Science of Computer Programming 247 (2026) 103349

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Original Software Publication

tAPP OpenWhisk: A serverless platform for topology-aware allocation priority policies

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

[a] Università di Bologna, Italy
[b] Southern Denmark University, Odense, Denmark
[c] INRIA, France

ARTICLE INFO

Keywords:
Serverless
Function-as-a-Service
Cloud optimization
Topology-awareness

ABSTRACT

The Function-as-a-Service (FaaS) paradigm offers a serverless approach that abstracts the management of underlying infrastructure, enabling developers to focus on application logic. However, leveraging infrastructure-aware features can further optimize serverless performance. We present a software prototype that enhances Apache OpenWhisk serverless platform with a novel architecture incorporating tAPP (topology-aware Allocation Priority Policies), a declarative language designed for specifying topology-aware scheduling policies. Through a case study involving distributed data access across multiple cloud regions, we show that tAPP can significantly reduce latency and minimizes performance variability compared to the standard OpenWhisk implementation.

1. Introduction

Function-as-a-Service (FaaS) is a serverless model in which users program architectures of stateless functions, leaving system operations entirely to the platform [1]. Although FaaS abstracts away infrastructural details, providing information on topological traits can enhance the performance of serverless applications. For example, function execution performance often depends on the specific computing resource, or *worker*, handling the task. When choosing a worker to execute a function, overlooking *data locality* [2] — the latency of accessing data at a repository — and *session locality* [2] — the latency of authenticating and initiating new sessions with stateful services— can cause delays and overhead, negatively affecting function runtime. The tAPP language [3] enables users to declaratively specify infrastructural constraints related to the topology of the infrastructure, to guide function scheduling.

We present the software artefact we developed to support tAPP-based scheduling policies obtained by extending Apache OpenWhisk. This article presents the software artefact of a previous contribution [4], providing more details on the deployment of the platform and a reproducible, self-contained case study. We illustrate its use by presenting a simple case study on the execution of serverless functions showing that topology-aware allocation policies can significantly improve the performance of locality-bound functions. The artefact is available at https://zenodo.org/records/15481898.

* Corresponding author.
E-mail address: giuseppe.depalma2@unibo.it (G. De Palma).

https://doi.org/10.1016/j.scico.2025.103349
Received 16 December 2024; Received in revised form 21 May 2025; Accepted 29 May 2025

Available online 9 June 2025

```
couchdb_query:
 - workers:
    - wrk: DB_worker1
    - wrk: DB_worker2
   strategy: random
   invalidate: capacity_used 50%
   controller: EUzone
   topology_tolerance: same
 followup: fail
```

Fig. 1. Example of a simple tAPP script.

2. The tAPP language

In serverless platforms, developers define functions in languages like JavaScript and Python and specify events that trigger the execution of these functions. For example, a data storage request may trigger the execution of a function that works on the stored data. The serverless platform manages all stages of function scheduling, deployment, and monitoring. Often, these platforms use containers [5] or VMs [6] to create the isolated execution environments where functions can run without interfering with each other, while sharing the same workers. Since the creation of these environments takes time (impacting the run time of functions), serverless platforms usually optimise performance by having workers keep a pool of "warm" environments where they ran their functions for later re-use. Besides the mentioned session and data localities, *code locality* [2] entails the possibility of identifying the workers with a warm environment for the function under scheduling, so they can run the function without enduring latencies due to environment initialisation.

tAPP The tAPP language specifies scheduling policies of functions on workers for serverless functions. Essentially, a policy couples a tag to a list of policy blocks, each specifying on which workers to schedule the function and how to select among them.

We report in Fig. 1 an example policy where the tag `couchdb_query` targets functions accessing a database. In the policy, we select specific workers (`DB_worker1` or `DB_worker2`) `random`ly and specify their validity to run the functions based on the occupancy of the other functions running on them — specifically, we indicate that the worker's available resources shall be greater than 50% of their total capacity.

