Function-specific Scheduling Policies in Cloud-Edge Serverless Systems

Giuseppe De Palma[a], Saverio Giallorenzo[a,b], Jacopo Mauro[c], Matteo Trentin[a], Gianluigi Zavattaro[a,b]

[a]*Università di Bologna, Dipartimento di Informatica — Scienza e Ingegneria Mura Anteo Zamboni 7, 40126, Bologna, Italy*
[b]*INRIA, Sophia-Antipolis, France*
[c]*University of Southern Denmark, Department of Mathematics and Computer Science (IMADA), University of Southern Denmark, Campusvej 55, Odense M, 5230, Denmark*

Abstract

Cloud-edge serverless applications span multiple regions, e.g. local private networks and public cloud, and introduce the need to govern the scheduling of functions to satisfy their functional constraints or avoid performance degradation. For instance, functions may require to be allocated to specific private (edge) nodes that have access to specialised resources or to nodes with low latency to access a certain database to decrease the overall latency of the application. State-of-the-art serverless platforms do not support directly the implementation of topological constraints on the scheduling of functions. We address this problem by presenting a declarative language for defining topology-aware, function-specific serverless scheduling policies, called tAPP. Given a tAPP script, a compatible serverless scheduler can enforce different, co-existing topological constraints without requiring ad-hoc platform deployments. We prove our approach feasible by implementing a tAPP-based serverless platform as an extension of the Apache OpenWhisk serverless platform. We show that our extension naturally allows for cloud-edge deployments with topology-aware requirements which cannot be supported by standard deployments of vanilla OpenWhisk.

Keywords: Serverless, Function-as-a-Service, Cloud Optimisation, Topology-awareness.

1. 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]. Its main benefits are that users a) save time by delegating resource allocation, maintenance, and scaling to the platform and b) pay only for the resources that perform work, avoid idle costs.

AWS Lambda, Google Cloud Functions, and 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 cases, the platform manages the allocation of functions 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-region deployments, where the application uses cloud resources located in different regions, and in the cloud-edge continuum, where the cloud includes edge nodes with computing/memory/energy consumption constraints and only specific workers can access local resources.

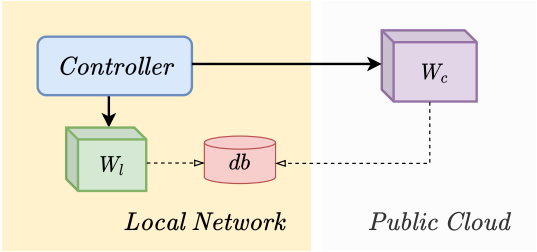

Figure 1: Example of function-execution scheduling problem.

To visualise the problem, consider an excerpt of the running example used in this article, reported in Figure 1. There, we have a simple serverless system composed of two workers. One worker, W_l, executes in the local network and the other, W_c, is in a public cloud. Both workers can execute functions that interact (represented by the dashed lines) with a database *db* deployed in the local network. When the *Controller* (acting as function scheduler) receives a request to execute the function, it must decide on which worker to execute it. To minimise the response time, the *Controller* should consider the different computational power of the workers as well as their current loads, which influence the time they take to execute the function. Moreover, the performance of the functions that interact with the database depends on the latter's access latency of the node they run on: W_l is close to the *db* and enjoys a faster interaction with it while W_c is farther away and can undergo heavier latencies.

The above observations motivate this work, where we provide

Email addresses: giuseppe.depalma2@unibo.it (Giuseppe De Palma), saverio.giallorenzo@unibo.it (Saverio Giallorenzo), mauro@imada.sdu.dk (Jacopo Mauro), matteo.trentin2@unibo.it (Matteo Trentin), gianluigi.zavattaro@unibo.it (Gianluigi Zavattaro)

[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/.

an answer to the following research question:

RQ. How can FaaS developers customise the scheduling policies of cloud-edge serverless platforms to improve function performance in locality-bound scenarios?

The relevance of an appropriate management of localities in geographically distributed applications has been already considered in state-of-the-art deployment technologies. For instance, Kubernetes allows for the specification of node affinities where the so-called topology keys can be used to give priority to nodes in a given geographical zone when a new pod must be scheduled. Nevertheless, such technologies (like Kubernetes) work at the level of container orchestration, which is a lower level of abstraction with respect to the Function-as-a-Service layer, as witnessed by FaaS platforms like Apache OpenWhisk which can, or cannot, use Kubernetes as an underlying deployment technology. Moreover, in the specific context of cloud-edge serverless platforms like FunLess [3], underlying powerful deployment technologies like Kubernetes are not used on purpose, in order to reduce the consumption of the limited resources within edge nodes.

For the above reasons, to the best of our knowledge, we consider our approach to be the first one which addresses the problem of managing customisable application dependent and topology-aware function scheduling policies at the FaaS level. We will discuss the above research question in Section 3, where we break it down in several main challenges to be addressed. In the remainder of this section, we substantiate our motivation via an example inspired by one of our industrial partners and describe our main contributions.

Motivating Example. We further clarify the concepts of locality-bound FaaS scheduling with a case study from our industry partners, which we use as an example throughout the article. We deem the case useful to help understand our contribution and clarify the motivation behind our work.

The case concerns a cloud-edge-continuum system to control and perform both predictive maintenance and anomaly detection over a fleet of robots in a production line. The system runs three kinds 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 them 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 kinds of tasks, which have no time constraints. Tasks iii) have resource-heavy requirements.

We depict the solution that we have designed for the deployment of the system in Figure 2. We consider three kinds of functions, one for each kind of tasks: critical functions (!), 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 $W_1, \ldots, W_i$) directly connected to the robots. Since machine-learning algorithms require considerable resources, which 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). The generic functions ◇ do not have specific, resource-heavy requirements, but they might need to access the database *db* in the local network. Hence, we schedule these preferably on the local cluster (W_{i+1},...,W_k) and use on-demand public-cloud workers when the local ones do not have enough free resources.

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*. $LocalCtl_1$ 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, the company requires that, instead of adopting a generic round-robin policy, the *Gateway* should forward critical functions (!) to $LocalCtl_1$, generic functions ◇ to one between $LocalCtl_1$ and $LocalCtl_2$, and cloud functions ☁ to *CloudCtl* (or to any other controller when the latter is not available).

Contributions. 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.

We propose an approach based on a new declarative language, called tAPP (Topology-aware Allocation Priority Policies), for writing topology-aware function-execution scheduling policies as *configuration* files. 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 downtimes due to system restarts to load new configurations.

More precisely, we start by presenting in Section 2 preliminary information about FaaS necessary to understand our contribution. We then detail in Section 3 the challenges to be addressed in order to answer to our research question concerning the customisability of function scheduling in cloud-edge serverless platforms. We present tAPP in Section 4. In Section 5, we validate our approach by implementing a serverless platform that supports tAPP-specified scheduling policies, as an extension of OpenWhisk where tAPP scripts and function-tag associations inform worker selection. In Section 6, we use our tAPP-based OpenWhisk version to implement a case study to evaluate how tAPP can actually support topology-aware function scheduling. We show that our prototype can capture typical functional scheduling requirements in cloud-edge deployments that cannot be supported by standard deployments of vanilla OpenWhisk.

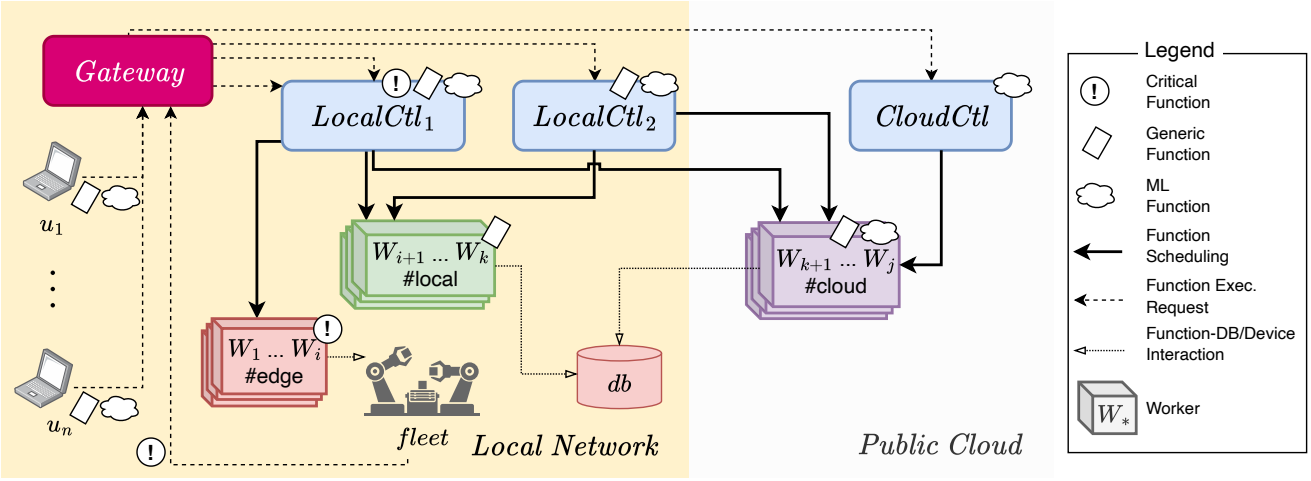

Figure 2: Representation of the case study.

Given that our prototype extends OpenWhisk, in Section 7 we quantify the overhead of our prototype w.r.t. the vanilla version of OpenWhisk through test cases drawn from *ServerlessBench* [4], a state-of-the-art benchmark suite for serverless platforms. Finally, Sections 9 and 10 respectively discuss the related literature and draw some concluding remarks.

This article mainly builds upon and substantially advances three previous conference contributions [5, 6, 7]. The first [5] introduces a prototypical version of tAPP, called APP, and proposes the tag-to-function mechanism, and implements it in OpenWhisk. The second, short paper [6] introduces the idea of tAPP to handle topological information. The third [7] details the extension of OpenWhisk to support tAPP-based policies and benchmark some of its performances using simple functions – the OpenWhisk artefact has a related journal publication [8].

