Formally Verifying Function Scheduling Properties in Serverless Applications

De Palma, Giuseppe; Giallorenzo, Saverio; Mauro, Jacopo; Trentin, Matteo; Zavattaro, Gianluigi

Published in:
IT Professional

DOI:
10.1109/mitp.2023.3333071

Publication date:
2023

Document version:
Final published version

EDITOR: Tiziana Margaria, tiziana.margaria@ul.ie

COLUMN: FORMAL METHODS IN INDUSTRY

Formally Verifying Function Scheduling Properties in Serverless Applications

Giuseppe De Palma and Saverio Giallorenzo, *Università di Bologna, 40126, Bologna, Italy*
Jacopo Mauro, *University of Southern Denmark, 5230, Odense, Denmark*
Matteo Trentin and Gianluigi Zavattaro, *Università di Bologna, 40126, Bologna, Italy*

Function as a service (FaaS) is a serverless cloud execution model offering cost-efficiency, scalability, and simplified development by enabling developers to focus on code and delegate server management and application scaling to the serverless platform. Early FaaS implementations provided no control to users over function placement, but raising data locality-bound scenarios motivated new implementations with user-defined constraints over function allocations, e.g., to keep functions accessing a database close to the latter, with the aim of reducing latency, enhancing security, or complying with regulations. In this article, we show how, by leveraging the Allocation Priority Policies language—used for controlling function scheduling—and state-of-the-art planning tools, it is possible to enforce security properties and data-locality constraints, thereby guiding the definition of fine-grained serverless scheduling policies.

Function as a service (FaaS), also known as serverless functions, is a programming paradigm supported by the serverless cloud execution model.[1] In FaaS, developers implement a distributed architecture from the composition of stateless functions and delegate concerns like execution runtimes, resource allocation, and application scaling to the serverless platform. From an engineering standpoint, the serverless style advocates for functions designed to perform a specific task or provide an individual functionality in response to events such as HTTP requests, database changes, file uploads, or timer-based triggers.

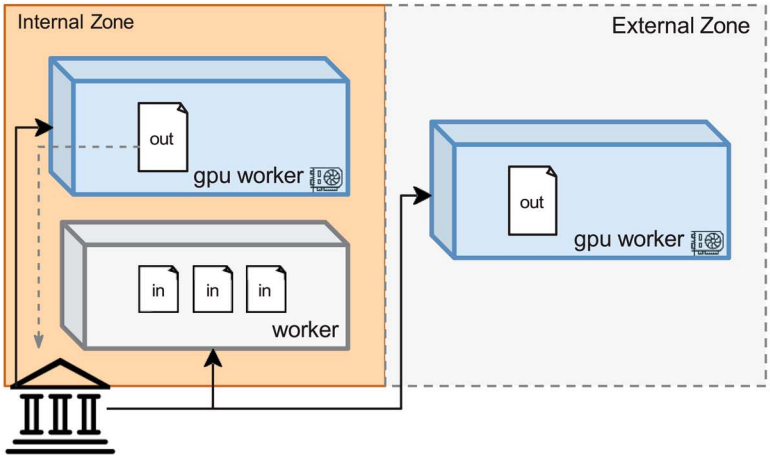
In the last decade, FaaS acquired considerable industrial traction[2,3] for several reasons:

- *Cost-efficiency*: With serverless and FaaS, users only pay for the actual execution time and resources consumed by their functions.
- *Scalability*: Serverless architectures automatically scale applications in response to varying workloads.
- *Simplified development and deployment*: FaaS allows developers to focus solely on writing code for the specific functionality they need without worrying about the underlying infrastructure.

While serverless has grown in popularity, with more and more complex use cases built/adapted as FaaS architectures, practitioners and researchers have noticed the many shortcomings of early models and implementations.[1,4] In particular, we focus on the recent call for endowing users of FaaS platforms with finer degrees of control over where (on which cloud nodes/zones) their functions run. For example, Jonas et al.[1] mention challenges like "Allow[ing] the developer to explicitly place the cloud functions on the same VM [virtual machine] instance," and "Let applications . . . co-locate the cloud functions to minimize communication overhead."