Since topology-aware scheduling regards the management of the allocation of functions across different zones, we assume to have a controller in each of the zones that make up the deployment of the platform. In the example, we assume to have a zone tagged `EUzone`, and we indicate that the function should be scheduled in that zone with the `controller` clause. Moreover, we can specify a `topology_tolerance` parameter, which indicates if the functions can be scheduled in other zones than the indicated one. Specifically, `same` constrains the function to run on workers in the same zone of the indicated controller.

The `followup` clause specifies what to do if all workers are invalid; in the example, `fail` indicates to stop the scheduling in case we found no available worker.

The interested reader can find a complete presentation of tAPP in the paper that introduced the language [3].

3. tAPP-based OpenWhisk architecture

Our tAPP prototype builds on Apache OpenWhisk, a widely-used, open-source FaaS platform. Specifically, we target an OpenWhisk Kubernetes-based deployment. For compactness, we show in Fig. 2 the architecture of OpenWhisk and highlight the components that we modify/introduce to support tAPP-based scheduling.

Commenting on Fig. 2, from left to right, we find *Nginx*, which acts as the system's entry point and load balancer, directing requests to *Controller*s. Controllers assign function execution to specific computation nodes called *Worker*s. Apache *Kafka* [7] is used for request routing and queueing, and *CouchDB* [8] is used for authorisation, function storage, and response management.

The key components that we modify/add in our tAPP extension include:

OpenWhisk controller We modify the Controller's load balancer (written in Scala) to interpret tAPP policies through a new `ConfigurableLoadBalancer` class, adding policy parsing and scheduling capabilities to support dynamic worker selection based on tAPP tags.

Watcher and NFS server services We introduce a Watcher service that interacts with the Kubernetes API to track cluster topology, mapping Kubernetes nodes to tAPP labels and storing this information in an NFS (Network File System) server (also introduced by us). Nginx and controllers access this data to resolve tags (irrespective of physical deployment details), allowing for dynamic topology changes.

Nginx gateway We modify Nginx using *njs* (Nginx JavaScript)[1] to parse function requests and forward tagged requests based on tAPP policy. The Nginx plugin caches policy data for efficient handling of inbound traffic.

[1] https://github.com/nginx/njs.

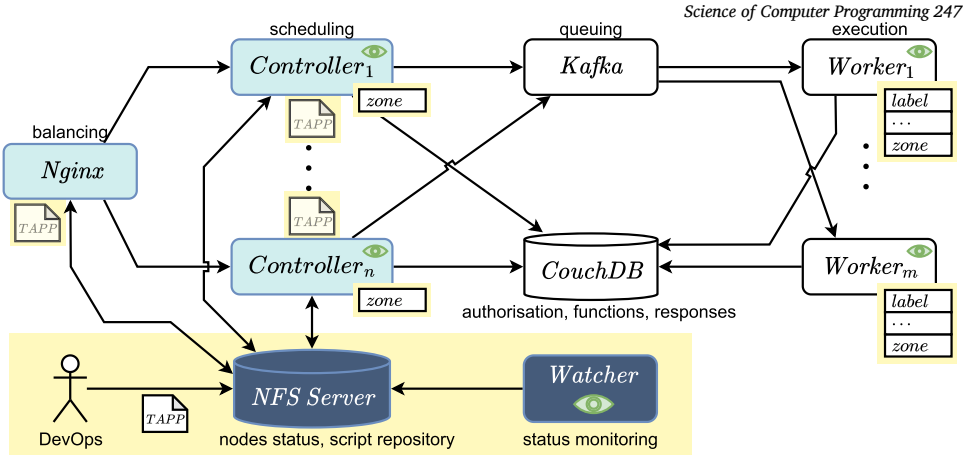

Fig. 2. Architectural view of our OpenWhisk extension. The existing OpenWhisk components we modified are in light blue while the new ones are in yellow. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

```
data:
 - controller: EUzone
   workers:
    - wrk: EU-w1
   topology_tolerance: all
 followup: default
```

Fig. 3. The tAPP script used in the tests.

4. Case study

We present an illustrative example of deployment and execution of functions on our prototype, in comparison with vanilla OpenWhisk. In particular, we focus on functions that present data-locality properties, showing how tAPP improves function runtimes.