This article expands these foundations along several directions:

- Section 4 offers a revised and extended presentation of the (t)APP language, including a systematic account of the technical details behind our tAPP-based OpenWhisk extension.
- Section 5 introduces new Infrastructure-as-Code solutions designed to streamline and automate deployment of our tAPP-based extension of OpenWhisk, thereby enhancing usability and fostering practical adoption.
- Section 6 details the impact of tAPP on the locality-bound scenario described in our motivating example.
- Section 7 quantifies the overhead of the tAPP-based extension of OpenWhisk w.r.t. the vanilla version through test cases drawn from *ServerlessBench* [4], a state-of-the-art benchmark suite for serverless platforms.

2. Preliminaries

Before detailing our solution, we describe the preliminary information necessary to understand its context and technicalities. First, we outline the problems that motivate our research—as found in the literature. Then, we give an overview of the OpenWhisk Serverless platform, which we use to implement a prototype of our solution to the function scheduling problem.

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—like AWS Lambda [9], Google Cloud Functions [10] or Microsoft Azure Functions [11]—is responsible to schedule functions on its workers, to control the scaling of the infrastructure by adjusting their available resources, and to bill its users on a per-execution basis.

When instantiating a function, the provider has to create the appropriate execution environment for the function. Containers [12] and Virtual Machines [13] are the main technologies used to implement isolated execution environments for functions. How the provider implements the allocation of resources and the instantiation of execution environments impacts the performance of the function execution. If the provider allocates a new container for every request, the initialisation overhead of the container would negatively affect both the performance of the single 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* [2]. Resource allocation also includes I/O operations that need to be properly considered. For example, Wang et al. [14] report that a single function in the AWS serverless platform has an average network bandwidth of 538Mbps; an order of magnitude slower than a modern hard drive (the authors report similar results from Google and Azure). These performances result from bad allocations over I/O-bound devices, which one can

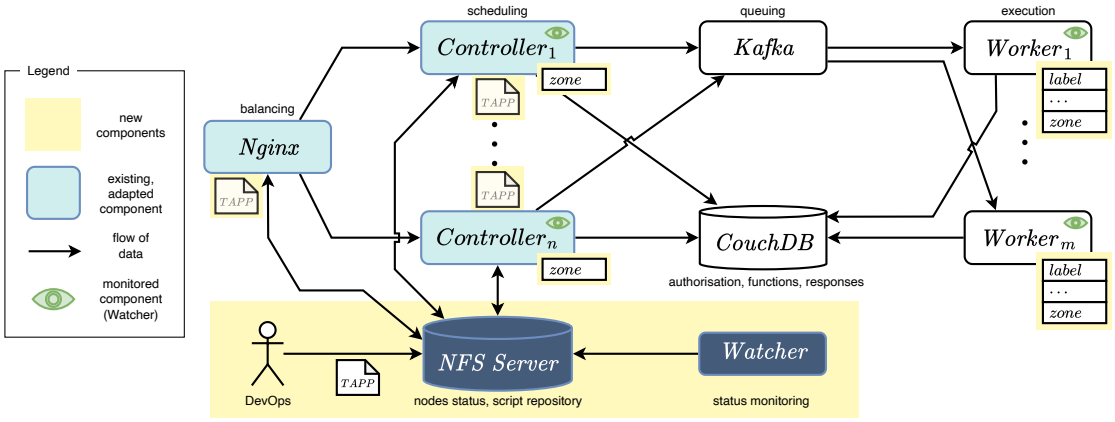

Figure 3: Architectural view of our OpenWhisk extension.

reduce following the principle of *session locality* [2], i.e., taking advantage of already established user connections to workers. Another important aspect to consider to schedule functions, as underlined by the example in Figure 1, 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. Data locality has been subject of research in neighbouring Cloud contexts [15, 16].

Apache OpenWhisk. The main proprietary serverless platforms are AWS Lambda, Google Cloud Functions, and Azure Functions. These solutions do not provide source code one can build upon and offer limited documentation on their inner workings. Fortunately, open-source alternatives are gaining traction in the serverless market. At the moment, popular ones are Apache OpenWhisk, OpenFaaS, and OpenLambda. Since these platforms have similar architectures [17] 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 3 a scheme of the architecture of OpenWhisk. For compactness, we include in Figure 3 the changed and new elements introduced by our extension, described in Section 5. Here, we focus only on the original components and functionalities of OpenWhisk.

In Figure 3, from left to right, we first find *Nginx*, the gateway and load balancer of incoming requests. *Nginx* forwards each request to one of the *Controller*s in the current deployment.

The *Controller*s decide on which of the available computation nodes, called *Worker*s[3], to schedule the execution of a given function. *Controller*s and *Worker*s indirectly interact via Apache *Kafka* [18] and *CouchDB* [19], which respectively handle the routing and queueing of execution requests and manage the authorisations and the storage of functions and of their outputs/responses.

[3]OpenWhisk's documentation uses the more specific term "invokers".

*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, 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*. 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.

3. Challenges of Customisable Serverless Scheduling Policies

Answering our research question requires navigating a space of interrelated design decisions, each introducing a distinct challenge. In this section, we characterise those challenges and identify the design principles that inform our solution.

The central design tension hinges on the fact that customisable scheduling policies require exposing enough information about the worker topology to let developers reason about placement, yet the serverless paradigm deliberately abstracts that topology away.

The design of tAPP resolves this tension through a hybrid approach that gives developers the minimal topological information needed to express constraints, while leaving providers free to manage their infrastructure transparently.

3.1. Expressiveness vs. Abstraction (C1–C4)

We identify four challenges related to the tension between expressiveness and abstraction. The first two challenges concern the information that a scheduling policy must be able to express.

C1. Express the relationship between functions and the workers that can run them.

C2. Define the priority to select the worker that shall run a function instance.

tAPP's answer to C1 and C2 is to rely on a *label-based* abstraction. Indeed, rather than naming physical nodes, tAPP

provides developers with labels to reason abstractly about groups of functions and groups of workers. In this way, developers can decouple functions from their policies and providers can decouple execution nodes from worker labels without exposing infrastructure details. Practically, tAPP adopts the de-facto standard language for Infrastructure-as-Code (IaC) tools, YAML, as the representation format of custom scheduling policies [20]. Following the IaC philosophy, tAPP policies become versioned configuration files that coexist with the application source, require no platform restart to take effect, and can be maintained by DevOps practitioners without knowledge of platform internals.

We deem this label-based design appropriate to balance between expressiveness and abstraction whereas alternatives, such as runtime annotations baked into function metadata, imperative policy scripts, or hard-coded platform-specific routing rules, would entail a tighter coupling of scheduling logic with the platform codebase or breaking the separation between function developers and platform administrators.

Another important concern regards the elasticity of cloud-edge resource management, which policies should abstract upon. Workers can join and leave a platform at runtime, hence, policies expressed over a fixed enumeration of worker identities risk becoming stale. This issue introduces the challenge:

> **C3.** Express scheduling policies that refer to a dynamically modifiable set of workers.

To tackle this challenge, besides targeting individual nodes, tAPP allows labels to identify *worker sets*. Thanks to worker sets, the membership of a set can change at runtime without interfering with policies, which remain stable. While worker sets extend the concept of label-based addressing to groups, it provides developers with the same abstraction of single worker selections—introducing minimal cognitive overhead.

Although worker sets capture cloud-edge elasticity, a single, flat pool of workers cannot capture the heterogeneous characteristics of multi-region deployments, where distinct data centres differ in hardware capabilities, network topology, and access to local storage. In essence, we need to capture cases where the same function class must follow different scheduling policies in different regions. This aspect gives rise to the challenge:

> **C4.** Express scheduling policies that adapt to the different characteristics of the regions in cloud-edge scenarios.

tAPP's answer to C4 is to model regional differences into zones, as a partition of the infrastructure, each managed by one or more controllers. Associating a controller with a zone allows the same policy tag to project different scheduling rules onto different parts of the deployment. Thus, zones serve as abstractions for the physical topology layer, while worker labels and sets constitute the abstractions for the logical topology layer. The interplay between worker labels/sets and zones determines which workers a controller may access under a given policy.

Crucially, C1–C4 are not independent: the label abstraction (C1) must work over dynamic sets (C3) that may be zone-scoped (C4), while selection priorities (C2) must be expressible both within a zone and across zones for graceful degradation. tAPP unifies these requirements in a single, composable declarative language, as detailed in Section 4.

3.2. Feasibility and Benefit (C5–C6)

While C1–C4 address the challenges of designing a language able to find a balance between abstraction and expressiveness, there are two challenges of practical nature: demonstrating that the proposed abstractions can be realised on a real platform and that doing so genuinely improves scheduling outcomes. We see these as validation challenges.

> **C5.** Show that assembling a serverless platform able to enforce custom scheduling policies is feasible.

> **C6.** Show that custom serverless scheduling policies increase performance in locality-bound scenarios and introduce negligible overhead on general function scheduling.

We address C5 by extending Apache OpenWhisk with a tAPP-aware scheduling layer (Section 5). The key architectural challenge is that OpenWhisk's scheduling algorithm is hard-coded and topology-oblivious. Our extension tackles the problem with new components to intercept and redirect scheduling decisions without forking the core platform or requiring per-deployment source modifications. Hence, besides showing that one can implement tAPP's semantics over a mainstream, production-grade FaaS platform, our implementation shows that doing so does not require a major overhaul of the platform's architecture.

We address C6 through two complementary evaluations: a cloud-edge case study (Section 6) that shows that tAPP reliably enforces topology constraints (that vanilla OpenWhisk cannot guarantee) and that tAPP can also increase the performance of locality-bound functions, and an overhead analysis (Section 7) using the ServerlessBench [4] benchmark suite, which quantifies the (negligible) cost of the added routing logic against vanilla OpenWhisk.

4. Topology-aware Serverless Scheduling

The Topology-aware Allocation Priority Policy (tAPP) language is our answer to challenges C1–C4 identified in Section 3: how to express, in a declarative and platform-agnostic way, the relationship between functions and workers (C1), the priority of worker selection (C2), dynamic membership of worker pools (C3), and the heterogeneous characteristics of multi-region zones (C4). The central design decision is a hybrid abstraction where labels expose the minimum topological information necessary for developers to write placement constraints, while keeping providers free to manage the underlying infrastructure. In this section, we present the tAPP language—its syntax, semantics, and composability properties—and show how its constructs are both necessary and sufficient to capture the scheduling requirements of the case study introduced in Section 1.

