “serverless calculi” [24], [25]. This is a stepping stone to mathematically reason on scheduling policies and formally prove they provide desirable guarantees.

Finally, we would like to support DevOps in the optimization of their serverless applications by studying and experimenting with heuristics and AI-based mechanisms that profile applications and suggest optimal TAPP policies.

REFERENCES

[1] E. Jonas et al., “Cloud programming simplified: A berkeley view on serverless computing,” EECS Department, University of California, Berkeley, Tech. Rep., 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,” inProc. of NetDB, 2011.
[6] J. C. Anderson et al., CouchDB: the definitive guide: time to relax. " O’Reilly Media, Inc.", 2010.
[7] N. Eskandani and G. Salvaneschi, “The wonderless dataset for serverless computing,” inProc. of IEEE/ACM MSR, 2021.
[8] TAPP-based openwhisk extension. https://github.com/mattrent/openwhisk, Apr. 2022.
[9] A. Kuntsevich et al., “A distributed analysis and benchmarking framework for apache openwhisk serverless platform,” in Proc. of Middleware (Posters), 2018.
[10] M. Shahrad et al., “Architectural implications of function-as-a-service computing,” inProc. of MICRO, 2019.
[11] J. Sampé et al., “Data-driven serverless functions for object storage,” inProc. of Middleware, Las Vegas, Nevada: ACM, 2017.
[12] A. Banaei and M. Sharif, “Etas: Predictive scheduling of functions on worker nodes of apache openwhisk platform,” The Journal of Supercomputing, Sep. 2021.
[13] C. L. Abad et al., “Package-aware scheduling of faas functions,” inProc. of ACM/SPEC ICPE, ACM, 2018.
[14] A. Suresh and A. Gandhi, “Fnsched: An efficient scheduler for serverless functions,” inProc. of WOSC@Middleware, ACM, 2019.
[15] M. Stein, “The serverless scheduling problem and noah,” arXiv preprint arXiv:1809.06100, 2018.
[16] I. E. Akkus et al., “SAND: Towards high-performance serverless computing,” inProc. of USENIXATC, 2018.
[17] S. Kotni et al., “Faastlane: Accelerating function-as-a-service workflows,” inProc. of USENIX ATC, USENIX Association, 2021.
[18] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” inProc. of USENIX ATC, USENIX Association, 2020.
[19] Z. Jia and E. Witchel, “Boki: Stateful serverless computing with shared logs,” inProc. of ACM SIGOPS SOSP, Virtual Event, Germany: ACM, 2021.
[20] J. M. Hellerstein et al., Serverless computing: One step forward, two steps back, 2019.
[21] M. Shahrad et al., “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” inProc. of USENIX ATC, 2020.
[22] P. Silva et al., “Prebaking functions to warm the serverless cold start,” inProc. of Middleware, Delft, Netherlands: ACM, 2020.
[23] A. Mohan et al., “Agile cold starts for scalable serverless,” inProc. of HotCloud 19, Renton, WA: USENIX Association, Jul. 2019.
[24] M. Gabbrielli et al., “No more, no less - A formal model for serverless computing,” in Proc. of COORDINATION, ser. LNCS, Springer, 2019.

[25] A. Jangda et al., “Formal foundations of serverless computing,” Proc. of ACM on Prog. Lang., 2019.