Experiments execution environment To run the experiments of our case study, we use a Kubernetes cluster deployed on five nodes across two Google Cloud Platform regions (Central US and West Europe). Of the five nodes, three are in the West Europe region and two in the Central US region. Two nodes in each region host an OpenWhisk Controller and an Invoker, respectively. The remaining one runs the Kubernetes control-plane. All the nodes are of type `e2-medium` with 2 vCPUs and 4 GB of memory, running Ubuntu 20.04. We automate the creation of the VMs and the installation of Kubernetes and OpenWhisk using Terraform and Ansible scripts.[2]

The Invoker in the European region hosts a web server, exposing a simple REST API with one endpoint to retrieve a large JSON document of 26.14 MB.

The behaviour of the function in the case study regards retrieving the JSON document from the web server and returning the document's byte size. We execute the function 100 times for each platform (vanilla and tAPP-based), measuring the latency between the sending of the execution request and the response.

tAPP configuration Regarding the testing of our prototype, in Fig. 3, we show the script we define to optimise function execution. The script specifies that the functions should be scheduled in the European region — i.e., by the `controller` located in the `EUzone`. We leave the strategy and invalidate options unspecified, which makes tAPP use the platform's default ones. In this way, we focus only on the impact of the topology-aware scheduling by using the same strategy and invalidation logic between vanilla OpenWhisk and our prototype. We set the topology tolerance to `all`, which allows the function to be scheduled on other regions, if the scheduling within the `EUzone` fails — thanks to the `followup` parameter set to `default`.

Results We show the latencies in Fig. 4, corresponding to the 100 function executions on vanilla OpenWhisk and our tAPP prototype, visualised as a cumulative distribution function showing the proportions of the requests completed in a given time. From the plots, we can observe a significant reduction in latency when using tAPP-based OpenWhisk with the tAPP script above. Interestingly, the plot illustrates also that, by using tAPP, we can stabilise an otherwise highly variable performance scenario due to OpenWhisk's default load-balancing behaviour that routes requests in a round-robin fashion. We observe this behaviour given the gradual increment of the tAPP-based requests, which continue until around 90%, while the cumulative distribution of the vanilla OpenWhisk plot becomes steeper at around the 50% mark. We explain this phenomena given that we have one worker in each of the two zones, one close to

[2] The scripts are available at https://github.com/giusdp/scp-artifact.

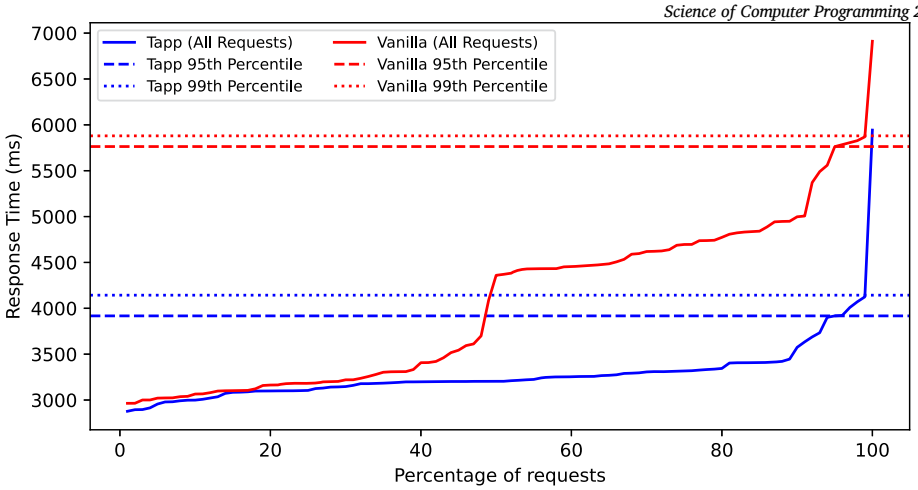

Fig. 4. Cumulative distribution funciton of the request-response latencies of 100 invocations of the use case function under vanilla and tAPP-based OpenWhisk.

Table 1
Latency measurements for vanilla OpenWhisk and our tAPP-based extension.