Essentially, tAPP relies on *policy tags* that associate functions to scheduling policies. A tag identifies a policy (e.g., we can use a

policy_tag∈*Identifiers* ∪ {`default`} *label*∈*Identifiers* $n \in \mathbb{N}$

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

Figure 4: The syntax of tAPP.

tag "critical" to identify the scheduling behaviour of the critical (!) functions of our case study, cf. Section 1) and it marks all those functions that shall follow the same scheduling behaviour (e.g., marking as "critical" any function that falls into that category).

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 2). 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 access. 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 2.

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., due to their resource quotas).

4.1. The tAPP language

tAPP scripts are YAML [20] files—Figure 4 reports tAPP's syntax. The basic entities considered in 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 formed with the accepted character set as defined in [20].

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 `workers` 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 the script shown in Figure 5. 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 Figure 5, 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` to represent all worker labels. The `strategy` is the `platform`'s default—in our prototype (cf. Section 5), this strategy corresponds to a selection algorithm, hinted in Section 2, which 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[4]). 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.

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

[4]The identifier of the primary worker results from the modulo between the hash and the number of workers. In case the primary is invalid, we find the secondary workers by increasing the function hash with the step-size and repeating the modulo operation.

```
- 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
```

Figure 5: Example of a tAPP script.

when it reaches the `50%` of capacity. Here the notion of capacity depends on the implementation (e.g., our OpenWhisk-based tAPP implementation in Section 5 uses information on the memory 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 use the one of the enclosing block or, if the latter is missing too, the default one.

Summarising, given a policy tag, the scheduler follows the policy defined in the `strategy` option to select the corresponding blocks. A block includes three parameters:

- `workers`: which 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;
- `strategy`: defines the policy of item selection at the levels of *policy_tag*, *workers* block, and workers `set`s. APP currently supports three strategies that select elements: `random`ly, via a uniform distribution; `best_first`, per top-to-bottom order of appearance; `platform`, via the default logic of the platform (only for worker selection, cf. Figure 4).
- `invalidate`: specifies when a worker (label) cannot host a function's execution. All invalidate options include, as preliminary condition, the unavailability of a worker. 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 we exhaust all blocks. In the latter case, we enact the `followup` behaviour. Current `invalidate` options are: `overload`, insufficient resources to host the function[5]; `capacity_used`, memory usage exceeds the set limit; `max_concurrent_invocations`, the number of buffered concurrent invocations exceeds the set limit.
- `followup`: specifies the policy enacted when all the blocks in a policy are invalid. The supported followup options are: `fail`, drop the scheduling of the function; `default`, apply the `default` tag.

Since the `default` block is the only "backup" tag used when all workers of a custom tag cannot execute a function, the `followup` value of the `default` tag is always set to `fail`.

Besides the above elements, to detail topological constraints of function execution scheduling, we have 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.

The *controller* clause has the optional parameter `topology_tolerance`. When deploying controllers and workers, users can label them with the topological *zone* they belong in[6]. 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` parameter specifies what workers an alternative controller can use. Specifically, `all` is the default

[5]The kind of resources that determine if a worker is `overload`ed depends on the APIs provided by the serverless platform. For example, in our prototype, we consider a worker label `overload`ed when OpenWhisk declares the related worker "unhealthy", considering its available memory and CPU load.

[6]Zone labels do not appear in tAPP scripts, which only specify co-location constraints, to constrain allocations on workers in the same zone of a given controller. The implementation relies on zone labels to enforce tAPP's constraints.

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; `none` forbids the forward to other controllers. As an example, we could take advantage of the topology zones and rewrite the previous tAPP script from Figure 5 for the couchdb_query tag as

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

which guarantees that the function will always be executed on the workers in the same zone of the database. Lastly, tAPP lets users express a selection strategy for policy blocks. This is represented by the optional *strategy* fragment of the *tag* rule (and the *c_strategy* syntactic category found in Figure 4). 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 uniformly distributing requests among the available controllers.

4.2. Case Study

As a final illustration of the tAPP language, we show and comment on the salient parts of a tAPP script—reported in Figure 6—that captures the scheduling semantics of the case in Figure 2.

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 2), 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` (cloud) functions. We define `CloudCtl` as the controller and consider all #cloud workers ($W_{k+1}, \ldots, W_j$ in Figure 2) as executors. 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 2)

```
1  - critical:
2    - controller: LocalCtl_1
3      workers:
4        - set: edge
5          strategy: random
6    followup: fail
7  - machine_learning:
8    - controller: CloudCtl
9      workers:
10       - set: cloud
11     topology_tolerance: same
12   followup: default
13 - default:
14   - controller: LocalCtl_1
15     workers:
16      - set: internal
17        strategy: random
18      - set: 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 6: A tAPP script that captures the semantics of the case study in Section 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).

4.3. How tAPP addresses challenges C1–C4

The constructs introduced in this section collectively resolve challenges C1–C4. Worker labels and policy tags address C1 and C2, where labels abstract physical nodes into named entities, and `strategy` parameters encode selection priorities without hard-coding routing logic. Worker sets address C3 by decoupling label identity from runtime membership so that the scheduler evaluates set contents at invocation time and the policy remains valid as workers join or leave. Zones and controllers address C4. Specifically, zones support partitioning the infrastructure, each managed by one or more controllers, allowing the same policy tag to project different scheduling rules onto different deployment regions, as illustrated by the case-study script in Figure 6. The full tAPP script of the case study confirms that the language is sufficient, i.e., each functional requirement identified in Section 1—strict edge placement for critical functions, cloud-only placement for machine-learning functions, and graceful degradation for generic functions—can be expressed at the FaaS level.

5. tAPP in OpenWhisk

Challenge C5 requires demonstrating that a real serverless platform can enforce tAPP-specified scheduling policies without incurring disruptive re-deployments. Vanilla OpenWhisk

presents three concrete obstacles to this goal: *(i)* its scheduling algorithm (co-prime hashing, cf. Section 2) is hard-coded and topology-oblivious; *(ii)* its gateway (Nginx) forwards requests by round-robin with no awareness of policy tags; *(iii)* the platform has no mechanism to propagate or reload scheduling configuration at runtime. We address these obstacles through a targeted extension of OpenWhisk that intercepts scheduling decisions at the gateway and controller layers, introduces a tAPP-aware function-distribution model, and supports live reloading of tAPP scripts without downtime. In our implementation, to manage the deployment of components, we pair OpenWhisk with the de-facto standard container orchestrator Kubernetes[7].

A key architectural decision is *where* in the OpenWhisk pipeline to intercept scheduling decisions. Two candidate interception points exist: the gateway (Nginx), which receives all incoming invocation requests, and the controller, which performs worker selection. Intercepting only at the gateway would enable tag-based routing to controllers but would leave worker selection topology-oblivious. Intercepting only at the controller would enable topology-aware worker selection but would prevent routing different function classes to different controller groups. Thus, we intercept at *both* levels. At the gateway level, we extend Nginx to read the function tag from the invocation request and forward it to the appropriate controller group as specified by the tAPP script. At the controller level, we replace the co-prime algorithm with a tAPP-aware worker-selection procedure that evaluates the applicable policy block at invocation time. This two-level design is necessary and sufficient to enforce the full expressiveness of tAPP: tag-level `controller` blocks are resolved at the gateway, and block-level `workers` specifications are resolved at the controller.

Figure 3 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 3).

The modifications mainly regard having Nginx and Controllers retrieve and interpret both tAPP scripts and nodes' status data to forward requests to the appropriate controllers and workers. The Watcher monitors the topology of the Kubernetes cluster and reconstructs its overall status into the NFS Server, which provides access to tAPP scripts and status data to the other components. Below, we detail the changes to the existing OpenWhisk components and the two new services, and how the proposed system supports live-reloading of tAPP configurations. We conclude by describing the deployment of the modified OpenWhisk platform.

5.1. OpenWhisk Gateway and Controller

OpenWhisk's gateway (Nginx) forwards requests to the available controllers, following a hard-coded round-robin policy. To support tAPP, we used *njs*[8]—a subset of the JavaScript language that Nginx provides to extend its functionalities—to inject a plug-in that analyses requests passing through the gateway, extract the tags from the request parameters, and compare them against a given tAPP script.

Since Nginx manages all inbound traffic, to keep the footprint of the plug-in small, we only interpret tAPP scripts and load the mappings when requests carry some tags, using caching to limit retrieval downtimes. 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.

The second modification to the existing OpenWhisk components regard the controller, which we extended (as a Scala class called `ConfigurableLoadBalancer` that extends the original `LoadBalancer` one) with a parser and an engine to interpret tAPP scripts.

5.2. New Services: Watcher and NFS Server

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 3), and stores the mapping into the *NFS Server*.

As shown in Figure 3, Nginx uses the output of the Watcher to forward function-execution requests to controllers. In this way, tAPP scripts define which controller to target without the need to specify a pod identifier, but rather use a label (e.g., `CloudCtl` in Figure 6). 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.

5.3. Topology-based Worker Distribution

A second architectural decision concerns how controllers partition the set of available workers with multiple controllers. This partition determines whether a controller may schedule functions on workers in zones other than its own—a consequential choice that affects both resource usage and zone-boundary enforcement. We identify four policies that span the relevant design space:

default Each controller uses the full set of registered workers, with no zone awareness. This policy replicates vanilla OpenWhisk behaviour and serves as the baseline.

min_memory Each controller uses its own zone's workers first, but may spill to workers in other zones when local capacity is exhausted. This policy maximises resource usage at the cost of occasional cross-zone scheduling.

[7] `https://kubernetes.io/`.
[8] `https://nginx.org/en/docs/njs/index.html`.

isolated Each controller uses exclusively the workers in its own zone. Zone boundaries are strictly enforced, guaranteeing that no function escapes its designated region, at the cost of potential under-usage if one zone is idle while another is saturated.

shared Workers are partitioned by explicit worker-set labels rather than by zone membership, allowing fine-grained sharing across zone boundaries as prescribed by the tAPP script.

5.4. 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.

This feature derives from our design, where the NFS Server stores the reference policies script, while Nginx and the controllers keep local copies thereof. Updating the NFS copy triggers a notification for Nginx and the controllers to invalidate their local copy and retrieve the new one.

5.5. Deploying tAPP-based OpenWhisk

A practical concern for any platform extension is whether it can be deployed and updated without manual, error-prone configuration steps. We address this issue with an Infrastructure-as-Code approach: the entire tAPP-extended OpenWhisk deployment—including cloud resource provisioning, platform configuration, and service orchestration—is fully automated, version-controlled, and publicly available.[9]

The automation pipeline covers three layers. Resource provisioning uses Terraform to allocate virtual machines and network topology for a target deployment, such as the multi-zone cloud-edge configuration of Section 1. Platform configuration uses Ansible to install and wire together the tAPP-extended OpenWhisk components—the modified Nginx gateway, the tAPP-aware controllers, the Watcher, and the NFS server—across the provisioned nodes. Service packaging uses Helm to bundle the extended platform for Kubernetes, enabling deployment on existing clusters without dedicated virtual machines. Together, these three layers allow a complete tAPP-based deployment to be reproduced from a single configuration repository, with no manual steps beyond specifying the target topology.

5.6. How tAPP in OpenWhisk addresses C5

The implementation described in this section demonstrates the feasibility of C5. We implemented tAPP's semantics on an existing, mainstream serverless platform through well-scoped extensions at the gateway and controller layers. Notably, the Watcher-NFS architecture resolves the consistency challenge inherent in dynamic topologies, where the NFS server maintains a single authoritative copy of each tAPP script, local caches at Nginx and each controller limit retrieval latency, and cache invalidation is triggered by explicit notifications on script changes, ensuring all components converge to the same policy without requiring a global synchronisation barrier.

[9] Source code and deployment scripts: `https://github.com/giusdp/ow-gcp`; pre-built container images: `https://hub.docker.com/r/mattrent/ow-controller`.

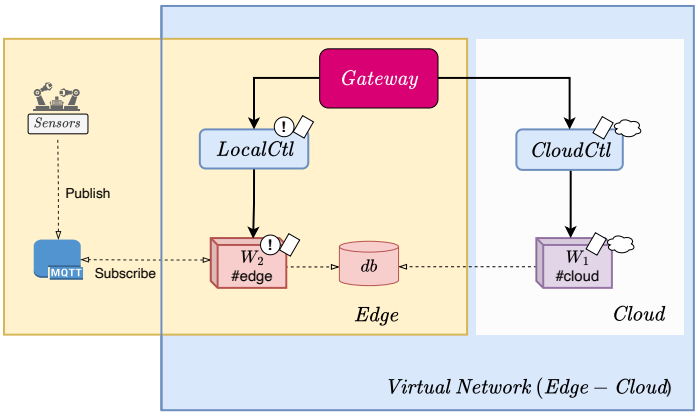

Figure 7: Case study architecture: the services are separated into two zones, named *Edge* and *Cloud*, and connected using two different networks, a local network corresponding to the *Edge* zone and a virtual private network used for the OpenWhisk cloud-edge deployment.

6. Case Study Implementation

Challenge C6 has two dimensions: increase of performance in locality-bound scenarios and negligible overhead on general function scheduling.

We address the first dimension in this section through a controlled cloud-edge case study drawn from the literature, using the IoT pipeline described in Section 1 as the experimental workload. We evaluate two deployments—vanilla OpenWhisk and our tAPP-extended prototype—against the same set of functional scheduling requirements, using the fraction of invocations that land on a topology-appropriate worker as the primary success criterion. We quantify the overhead of tAPP's routing layer in Section 7.

6.1. Case Study Architecture

The case study instantiates the architecture presented in Section 1 as a serverless cloud-edge reinterpretation of a power-transformer anomaly detection system [21, 22], depicted in Figure 7. The system monitors a fleet of transformers, each equipped with six accelerometers sampling at 10 kHz; every minute, one second of sensor data is collected, two features per sensor are extracted (FCA and DET), and the resulting feature vectors are classified using machine learning. We implement the workflow as a pipeline of three serverless functions—**data-collection**, **feature-extraction**, and **feature-analysis**—whose distinct resource and locality requirements make this an effective testbed for tAPP:

- **data-collection** contacts an MQTT broker [23] and subscribes to six sensor topics, collecting and storing the raw data in a local database. The broker is accessible only from within the local network (*Edge* zone) to prevent external attacks, making co-location with the broker a *hard* placement constraint that standard OpenWhisk cannot express.