Indeed, companies are recognizing the benefits of minimizing latency, improving performance, and ensuring data security by strategically managing data placement and considering data locality as an essential factor in the design and deployment of their serverless architectures.[5] Specifically, data locality is a principle that refers to the concept of consuming/producing

1520-9202 © 2023 IEEE
Digital Object Identifier 10.1109/MITP.2023.3333071
Date of current version 12 January 2024.




```
- default:
  - workers: *
    strategy: random
- outsourced:
  - workers:
      - int_gpu
      - ext_gpu
    affinity:
      - !outsourced
- internal:
  - workers:
      - int_cpu
      - int_gpu
    affinity:
      - !outsourced
      - internal
```

FIGURE 1. The bank scenario with the outsourced (out) and internal (in) functions. The infrastructure (left) and an Allocation Priority Policies script (right).

data in proximity to where it is/will be stored. This locality principle is important for several reasons:

- *Reduced latency*: Serverless applications can minimize latency by executing functions on compute resources in proximity to where the data reside.
- *Enhanced security*: Limiting the distance (sensible) data travel to/from functions means possibly reducing their exposure to actors found in their network path.
- *Compliance and regulations*: Companies in regulated sectors may have to comply with data residency and governance regulations by ensuring that data are stored and processed within specific geographical boundaries.
- *Reduced energy consumption*: Data locality minimizes data movement, network traffic, and memory accesses, resulting in more efficient execution and reduced resource usage.

New approaches are adopted to exploit data locality. One of the current industrial trends includes applying heuristics and hints to a running system according to one or more locality principles and applying them to improve (in a best-effort fashion) the application performance.[6,7,8,9,10,11,12] These approaches, however, are not geared toward providing guarantees of the satisfaction of data locality constraints.

For this reason, here, we present recent results that address data locality constraints by explicitly expressing them through a declarative language called Allocation Priority Policies (APP).[13] By using APP, developers can have fine control over where their functions can be scheduled, providing them with support for defining and enforcing data locality constraints. Then, we provide APP users with a checker that, using state-of-the-art planning tools, allows them to statically verify their scheduling policies (e.g., security properties, like "It never happens that functions *A* and *B* run on the same node"). In the following, we present a brief introduction, guided by examples, of how we can leverage formal methods to check and guide the development of serverless applications in the presence of data locality constraints. In the following, we illustrate how formal verification tools can support expressing and checking scheduling-policy properties and guide their development through a use case of a serverless application in a bank setting.

CASE STUDY: THE BANK SCENARIO

A bank provides its digital services using a serverless platform. As shown in Figure 1 (left), the serverless infrastructure of the bank is composed of two zones: one internal and one external. The internal zone consists of workers deployed within the bank's secure network, while the external zone comprises workers hosted in a third-party public cloud environment.

Among all of the functions running in the system, a group, called `internal`, deals with processing confidential customer data, and, for security reasons, this group of functions can only run in the internal zone. Indeed, the `internal` functions involve accessing sensitive information, such as account details and transaction records, and, thus, the bank wants to restrict the

execution of this function solely to the internal zone. Moreover, since these functions access the client database, to increase performance, it would be preferable to schedule these functions on the same workers, since they can reuse already-established connections to the database—thus avoiding wasting time to establish a new connection at each function invocation.

A second group of functions, tagged as `outsourced`, was instead commissioned by the bank to a third-party developer to use machine learning techniques for building various prediction models. These functions require specialized hardware for efficient execution, and, therefore, both zones have some workers equipped with GPUs, which can accelerate the functions' execution. To save costs, the bank would prioritize running these functions on internal workers (with GPUs), but it is willing to also use external cloud resources if the internal nodes are overloaded. To keep the maximum level of security and avoid any possible leakage of sensitive information, the bank requires the outsourced code to never run on a node in which an `internal` function is running. (Outsourced code could, indeed, contain malicious code that escapes the containerized environment of the serverless framework or use side channels to potentially leak the sensitive information process by the internal functions.)

APP: SPECIFYING POLICY CONSTRAINTS

We can express scheduling constraints like the ones described for the internal and outsource functions with APP, which allows users to define scheduling policies in a concise, declarative manner.

APP follows the current trend in DevOps and cloud tools (e.g., Kubernetes) by adopting a YAML Ain't Markup Language-compatible syntax. Function-specific policies are associated with tags, allowing the platform's scheduling to pair each function with its corresponding APP policy at runtime.

An example of policies to capture the requirements of the bank functions is presented on the right side of Figure 1. As can be seen, each policy (the tags) consists of a list of one or more blocks. Each block identifies a list of workers (the nodes where a function can be allocated) by their labels. Then, each block has one or more optional parameters: the scheduling strategy for worker selection, an invalidate condition, and potential affinity rules—in particular, the last of these specify whether a function requires or needs to avoid nodes that run functions associated with a specific tag. If a selected worker is invalid (e.g., it is at maximal capacity), other available workers in the block are tried. When all workers in a block are invalid, the scheduling proceeds on the subsequent blocks. If no block provides a valid worker, a follow-up clause determines the action: fail or try to allocate the function following the logic of the default tag (the same used by untagged functions).