Configuration	Average (ms)	Median (ms)	Std Dev (ms)
tAPP-based OW	3268.13	3204.22	251.87
Vanilla OW	4054.36	4365.03	902.14

the service where we fetch the data and one further from it, shorter latencies correspond to scheduling instances that selected the worker in the European region, while longer ones correspond to instances where the platform selected the worker in the US region. Our observations are confirmed by the 95th, 99th and 100th percentiles of the cumulative distributions of the two cases, where the three thresholds of the tAPP-based one are more than 1 second lower than the corresponding values of the vanilla OpenWhisk case.

We summarise the results of the experiments in Table 1, where we report the measured average, median, standard deviation, and maximum latency of vanilla OpenWhisk and our tAPP prototype.

From the results in Table 1, we can see how our tAPP-based variant significantly reduces the average and median latencies compared to vanilla OpenWhisk.

4.1. Threats to validity and limitations

While our case study demonstrates significant latency reductions when using the tAPP-enhanced version of OpenWhisk, several threats to validity may affect the generalisability of these results. First, the experiments were conducted in a controlled environment with a specific cloud provider (Google Cloud Platform) and a particular infrastructure setup (two regions with nodes of type e2-medium). Different cloud providers, node types, or network conditions could impact the effectiveness of topology-aware scheduling differently. Second, the case study focused on a specific type of workload characterised by data locality requirements; workloads with different computational or networking patterns might not benefit to the same extent from tAPP policies. Third, the current evaluation is based on a single function and a limited set of invocation scenarios; more diverse workloads and larger-scale deployments could introduce additional variability. Finally, potential external factors such as transient network congestion and region-specific performance variability may influence the observed latencies. Future studies encompassing broader settings, varied workloads, and stress-testing under dynamic conditions would strengthen the generalisability of our findings.

Although the tAPP-enhanced OpenWhisk shows clear improvements in average and median latencies, several limitations remain. As seen in the experimental results, sporadic spikes in response times persist even under topology-aware scheduling. These can largely be attributed to cold start phenomena [9], where a function execution requires the initialisation of a new container, introducing unavoidable latency overhead. While tAPP can guide scheduling based on geographical locality (data locality), improving performance using other locality principles, like code locality, requires further extensions of the APP language, e.g., supporting affinity-based approaches — these improve performance exploiting the co-occurrence of functions on the same workers can significantly reduce cold starts and better exploit locality at a finer granularity [10]. Integrating code locality awareness and affinity-driven scheduling into tAPP would be a promising direction for further reducing latency variability and improving cold start resilience. Another mitigation strategy is using alternative serverless implementation technologies than standard virtual machines and containers, e.g., through lightweight runtime like WebAssembly [11,12].

5. Related work and conclusion

Related work There are many proposals that optimise serverless function latency via scheduling [13,14], investigating among other resource-aware scheduling [15,16], package-aware caching [17], prioritising by data locality [18], and function isolation and state sharing [19,20]. The main difference between these approaches and tAPP is that the former have an implicit notion of topology related to some proxy property (resources, caching, partitioning), whereas tAPP introduces topology-specific constraints that allow users to model explicitly topological properties of the scheduling logic.

Conclusion We presented the implementation of a tAPP-based serverless platform obtained by extending Apache OpehWhisk. tAPP enables topology-aware scheduling and we present an experiment, consisting of a locality-sensitive use case, in which tAPP boosts performances w.r.t. vanilla OpenWhisk. As future work, we aim to extend tAPP compatibility to platforms like OpenLambda and OpenFAAS and explore cloud-edge use cases.

6. Metadata

See Table 2.

Table 2
Code metadata.