```
- default:
  - workers:
      - set:
- MQTT:
  - controller: LocalCtl
    workers:
      - set: edge
    topology_tolerance: none
    followup: fail
- DB:
  - workers:
    - wrk: W_2
      invalidate: capacity_used 50%
    - wrk: W_1
    strategy: best-first
- Cloud:
  - controller: CloudCtl
    workers:
      - set: cloud
    topology_tolerance: none
    followup: fail
```

Figure 8: Script used in the tAPP-based use case deployments.

- **feature-extraction** queries the local database and extracts the FCA and DET features. The function has a soft data-locality constraint: execution on a local worker is preferred to minimise database access latency, but cloud workers are acceptable when local capacity is exhausted.
- **feature-analysis** classifies the extracted feature vectors using machine learning. Its resource-heavy profile makes it a candidate for cloud execution, independently of locality considerations.

These three functions each exercise a qualitatively different scheduling requirement: a hard zone constraint (C4), an ordered locality preference (C2), and a resource-driven placement (C1), exploiting the full expressiveness of tAPP.

6.2. Experimental Setup

We deploy the platform across two zones, as depicted in Figure 7: a *cloud* zone hosting the Kubernetes master node, one OpenWhisk controller, and one worker; and an *edge* zone hosting the MQTT broker, the local database, one OpenWhisk controller, and one worker. The network topology enforces the key placement constraint of the case study: the MQTT broker is reachable only from within the edge zone, while the database is accessible from both zones. This asymmetry is what makes **data-collection** a hard edge-only constraint and **feature-extraction** a soft locality preference. The deployment uses six VMs (three per zone, 2 vCPUs and 4 GB RAM each), connected via a virtual private network spanning the full Kubernetes cluster and a local network scoped to the edge zone—Cloud: Google Cloud Platform, `europe-west1-b`; edge: DigitalOcean, Frankfurt.

To reproduce the temporal profile of the target application, we simulate sensor data via a Python script co-located with the MQTT broker, generating input at the same frequency as the real system (one pipeline invocation per minute, sensor push rate of 10 kHz per sensor). Since the specific machine learning model used for classification does not affect the scheduling evaluation, **feature-analysis** is implemented as a compute-bound stub performing matrix multiplications of equivalent cost, returning a predetermined response. This controlled substitution isolates scheduling behaviour from model-specific variance, ensuring that placement decisions—not computation time—drive the evaluation results.

6.3. Qualitative Results

Under vanilla OpenWhisk, correct pipeline execution is not guaranteed by construction. The co-prime scheduling algorithm assigns each function a home worker by hashing its identifier modulo the number of registered workers, with worker identifiers assigned randomly at deployment time. This implies that, in our deployment, **data-collection** can be randomly assigned to the cloud worker, which cannot reach the MQTT broker. When this event occurs, every invocation of the function fails, invalidating the entire pipeline. Across 10 repeated deployments, we observed this failure in 8 runs[10]—confirming that correct placement is a matter of chance rather than a constraint consistently enforceable under vanilla OpenWhisk.

The tAPP deployment eliminates this non-determinism by making placement constraints explicit in the script of Figure 8. Each policy tag resolves a distinct scheduling requirement:

- **data-collection** () is tagged *MQTT*, routing all invocations through `LocalCtl` and restricting execution to the edge zone. Setting `topology_tolerance` to `none` enforces this hard constraint—forwarding to any other controller is forbidden—resolving the hard zone placement requirement (C4) and guaranteeing that the function always executes on a worker co-located with the MQTT broker.
- **feature-extraction** () is tagged *DB*, with an ordered worker list that prefers the edge worker—closest to the database—until its capacity exceeds 50%, at which point the cloud worker is used. No controller selector is specified, so any controller may dispatch this function; the ordering and invalidation condition together express the soft locality preference (C2) while preserving availability under load.
- **feature-analysis** () is tagged *Cloud*, routing all invocations through `CloudCtl` with `topology_tolerance` `none`. This configuration enforces a hard cloud-only constraint (C4), ensuring that resource-intensive ML tasks never consume edge capacity, independently of load conditions.

The result is a deployment in which all three placement requirements are satisfied deterministically across all invocations and all repeated runs, in contrast to the probabilistic and uncontrollable behaviour of vanilla OpenWhisk.

[10] We remark that, while we were expecting a failure in 50% of the cases, the chosen worker from the vanilla scheduler was almost always the cloud worker. It is beyond the scope of this paper to ascertain if this was due to a statistical anomaly or to OpenWhisk's internal decisions (e.g., decisions based on a hash from the namespace and the function name) that favour particular deployment choices.

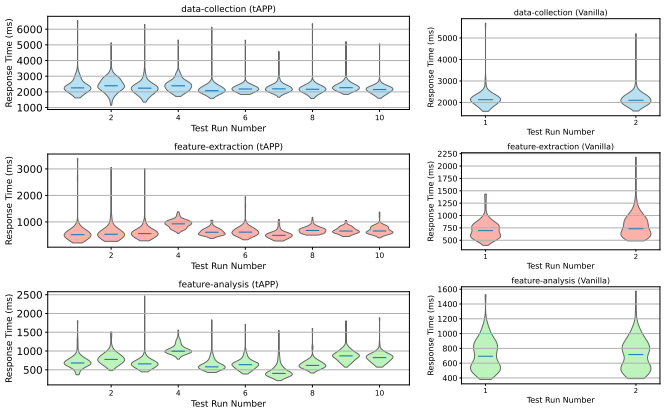

Figure 9: tAPP-based (left) and vanilla (right) OpenWhisk latencies.

6.4. Quantitative Results

We recorded the performance of the system using the vanilla and tAPP-based OpenWhisk variants. The starting versions for both tAPP-based OpenWhisk and the vanilla OpenWhisk in the experiments originate from commit `aa7e6e2` of the official Apache OpenWhisk repository.[11] We add a logging mechanism to vanilla OpenWhisk to record the scheduling times.

We repeated the deployments of both versions of the platform 10 times. Once deployed, we tested the case study by performing 100 sequential invocations of the pipeline with an interval of 1 minute between each invocation as described in the case study, and recording the latencies of the function invocations, i.e., the time passed between a request to a function and a response (successful or failing), and the scheduling time, i.e., the time the scheduler takes to pick a worker for the function invocation. We performed these tests with Locust[12], a load testing tool.

We observed that vanilla OpenWhisk picks the cloud worker as the main worker for the **data-collection** function a total of 8 out of 10 times. In these cases, the pipeline fails to connect to the MQTT broker and the **data-collection** stalls until the timeout mechanism is triggered after 60 seconds. The other 2 functions are not invoked as the function pipeline stops at its first step. The pipeline invocation under the tAPP-based variant has 100% success rate, with the **data-collection** function being always allocated to the edge worker, resulting in consistent performance.

Figure 9 shows the violin plots related to the response time of the 3 functions in all the 10 tAPP-based OpenWhisk deployments and, for fairness reasons, the 2 functioning vanilla OpenWhisk deployments. The response times cluster around 2000 ms for **data-collection**, less than 1000 ms for **feature-extraction**, and between 500 and 1000 ms for **feature-analysis** for both the tAPP and vanilla OpenWhisk. We find outliers in both scenarios due to the high response times of first-time functions invocation cold starts.

A relevant difference between tAPP and vanilla OpenWhisk can be observed on the violin plots of the **feature-extraction** and **feature-analysis** functions (see Figure 9, lines 2 and 3): differently from tAPP, OpenWhisk shows a bimodal distribution of the response time of these two functions. This difference follows from the different adopted scheduling policies. The deterministic policy of tAPP schedules **feature-extraction** preferably on the edge worker (when its used capacity is below 50%) and places **feature-analysis** functions always on the cloud worker. On the contrary, OpenWhisk does not give preference to any worker and changes the initially selected worker when it is overloaded. The bimodal distributions show that both workers (cloud and edge) are used for both functions during the experiments.

For completeness, we report in Tables 1 and 2 the aggregated mean, median, tail latency (i.e., 99 latency percentile), and standard deviation of the tAPP-based and the 2 functioning vanilla OpenWhisk deployments—for compactness, we use the icons ⓘ, ◇, ☁ to resp. represent the data-collection, feature-extraction, and feature-analysis functions. As can be seen, in case the deployment of vanilla OpenWhisk did allow the scheduling of the **data-collection** function, the response times were similar for the **data-collection** and the **feature-analysis** functions. On the contrary, the performance of the **feature-extraction** function appears improved in the tAPP case. This improvement follows from the fact that this function accesses the database which is deployed in the edge zone; in other terms, **feature-extraction** is a locality-bound function because it expects to interact with a component (the database) which is physically located in a specific zone. Our experiments show that the tAPP scheduling policy, which gives preference to the edge worker for the placement of the **feature-extraction** function, has the effect of improving the function's performance. This result addresses the first dimension of challenge C6.