As a first attempt at implementing the requirements of the bank case, in the APP script in Figure 1, we allow the `internal` functions to be allocated on nodes tagged with `int_gpu` and `int_cpu`, which, respectively, identify the internal nodes with and without a GPU. Moreover, the `internal` functions declare an affinity for other functions tagged as `internal` and anti-affinity (denoted with `!`) for functions tagged as `outsourced`.

PDDL: CHECKING POLICY PROPERTIES

While APP allows users to specify complex and versatile policies, it is not trivial to assert whether a given set of policies implements the properties in the mind of the developer. Reasoning on the behavior of the platform under a certain APP script can be difficult, making it hard to figure out whether the current configuration can lead to undesired deployments, e.g., if it never happens that two specific functions can end up together on the same worker.

To help users check that their APP scripts exclude said undesired deployments, we encoded the semantics of APP constructs as an automated planning task, using Planning Domain Definition Language (PDDL)[14]—the de facto standard for encoding planning problems. PDDL specifications consist of a *domain* file and one or more *problem* files. The former contains all available actions, each with its preconditions and effects (i.e., how the action affects the state); the latter files contain the initial state of all variables and the desired goal of a specific planning problem instance. Multiple problems can refer to the same domain.

In detail, we defined the PDDL domain following the APP specification to describe how an APP-based serverless platform allocates functions on workers. Then, an encoder from APP to PDDL problems allows us to analyze user-defined scripts using pre-existing, efficient planners.

With this approach, we can automatically verify whether a given platform configuration—its nodes and APP script—can lead to (un)desired states. Moreover, in case the planner can find a solution, this comes as a sequence of actions that lead to the goal, which the developers can use to understand what steps led to that (positive/negative) state.

In the bank case, possible tests that encode the properties of the case are as follows (expressed in plain English):

1) Can `outsourced` functions be scheduled?
2) Can `internal` functions be scheduled?
3) Can `outsourced` and `internal` functions be scheduled on the same worker?
4) Can `outsourced` functions be scheduled on a worker without a GPU?
5) Can two `internal` functions be scheduled on two separate workers?

Tests 1 and 2 ensure that both groups of functions can execute and are considered satisfied if the planner finds a sequence of actions that schedule them on a node. Tests 3 and 4, respectively, check whether functions of different groups are kept separate and if outsourced ones only appear on certain nodes. We consider these tests passed if they cannot be satisfied; i.e., if the planner finds no solution. Finally, test 5 checks whether the platform respects the requirement that internal functions are scheduled on the same node to share connections to the database. We express this property as whether it happens that two `internal` functions are scheduled on different workers; thus, this test is *passed* if the planner finds no solution.

Encoding these tests in PDDL and running them against the script in Figure 1 shows that the planner finds a sequence of steps for all of them, thereby failing tests 3–5. Hence, the checker tells us that the script in the figure does not guarantee three of the five properties we encoded.

As mentioned, the checker provides the plans that violate the properties, which we use as hints to fix the script; e.g., consider the following plan found checking test 3:

```
(ADD_F_TO_W_WITH_BLOCK INTERNAL
  INT_GPU DEFAULT_B 0)
(ADD_F_TO_W_WITH_BLOCK
  OUTSOURCED INT_GPU OUTSOURCED_B 0)
(REACH - GOAL)
```

The trace shows that `internal` can use the `default` policy, which allows the functions tagged `internal` to ignore affinity constraints. (`DEFAULT_B0` is the first block of the `default` tag.) The issue is that we missed specifying that, if we cannot find an available worker in the internal zone, we shall fail the allocation and avoid using the `default` policy.

This issue shows how even a brief, simple script can hide some hard-to-spot corner cases and how the planner helps in finding which steps bring the platform to undesired states. Thus, we can use the planner to check the properties and improve the APP script incrementally, with each step building on the previous step's failed tests.

In Table 1, from left to right, we show a possible sequence of changes guided by the planner. The first column of the table shows that preventing the `internal` functions from using the default policy is not enough. Indeed, test 2 fails because we are preventing the allocation of `internal` functions due to their affinity constraint. In the second column, we patch this bug by adding a second block, under the `internal` tag, without that affinity constraint. Unfortunately, this update also falls short. Test 3 fails because we can schedule `internal` and `outsourced` functions together.