Nr.	Code metadata description	
C1	Current code version	commit 0bd27a5
C2	Permanent link to code/repository used for this code version	https://github.com/mattrent/openwhisk
C3	Permanent link to Reproducible Capsule	https://zenodo.org/records/15481898
C4	Legal Code License	Apache 2.0
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Scala, Kubernetes, GCP Compute Engine
C7	Compilation requirements, operating environments and dependencies	Linux, Docker, Scala 2.12, OpenJDK 8
C8	If available, link to developer documentation/manual	https://github.com/mattrent/openwhisk/blob/master/README.md
C9	Support email for questions	

CRediT authorship contribution statement

Giuseppe De Palma: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Saverio Giallorenzo:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Jacopo Mauro:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Matteo Trentin:** Writing – review & editing, Writing – original draft, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Gianluigi Zavattaro:** Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Methodology, Investigation, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R.A. Popa, I. Stoica, D.A. Patterson, 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, S. Sturdevant, T. Harter, V. Venkataramani, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau, Serverless computation with openlambda, in: Proc. of USENIX HotCloud, 2016.
[3] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, 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.
[4] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, An openwhisk extension for topology-aware allocation priority policies, in: I. Castellani, F. Tiezzi (Eds.), Coordination Models and Languages - 26th IFIP WG 6.1 International Conference, COORDINATION 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, Proceedings, DisCoTec 2024, Groningen, the Netherlands, June 17-21, 2024, in: Lecture Notes in Computer Science, vol. 14676, Springer, 2024, pp. 201–218.
[5] D. Bernstein, Containers and cloud: from lxc to docker to kubernetes, IEEE Cloud Comput. 1 (3) (2014) 81–84.
[6] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, Above the clouds: a Berkeley view of cloud computing, University of California, Berkeley, 2009, Rep. UCB/EECS 28 (13) (2009).

[7] J. Kreps, N. Narkhede, J. Rao, et al., Kafka: a distributed messaging system for log processing, in: Proc. of NetDB, vol. 11, 2011, pp. 1–7.
[8] J.C. Anderson, J. Lehnardt, N. Slater, CouchDB: the Definitive Guide: Time to Relax, O'Reilly Media, Inc., 2010.
[9] P. Vahidinia, B.J. Farahani, F.S. Aliee, Cold start in serverless computing: current trends and mitigation strategies, in: COINS, IEEE, 2020, pp. 1–7.
[10] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Affinity-aware serverless function scheduling, in: 22nd IEEE International Conference on Software Architecture, ICSA 2025, Odense, Denmark, March 31-April 4, 2025, IEEE, 2025.
[11] G. De 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.
[12] G. De Palma, S. Giallorenzo, J. Mauro, M. Trentin, G. Zavattaro, Webassembly at the edge: benchmarking a serverless platform for private edge cloud systems, IEEE Internet Comput. 01 (2024) 1–8, https://doi.org/10.1109/MIC.2024.3513035.
[13] A. Kuntsevich, P. Nasirifard, H.-A. Jacobsen, A distributed analysis and benchmarking framework for apache openwhisk serverless platform, in: Proc. of Middleware (Posters), 2018, pp. 3–4.
[14] M. Shahrad, J. Balkind, D. Wentzlaff, Architectural implications of function-as-a-service computing, in: Proc. of MICRO, 2019, pp. 1063–1075.
[15] A. Banaei, M. Sharifi, Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform, J. Supercomput. 9 (2021), https://doi.org/10.1007/s11227-021-04057-z.
[16] A. Suresh, A. Gandhi, Fnsched: an efficient scheduler for serverless functions, in: Proc. of WOSC@Middleware, ACM, 2019, pp. 19–24.
[17] C.L. Abad, E.F. Boza, E.V. Eyk, Package-aware scheduling of faas functions, in: Proc. of ACM/SPEC ICPE, ACM, 2018, pp. 101–106.
[18] J. Sampé, M. Sánchez-Artigas, P. García-López, G. París, Data-driven serverless functions for object storage, in: Proc. of Middleware, ACM, 2017, pp. 121–133.
[19] S. Kotni, A. Nayak, V. Ganapathy, A. Basu, Faastlane: accelerating function-as-a-service workflows, in: Proc. of USENIX ATC, USENIX Association, 2021, pp. 805–820.
[20] S. Shillaker, P. Pietzuch, Faasm: lightweight isolation for efficient stateful serverless computing, in: Proc. of USENIX ATC, USENIX Association, 2020, pp. 419–433.