	Mean	Median	Tail	Std. Dev.
ⓘ	2272.18	2231.49	2542.54	345.29
◇	652.08	628.70	981.24	90.94
☁	728.71	715.58	829.69	112.88

Table 1: Latency of the tAPP-based OpenWhisk deployments (ms).

	Mean	Median	Tail	St. D.
ⓘ	2148.38	2113.63	2522.17	360.85
◇	748.83	727.21	1096.80	141.42
☁	706.95	690.70	975.16	111.48

Table 2: Latency of the functioning vanilla OpenWhisk deployments (ms).

Finally, Table 3 (resp. Table 4) reports the mean, median, tail, and standard deviation of the scheduling time of the tAPP-based (resp. vanilla) deployments, i.e., the time it takes the controller to schedule a function from when the function invocation is received to when the scheduling logic is executed and a worker is chosen. From the results, we can conclude that the performance of vanilla OpenWhisk and tAPP are comparable—the average scheduling time is below 2 ms for both the tAPP and vanilla OpenWhisk deployments.

[11] https://github.com/apache/openwhisk/commit/aa7e6e2af196ac017ae4b9ea36656bec868a9931

[12] https://locust.io/

	Mean	Median	Tail	Std. Dev.
ⓘ	1.42	1.29	2.60	0.57
◇	1.62	1.43	4.23	0.93
☁	1.68	1.68	2.85	0.64

Table 3: Scheduling times from tAPP deployment (ms).

	Mean	Median	Tail	Std. Dev.
ⓘ	1.78	1.54	3.75	1.22
◇	2.0	1.74	3.82	1.20
☁	1.95	1.73	3.82	1.02

Table 4: Scheduling times from Vanilla deployment (ms).

7. Overhead Analysis

We now address the second dimension of challenge C6, by evaluating the overhead of our tAPP extension w.r.t. vanilla OpenWhisk. To this aim, we run a set of experiments using a benchmark suite for serverless platforms, called *Serverless-Bench* [4]. Running the tests, we measure both the scheduling time (the time taken to pick on which worker to allocate a function) and the request-reply latency of functions.

ServerlessBench consists of 12 test cases exploring several serverless metrics, like communication efficiency and startup latency (cold starts). Of these 12 test cases, 6 apply to OpenWhisk: 3–6, 9, and 10. Of these, case 4 consists of 4 subtests, each with different example applications requiring different resources (i.e., databases, Alexa devices). Due to these special requirements, we exclude case 4 from our analysis and focus on the remaining 5.

We run cases on both vanilla OpenWhisk and tAPP-based OpenWhisk deployments. For the tAPP variant, we use a simple tAPP script with default settings that make the tAPP variant behave like vanilla OpenWhisk. In this way, we perform a same-settings comparison of the two versions of the platform. We run each case 5 times to obtain consistent data.

Case 3. This case focuses on function composition, obtained by invoking a long pipeline of functions. The functions in the pipeline are instances of the same one, written in JavaScript, which increments by one the input value and returns it. We create a "sequence" (the OpenWhisk built-in pipeline construct) with 50 of these functions so that the platform invokes them in sequence, passing the output of one function as the input of the next one. We invoke the pipeline 20 times to have a total of 1000 invocations in a single test run. We report the latencies and scheduling times in Figure 10.[13] From the results, the time required to pick a worker is essentially the same for both versions of OpenWhisk. Similarly, the total latency of the invocations—each spanning the entire sequence and corresponding to the time required to execute all the functions in the pipeline—is comparable, with the majority falling between the 17 to 24 seconds range. In particular, vanilla OpenWhisk shows a higher fluctuation in the total latencies with a higher standard deviation of 2435.11 ms (and mean of 20096.86 ms), while tAPP had a more stable performance (standard deviation of 1139.1 ms and mean of 18901.26 ms).

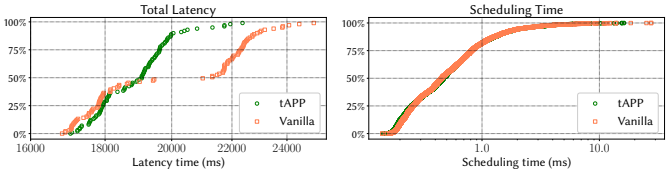

Figure 10: Test Case 3 latencies (left) and scheduling time (right).

Case 5. This case focuses on data transfer costs, using a sequence of 2 functions where a file of 32KB is passed to the first function, which passes it to the second one. We invoke the sequence 500 times to have a total of 1000 invocations in a single test run, as in the previous case. We show the latencies and scheduling times in Figure 11. Consistently with the observations of case 3, we have negligible differences in scheduling times between the two platforms, and the total latencies are stable between 500 ms and 2000 ms for both platforms, with some outliers reaching 4000 ms due to cold starts. Regarding invocation latency, vanilla OpenWhisk has a slightly higher standard deviation of 357.27 ms and a mean of 967.55 ms, while tAPP has a standard deviation of 186.67 ms and a mean of 855.82 ms.

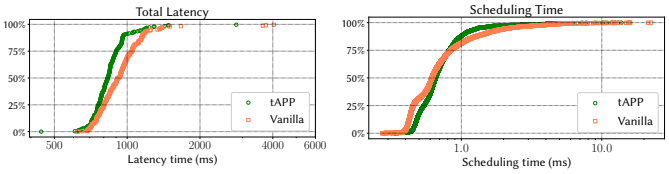

Figure 11: Test Case 5 latencies (left) and scheduling time (right).

Both cases 6 and 9 focus on startup latency, tested by invoking functions while inducing cold starts.

Case 6. In case 6, we invoke a Java function that uses a custom Java container runtime for OpenWhisk, which incurs long initialisation times. To enforce cold starts, one must either invoke the function after 10 minutes since the last invocation—OpenWhisk keeps a function's container alive for 10 minutes since the last call to reduce cold starts—or manually stop the function container on the worker. To avoid interfering with the platform's internal dynamics, we preferred the first option and, to keep the test times reasonable, we opted to perform 10 invocations for each test run. We present the latencies and scheduling times in Figure 12. The results are in line with the previous test cases, with negligible differences in scheduling times between the two platforms. The total latencies are stable between 2000 ms and 4000 ms with a mean of 2943.87 ms for tAPP OpenWhisk and a standard deviation of 476.39 ms, and a mean of 3494.07 ms and a standard deviation of 672.62 ms for vanilla OpenWhisk.

[13] Figure 10, as well as the other figures for the other test cases, reports the cumulative distribution functions of the latencies and the scheduling time, hence the y-axis represent probabilities (namely, the probability that latency or scheduling time is below a given value in the x-axis).

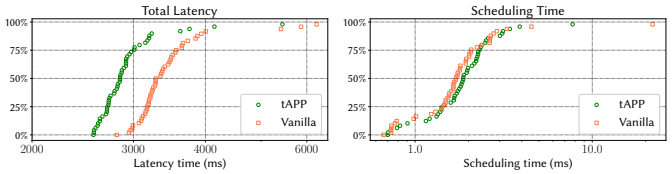

Figure 12: Test Case 6 latencies (left) and scheduling time (right).

Case 9. This test case covers a concurrent startup scenario to analyse how auto-scaling impacts function startup. We use the Java and C functions of the case, and we run 10 invocations of the functions in a single test run. For each function, we send 40 requests simultaneously, once with a maximum concurrency limit (i.e., the maximum number of concurrent invocations of the same function per container) set to 1 and once set to 40, effectively resulting in four subtests. We show the latencies and scheduling times in Figure 13 and Figure 14. In all subtests, the two platforms have similar performance. We report in Table 5 and Table 6 the aggregated mean, median, tail latency (i.e., 95 latency percentile), and the standard deviation of the tAPP-based and vanilla OpenWhisk deployments.

	C. lim.	Mean	Median	Tail	S. Dev.
C	1	8891.63	8692.05	10194.76	693.24
Java	1	9860.9	9601.95	11738.73,	930.73
C	40	10098.28	10187.49	11731.16	914.94
Java	40	8925.46	8766.97	10379.51	742.24

Table 5: Statistics of the tAPP-based OpenWhisk deployment for Test Case 9 (ms).

	C. lim.	Mean	Median	Tail	S. Dev.
C	1	8808.07	8524.01	10421.28	776.21
Java	1	10201.73	10245.48	11665.09	1025.93
C	40	10171.11	9961.97	12563.33	1288.63
Java	40	9433.21	9042.56	11360.72	1078.27

Table 6: Statistics of the Vanilla OpenWhisk deployment for Test Case 9 (ms).

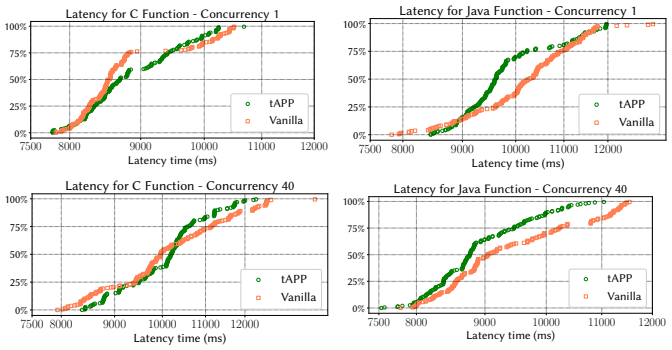

Figure 13: Test Case 9 latencies for the four subtests: C and Java functions with concurrency 1 (top-left and top-right), C and Java functions with concurrency 40 (bottom-left and bottom-right).

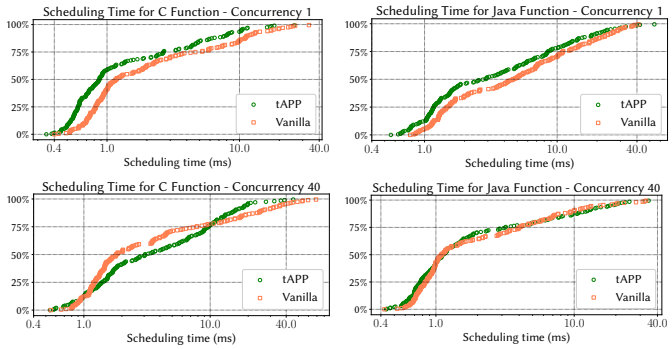

Figure 14: Test Case 9 scheduling time for the four subtests: C and Java functions with concurrency 1 (top-left and top-right), C and Java functions with concurrency 40 (bottom-left and bottom-right).

Case 10. This test case focuses on the effect of implicit state with a Java function that performs image resizing using a custom Java runtime. The function takes advantage of "warm" containers by re-using the implicit state of the Java runtime. We invoke the function 1000 times in a single test run, reporting in Figure 15 the latencies and scheduling times. Similarly to the previous test cases, the scheduling time differences between the two platforms are negligible. The request-reply latencies go from below 500 ms to more than 4000 ms for both platforms, with a mean of 610.67 ms and a standard deviation of 256.67 ms for tAPP OpenWhisk and a mean of 811.44 ms and a standard deviation of 431.69 ms for vanilla OpenWhisk.

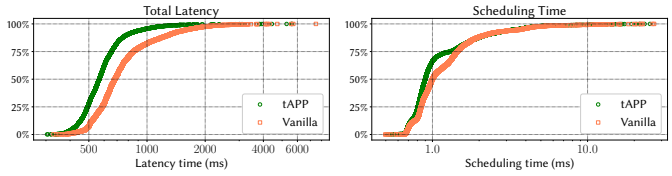

Figure 15: Test Case 10 latencies (left) and scheduling time (right).

Overall, the experiments show that our solution presents negligible performance overhead, since the scheduling time of our tAPP prototype is comparable to, if not better than, vanilla OpenWhisk.

These experiments complement earlier results [7] on the four topology-based worker distribution policies (cf. Section 5.3). From those results, the *default* policy achieves the strongest overall performance with 44% latency reduction, *isolated* reduces worker overloading but suffers from zone saturation (smaller zones can become saturated and unable to handle incoming requests), *min_memory* generally underperforms except in heavy data-locality scenarios (since it primarily confines function execution to the local zone, minimising latencies due to transferring large payloads), and *shared* excels in lightweight data-locality tests by effectively balancing local and remote worker usage.

8. Case Study and Overhead Analysis: Discussion, Limitations, and Threats to Validity

The results of the case study resolve the feasibility of C5 and function performance-improvement dimension of C6. Vanilla OpenWhisk's co-prime scheduling algorithm assigns workers based on a topology-oblivious hash, yielding correct placement only when, by chance, the hashed worker satisfies the topological constraint—a condition that depends on inaccessible internal configuration state and, therefore, is generally unreliable. By contrast, the tAPP-based deployment enforces correct placement deterministically—e.g., the `MQTT` tag with `topology_tolerance` set to `none` guarantees that **data-collection** functions execute exclusively on edge workers co-located with the MQTT broker. Regarding locality-bound function performance, function **feature-extraction** execution time, bound to accessing the database found in a specific (edge) zone, improves thanks to tAPP's policy, which prioritises the deployment of the function in the same zone of the database.

The ServerlessBench [4] benchmarks address the overhead dimension of C6, showing that tAPP's routing layer introduces negligible overhead relative to vanilla OpenWhisk across a representative range of workloads. The additional latency incurred by policy lookup and tag-based dispatching is consistently within experimental noise, confirming that topology awareness comes at no significant cost to general scheduling performance.

8.1. Threat to Validity

We conclude this section by discussing limitations and potential confounding factors that may affect the interpretation of our results, outlining to what extent the reported findings generalise beyond the evaluated setting.

Internal Validity. Our evaluation relies on an OpenWhisk-based prototype that extends the standard platform. As a result, we cannot specifically separate the measurement of the tAPP decision logic from the behaviour of the modified routing components and the underlying monitoring interfaces in our end-to-end latency and scheduling overhead measurements. To reduce the risk of attributing unrelated overhead to tAPP, we compare against unmodified OpenWhisk in the same environment and report timing distributions over repeated runs. Nevertheless, uncontrollable variables like available bandwidth and fluctuating performance of cloud cluster nodes may influence measured latencies and could introduce noise, particularly in tail latency. Additionally, the placement "misbehaviour" observed under vanilla OpenWhisk depends on platform internals (e.g., invoker selection logic, hashing decisions, and system state). While we repeat deployments to quantify the frequency of misplacement, we do not attempt to reverse-engineer the exact root cause. Therefore, the reported misplacement probability should be interpreted as empirical evidence of practical unpredictability in the tested configuration, rather than a platform-wide characterisation.

External Validity. Our implementation, based on OpenWhisk, relies on the latter's hooks for load balancing and routing. While the tAPP language is designed to be platform-agnostic, the enforcement of its semantics depends on the presence of amenable routing and placement control points. Indeed, platforms such as Knative or OpenFaaS provide similar hooks to realise a comparable semantics, but the exact implementation might differ. Therefore, our results demonstrate feasibility and low overhead of topology-aware policies on OpenWhisk-like architectures, but generalisation to other serverless stacks requires specific implementations and may yield different performance profiles. Furthermore, our case study emphasises geographically or topologically constrained workflows (edge/cloud separation). While such scenarios are common in IoT and hybrid deployments, workloads dominated by compute-bound functions without locality constraints may enjoy limited benefit from topology-aware scheduling. In these cases, tAPP primarily provides expressive control rather than performance gains.

Construct Validity. Some policy constructs intentionally abstract over implementation details. For example, `capacity_used n%` denotes a threshold on a notion of worker "capacity" that depends on available metrics and platform instrumentation. In our prototype this is instantiated using memory-related load information, but different environments might incorporate different resources (e.g., memory, I/O pressure) or smoothing windows. Consequently, the semantics are best understood as a declarative constraint instantiated with different operational definitions; comparisons across platforms must account for these metric differences. We conduct our experiments in a controlled environment, and they do not fully capture the dynamics of large-scale multi-tenant deployments where background load, competing policies, and resource contention may affect placement decisions.

9. Related Work

We position our work starting from the context of the APP language and its evolutions. Then, we broaden our scope to consider general work on serverless scheduling.

As remarked in Section 1, this work builds upon preliminary conference publications [5, 6, 7], which introduced the language APP, its extension tAPP, and demonstrated an extension of the Apache OpenWhisk serverless platform supporting tAPP-based scripts. Recent work on APP [24] has focused on affinity-aware scheduling, proposed to co-locate or separate functions depending on (anti-)affinity constraints. While these constraints are fundamental for optimising execution locality and avoiding contention, their integration in the case of topological deployments across multiple controllers (e.g., spanning cloud and edge environments) is challenging. Indeed, these constraints potentially require controllers to disseminate and synchronise information on the allocation of functions to maintain a coherent global view, thereby introducing scheduling overheads and delays.

On the theoretical side, the semantics of APP has been formally studied, in [25]. There is no formal treatment of the semantics of tAPP. Future work can complement this line of research by extending the expressiveness of APP with topology-specific constraints and possibly study scheduling properties in multi-controller settings.

The broader research program around APP has also initiated investigations into complementary aspects. For instance, De Palma et al. [26] employed static analysis to design a cost-aware APP extension that can guide scheduling decisions by estimating execution latency on workers with respect to external service calls made by functions. Similarly, formal verification techniques have been explored to provide correctness guarantees for APP function scheduling properties [27]. These initial investigations highlight the potential of combining formal methods with practical mechanisms to achieve predictable, verifiable, and efficient scheduling in serverless platforms.

Widening our scope, recent surveys [28, 29, 30] present comprehensive overviews and reflect on growing interests in serverless scheduling, especially across cloud, edge, and fog environments. Many works on serverless focus on minimising the latency of function invocations. Several of them tackle the problem by optimising function scheduling [31, 32]. Other approaches focus on minimising cold start latency, as discussed in the survey [33].

One work close to ours is by [34], 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, [35] introduce a scheduling policy that governs the order of invocation processing, depending on the availability of the resources they use. [36, 37] propose an energy-aware scheduler, where function scheduling decisions are taken by considering the energy position of the active nodes in the network. [38] present a package-aware technique that favours re-using the same workers for the same functions to cache dependencies. [39] show a scheduling policy oriented by resource usage of co-located functions on workers. [40] and [41] 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, [42] 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. [43] use state by supporting both global and local state access, aiming at performance improvements for data-intensive applications. Similarly, [44] associate each function invocation with a shared log among serverless functions. Additionally, approaches like Pheromone, by [45], combine local schedulers, which locally execute function workflows, and global coordinators, which offload the functions when local executors are busy. The local schedulers, combined with worker-specific shared-memory object stores, allow functions to rapidly exchange data without going through external storage.

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.

Recent developments in FaaS also involve the definition and management of function compositions or workflows, exemplified by AWS Step Functions [46] and Azure Durable Functions [47] or the open source solutions of Apache OpenWhisk Composer [48], FaaSFlow [49], Triggerflow [50], and Beldi [51]. The fundamental concept beyond these advancements is to allow users to specify workflows by combining functions with branching logic, parallel execution, and error-handling capabilities. Then, the orchestrator or controller of the platform uses the defined workflow to oversee function executions, managing aspects such as retries, timeouts, and error resolution. The problem of workflow specification/control is orthogonal to the problem of function scheduling that tAPP addresses. However, we see an interesting branch of future work in coordinating tools for the specification/management of serverless workflow with tAPP, e.g., one can extend tAPP with constructs that specify the allocation of functions on workers where functions of the same workflow run (e.g., to avoid traversing the network stack for communication or reuse the pool of connections to a database).

Inspiring approaches in this direction are by [52], who present a serverless platform where developers can constrain the information flow among functions to avoid attacks due to container reuse and data exfiltration, and by [53], who propose an extension of the TOSCA standard to control the flow of data inside Cloud applications with serverless components.

Looking at other work that uses localities to improve FaaS performance, Lambdata [54] is an OpenWhisk extension that improves its performance considering locality and cold starts. Lambdata builds on top of both the OpenWhisk's Controller and Invoker components to allow users to annotate functions with explicit *data intents*, specifying which buckets they intend to use for reading and writing. Lambdata's approach is complementary to ours since tAPP allows for more fine-grained control over function-Invoker assignment, while Lambdata infers such assignments from annotations. Steinbach et al. explore a different line of work with TppFaaS [55], where they model function compositions using Temporal Point Processes. We consider the work on TppFaaS orthogonal to our approach since they require no explicit locality requirements or configurations, and the user mostly relies on the accuracy of the underlying model. While the authors only tested their proposal in terms of accuracy over generated trace datasets—i.e., they did not apply it to locality issues—we see their approach interesting for applications for predictive scheduling and scaling.

Beyond single-cloud deployments—i.e., which require coordination between different providers—we mention xAFCL [56] and SkyPilot [57] (although the latter is not directly related to FaaS). xAFCL [56] handles invocations over several FaaS providers, to optimise the execution of function workflows by estimating the duration of each function and forwarding its invocation to the appropriate provider. SkyPilot [57] follows a similar approach, but it acts as middleware between the user and several cloud providers, to dynamically select the appropriate target for requests according to cost, latency, and security requirements. Both xAFCL and SkyPilot work at a higher level of abstraction compared to our tAPP, intervening between the user and the target platform, and one could follow their approach to coordinate work between tAPP-based OpenWhisk and commercial solutions.

Finally, an interesting domain of application is that of Sky Com-

puting [58], where brokers handle the placement and oversee the execution of cloud jobs over multiple cloud providers. In the case of FaaS, we mention funcX [59], which is a federated serverless solution that allows users to register their infrastructure as part of the platform's deployment and run their functions on any node they are authorised to access. While OpenWhisk is not suitable for such an approach (since it has no notion of federation), one can apply tAPP to this domain by employing topology-aware scheduling policies when users wish to run a function on a certain endpoint or group of endpoints. We also mention a recent work by Nardelli and Russo [60], which explores the concept of a decentralised serverless platform, where each node acts as entrypoint, and can either compute functions locally or offload them to other nodes. The scheduling in this case is completely automatic, and relies on data access probability estimates to predict the optimal node for function invocation. While tAPP is based on a more centralised architecture, it can be easily extended to target zones directly, without relying on a separate controller, to integrate user knowledge with the existing estimates.

10. Conclusion

We introduced tAPP in response to the following research question discussed in Section 1:

RQ. How can FaaS developers customise the scheduling policies of cloud-edge serverless platforms to improve function performance in locality-bound scenarios?

In Section 3, we broke down the above research question into six challenges that we have then addressed in Sections 4.3, 5.6 and 8. Briefly, tAPP is a declarative language that provides DevOps with fine control over 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 (C1–C4). To validate our approach, we presented a prototype tAPP-based serverless platform (C5), developed on top of OpenWhisk, and we used it to show that tAPP allows for an low-overhead deployment of cloud-edge serverless systems with typical topology-aware scheduling constraints that cannot be guaranteed by standard vanilla OpenWhisk deployments (C6).

Although tAPP addresses the core challenges of topology-aware serverless scheduling, future work can extend its design and that of its current prototype.

One limitation concerns policy tags, which are created implicitly when the first function bearing a tag is uploaded, and deleted by manually editing the tAPP script. The current prototype does not support reassigning a tag to an already-deployed function—the workaround is redeployment under a new tag. Supporting in-place tag mutation is left for future work, as it requires a distributed cache-consistency protocol or a versioning scheme for tag bindings. In practice, this limitation is partially mitigated at the language level by worker sets, whose membership is evaluated at invocation time and therefore evolves without any script change.

tAPP's `strategy` parameter supports non-deterministic controller selection via the `random` option, which distributes invocations uniformly across controller blocks—while the default, `best-first` option enforces sequential ordering. Thus, the interplay between tag-level and block-level `strategy` options encodes a form of branching. A natural extension can support content-based branching, where the controller block is chosen based on request attributes such as function parameter's size or caller identity.

tAPP enforces a one-tag-per-function constraint, keeping scheduling lookup constant-time at the cost of limited composability: functions with multiple orthogonal constraints must fold them into a single tag. Lifting this restriction would require merging potentially conflicting multi-tag policies at invocation time, a problem analogous to access-control policy composition.

In the prototype, the set of controllers is fixed at deployment time—a constraint inherited from OpenWhisk's replica management model rather than from tAPP itself. At the language level there is no restriction on controller cardinality, and new controller blocks can be activated via live-reload, provided the corresponding process is already registered with Kafka and CouchDB. Decoupling controller lifecycle from platform redeployment—for instance via Kubernetes operators—is an important extension for cloud-edge deployments where edge controllers may be intermittently available.

Further future work includes applying tAPP on different platforms, e.g., OpenLambda, OpenFaaS, and Fission. Technically, to adapt tAPP to other solutions, like Kubernetes-based serverless platforms, we would need to leverage the concepts of node affinity and zone labels. Since such platforms generally approach function invocation using the deployment of a pod (and, in the case of KNative, a service), we would need to specify that pods coming from functions with specific tags have a soft affinity constraint towards the targeted zone. When the platform does not have a specific "Controller" component but relies on the Kubernetes scheduler for function placement tAPP could simply target the required zone, instead of the local Controller. Note that we already make use of Kubernetes zone labels for the current tAPP implementation, using them as a way to inform the Controllers about their geographical location, and which Invokers are in it.

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. Regarding tests, we remark on the general need for more platform-agnostic and realistic suites, to obtain fairer and thorough comparisons.

Finally, we would like to support DevOps in the optimisation of serverless applications by studying and experimenting with heuristics and AI-based mechanisms that profile applications and suggest optimal tAPP policies. Similarly, extensions of tAPP could benefit from interactions with frameworks able to specify function compositions, e.g., Yussupov et al. [61] recently introduced a method for modelling and deploying serverless function orchestrations which one could use to extract execution dependencies among functions and inform the synthesis of tAPP policies that optimise the overall execution of compositions.

References

[1] E. Jonas, et al., Cloud Programming Simplified: A Berkeley View on Serverless Computing, Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley (2019).

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

[3] G. D. Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Funless: Functions-as-a-service for private edge cloud systems, in: IEEE International Conference on Web Services, ICWS 2024, Shenzhen, China, July 7-13, 2024, IEEE, 2024, pp. 961–967. doi:10.1109/ICWS62655.2024.00114.
URL https://doi.org/10.1109/ICWS62655.2024.00114

[4] T. Yu, et al., Characterizing serverless platforms with serverlessbench, in: Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 30–44. doi:10.1145/3419111.3421280.

[5] G. De Palma, et al., Allocation Priority Policies for Serverless Function-Execution Scheduling Optimisation, in: Proc. of ICSOC, Vol. 12571 of LNCS, Springer, 2020, pp. 416–430. doi:10.1007/978-3-030-65310-1_29.

[6] G. De Palma, et al., 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.

[7] G. De Palma, et al., An OpenWhisk Extension for Topology-Aware Allocation Priority Policies, in: COORDINATION, Vol. 14676 of LNCS, Springer, 2024, pp. 201–218. doi:10.1007/978-3-031-62697-5_11.

[8] G. D. Palma, et al., tAPP OpenWhisk: A serverless platform for topology-aware allocation priority policies, Sci. Comput. Program. 247 (2026) 103349. doi:10.1016/J.SCICO.2025.103349.
URL https://doi.org/10.1016/j.scico.2025.103349

[9] Amazon, AWS Lambda, https://aws.amazon.com/lambda/ (2025).

[10] Google, Google Cloud Functions, https://cloud.google.com/functions/ (2025).

[11] Microsoft, Microsoft Azure Functions, https://azure.microsoft.com/ (2025).

[12] D. Bernstein, Containers and cloud: From lxc to docker to kubernetes, IEEE Cloud Computing 1 (3) (2014) 81–84.

[13] M. Armbrust, et al., Above the clouds: A berkeley view of cloud computing, University of California, Berkeley, Rep. UCB/EECS 28 (13) (2009) 2009.

[14] L. Wang, et al., Peeking behind the curtains of serverless platforms, in: 2018 USENIX Annual Technical Conference (USENIX/ATC 18), 2018, pp. 133–146.

[15] Q. Xie, et al., Pandas: robust locality-aware scheduling with stochastic delay optimality, IEEE/ACM Trans. on Networking 25 (2) (2016) 662–675.

[16] W. Wang, et al., MapTask Scheduling in MapReduce With Data Locality: Throughput and Heavy-Traffic Optimality, IEEE/ACM Trans. Netw. 24 (2016) 190–203. doi:10.1109/TNET.2014.2362745.

[17] H. B. Hassan, et al., Survey on serverless computing, Journal of Cloud Computing 10 (1) (2021) 1–29.

[18] J. Kreps, et al., Kafka: A distributed messaging system for log processing, in: Proc. of NetDB, Vol. 11, 2011, pp. 1–7.

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

[20] O. Ben-Kiki, et al., Yaml ain't markup language (yaml™) version 1.2, https://yaml.org/spec/1.2.2/ (2021).

[21] K. Hong, et al., A method of real-time fault diagnosis for power transformers based on vibration analysis, Measurement Science and Technology 26 (11) (2015) 115011. doi:10.1088/0957-0233/26/11/115011.

[22] K. Hong, et al., Transformer Winding Fault Diagnosis Using Vibration Image and Deep Learning, IEEE Transactions on Power Delivery 36 (2) (2021) 676–685. doi:10.1109/TPWRD.2020.2988820.

[23] MQTT, mqtt.org, MQ Telemetry Transport, http://mqtt.org/ (2025).

[24] G. De Palma, et al., Affinity-aware Serverless Function Scheduling, in: ICSA, IEEE, 2025, pp. 49–59. doi:10.1109/ICSA65012.2025.00015.

[25] G. De Palma, et al., Function-as-a-Service Allocation Policies Made Formal, in: REoCAS Colloquium, ISoLA, Vol. 15219 of LNCS, Springer, 2024, pp. 306–321. doi:10.1007/978-3-031-73709-1_19.

[26] G. De Palma, et al., Leveraging static analysis for cost-aware serverless scheduling policies, Int. J. Softw. Tools Technol. Transf. 26 (6) (2024) 781–796. doi:10.1007/S10009-024-00776-9.

[27] G. De Palma, , et al., Formally Verifying Function Scheduling Properties in Serverless Applications, IT Prof. 25 (6) (2023) 94–99. doi:10.1109/MITP.2023.3333071.

[28] M. Ghorbian, et al., Function Placement Approaches in Serverless Computing: A Survey, J. Syst. Archit. 157 (2024) 103291. doi:10.1016/J.SYSARC.2024.103291.

[29] M. Ghorbian, et al., A survey on the scheduling mechanisms in serverless computing: a taxonomy, challenges, and trends, Clust. Comput. 27 (5) (2024) 5571–5610. doi:10.1007/S10586-023-04264-8.

[30] M. Ghorbian, M. Ghobaei-Arani, Function offloading approaches in serverless computing: A survey, Comput. Electr. Eng. 120 (2024) 109832. doi:10.1016/J.COMPELECENG.2024.109832.
URL https://doi.org/10.1016/j.compeleceng.2024.109832

[31] A. Kuntsevich, et al., A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform, in: Proc. of Middleware (Posters), 2018, pp. 3–4.

[32] M. Shahrad, et al., Architectural implications of function-as-a-service computing, in: Proc. of MICRO, 2019, pp. 1063–1075.

[33] M. Ghorbian, M. Ghobaei-Arani, A survey on the cold start latency approaches in serverless computing: an optimization-based perspective, Computing 106 (11) (2024) 3755–3809. doi:10.1007/S00607-024-01335-5.

[34] J. Sampé, et al., Data-Driven Serverless Functions for Object Storage, in: Proc. of Middleware, ACM, 2017, pp. 121–133. doi:10.1145/3135974.3135980.

[35] A. Banaei, M. Sharifi, ETAS: predictive scheduling of functions on worker nodes of Apache OpenWhisk platform, The Journal of Supercomputingdoi:10.1007/s11227-021-04057-z.

[36] M. Ghorbian, M. Ghobaei-Arani, L. Esmaeili, An energy-conscious scheduling framework for serverless edge computing in IoT, J. Cloud Comput. 14 (1) (2025) 52. doi:10.1186/S13677-025-00780-7.

[37] M. Ghorbian, M. Ghobaei-Arani, L. Esmaeili, Autonomous scheduling mechanism based on energy awareness for improving resource allocation in serverless IoT edge, Scientific Reports 15 (24173). doi:10.1038/s41598-025-04214-x.

[38] C. L. Abad, et al., Package-Aware Scheduling of FaaS Functions, in: Proc. of ACM/SPEC ICPE, ACM, 2018, pp. 101–106. doi:10.1145/3185768.3186294.

[39] A. Suresh, A. Gandhi, FnSched: An Efficient Scheduler for Serverless Functions, in: Proc. of WOSC@Middleware, ACM, 2019, pp. 19–24. doi:10.1145/3366623.3368136.

[40] M. Stein, The serverless scheduling problem and NOAH, arXiv preprint arXiv:1809.06100.

[41] I. E. Akkus, et al., SAND: Towards High-Performance Serverless Computing, in: Proc. of USENIX/ATC, 2018, pp. 923–935.

[42] S. Kotni, et al., Faastlane: Accelerating Function-as-a-Service Workflows, in: Proc. of USENIX ATC, USENIX Association, 2021, pp. 805–820.

[43] S. Shillaker, P. Pietzuch, Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing, in: Proc. of USENIX ATC, USENIX Association, 2020, pp. 419–433.

[44] Z. Jia, E. Witchel, Boki: Stateful Serverless Computing with Shared Logs, in: Proc. of ACM SIGOPS SOSP, ACM, New York, NY, USA, 2021, pp. 691–707. doi:10.1145/3477132.3483541.

[45] M. Yu, et al., Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing, in: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), USENIX Association, Boston, MA, 2023, pp. 1489–1504.

[46] Amazon, AWS Step Functions, https://aws.amazon.com/step-functions/ (2025).

[47] Microsoft, Azure Durable Functions, https://docs.microsoft.com/en-us/azure/azure-functions/durable/ (2025).

[48] Apache Software Foundation, Apache OpenWhisk Composer, https://github.com/apache/openwhisk-composer (2024).

[49] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, M. Guo, Faasflow: enable efficient workflow execution for function-as-a-service, in: ASPLOS, ACM, 2022, pp. 782–796.

[50] A. Arjona, P. G. López, J. Sampé, A. Slominski, L. Villard, Triggerflow: Trigger-based orchestration of serverless workflows, Future Gener. Comput. Syst. 124 (2021) 215–229.

[51] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, V. Liu, Fault-tolerant and transactional stateful serverless workflows, in: USENIX OSDI, USENIX Association, 2020, pp. 1187–1204.

[52] P. Datta, et al., Valve: Securing function workflows on serverless computing platforms, in: Proceedings of The Web Conference 2020, 2020, pp. 939–950.

[53] C. K. Dehury, et al., TOSCAdata: Modeling data pipeline applications in TOSCA, J. Syst. Softw. 186 (2022) 111164. doi:10.1016/J.JSS.2021.111164.

[54] Y. Tang, J. Yang, Lambdata: Optimizing Serverless Computing by Making Data Intents Explicit, in: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 294–303. doi:10.1109/CLOUD49709.2020.00049.

[55] M. Steinbach, et al., TppFaaS: Modeling Serverless Functions Invocations via Temporal Point Processes, IEEE Access 10 (2022) 9059–9084. doi:10.1109/ACCESS.2022.3144078.

[56] S. Ristov, et al., xAFCL: Run Scalable Function Choreographies Across Multiple FaaS Systems, IEEE Transactions on Services Computing 16 (1) (2023) 711–723. doi:10.1109/TSC.2021.3128137.

[57] Z. Yang, et al., SkyPilot: An Intercloud Broker for Sky Computing, in: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), USENIX Association, Boston, MA, 2023, pp. 437–455.

[58] I. Stoica, S. Shenker, From cloud computing to sky computing, in: S. Angel, B. Kasikci, E. Kohler (Eds.), HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021, ACM, 2021, pp. 26–32. doi:10.1145/3458336.3465301.

[59] R. Chard, et al., funcX: A Federated Function Serving Fabric for Science, in: Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 65–76. doi:10.1145/3369583.3392683.

[60] M. Nardelli, G. R. Russo, Function Offloading and Data Migration for Stateful Serverless Edge Computing, in: S. Balsamo, W. J. Knottenbelt, C. L. Abad, W. Shang (Eds.), Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, ICPE 2024, London, United Kingdom, May 7-11, 2024, ACM, 2024, pp. 247–257. doi:10.1145/3629526.3649293.

[61] V. Yussupov, et al., Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA, Software: Practice and Experiencedoi:10.1002/spe.3073.