We add the missing constraint in the third column of Table 1, finally satisfying all of the tests except for test 5. This outstanding failing test allows us to notice how, besides debugging APP scripts, one can use the

TABLE 1. Top down from the first row: the incremental improvements of the APP scripts from Figure 1, the results of the tests (✓ for passed and ✗ for failed), and an explanation of the main failing tests.*

`. . .` `- outsourced :` `  . . .` `  followup : fail` `- internal:` `  . . .` `  followup : fail`					`. . .` `- internal:` `  . . .` `  - workers:` `  - int_cpu` `  - int_gpu` `  affinity :` `  - "! outsourced "`					`. . .` `- outsourced :` `  . . .` `  - affinity :` `  - "! internal "` `. . .`				
T1: ✓	T2: ✗	T3: ✓	T4: ✓	T5: ✓	T1: ✓	T2: ✓	T3: ✗	T4: ✓	T5: ✗	T1: ✓	T2: ✓	T3: ✓	T4: ✓	T5: ✗
We add `followup: fail` to `internal` and `outsourced`. This breaks test 2, as now the former requires itself to already be on the targeted worker and has no default to fall back to. (Tests 3 and 5 pass because `internal` is never scheduled.)					We add a fallback block for `internal`, removing the self-affinity. Now, `internal` functions can be scheduled using this block. Test 3 fails since `outsourced` does not list `internal` as anti-affinity (which is not symmetric in APP).					We add `!internal` anti-affinity to `outsourced`, and we see that all tests pass except for test 5.				

*APP: Allocation Priority Policies; T: test.

checker to also reason on the properties one imposes on a system. Indeed, for test 5, the plan found by the checker is to first overload one of the workers and then schedule the function on the other. This is a desirable situation since, otherwise, we would not be able to fully exploit the computational resources at our disposal. Reasoning on the purpose of the property behind test 5, we can argue that it enforces an antipattern for cloud deployments and, by extension, a property we do not need to hold in the system.

CONCLUSION

We illustrated how formal methods can help check and improve APP scripts, identifying potential corner cases and guiding incremental refinements. These methods provide insights into the behavior of serverless platforms and help identify undesired deployments, ensuring the reliable and secure execution of functions.

APP is a growing language; e.g., it was recently extended to deal with topology-aware scheduling policies.[15] In the future, further exploration of formal verification techniques can enable the specification and verification of even more complex policies that can go beyond scheduling (e.g., properties on the workflows specified as durable/step functions[16]). We also plan to study how to integrate these techniques into the development process of serverless applications, providing developers with tools and frameworks integrated within their continuous integration/continuous delivery pipeline.

Our technical results show that, by bringing powerful formal methods to FaaS, developers will have tools to reason about and enforce critical properties and constraints on their applications, paving the way for the development of robust, secure, and efficient serverless architectures.

ACKNOWLEDGMENTS

Open Access funding provided by 'Alma Mater Studiorum - Università di Bologna' within the CRUI CARE Agreement.

REFERENCES

1. E. Jonas et al., "Cloud programming simplified: A Berkeley view on serverless computing," Univ. of California at Berkeley, Berkeley, USA, Tech. Rep. UCB/EECS-2019-3, Feb. 2019.
2. H. B. Hassan et al., "Survey on serverless computing," *J. Cloud Comput.*, vol. 10, pp. 1–29, Jul. 2021, doi: 10.1186/s13677-021-00253-7.
3. J. Schleier-Smith et al., "What serverless computing is and should become: The next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, pp. 76–84, 2021, doi: 10.1145/3406011.
4. J. M. Hellerstein et al., "Serverless computing: One step forward, two steps back," in *Proc. Conf. Innov. Data Syst. Res. (CIDR)*, Jan. 2019. [Online]. Available: www.cidrdb.org
5. S. Hendrickson et al., "Serverless computation with openLambda," in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2016, pp. 33–39, doi: 10.5555/3027041.3027047.
6. D. Kelly, F. Glavin, and E. Barrett, "Serverless computing: Behind the scenes of major platforms," in *Proc. EEE 13th Int. Conf. Cloud Comput. (CLOUD)*, 2020, pp. 304–312, doi: 10.1109/CLOUD49709.2020.00050.
7. A. Banaei and M. Sharifi, "ETAS: PREDICTIVE scheduling of functions on worker nodes of Apache OpenWhisk platform," *J. Supercomput.*, vol. 78, no. 4, pp. 5358–5393, Sep. 2021, doi: 10.1007/s11227-021-04057-z.
8. L. Baresi and G. Quattrocchi, "PAPS: A serverless platform for edge computing infrastructures," *Frontiers Sustain. Cities*, vol. 3, Jul. 2021, Art. no. 690660, doi: 10.3389/frsc.2021.690660.
9. Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Principles*, New York, NY, USA: ACM, 2021, pp. 691–707, doi: 10.1145/3477132.3483541.
10. S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *Proc. USENIX Annu. Tech. Conf.*, Berkeley, CA, USA: USENIX Association, 2021, pp. 805–820.
11. C. P. Smith et al., "FaDO: FaaS functions and data orchestrator for multiple serverless edge-cloud clusters," in *Proc. IEEE 6th Int. Conf. Fog Edge Comput. (ICFEC)*, 2022, pp. 17–25, doi: 10.1109/ICFEC54809.2022.00010.
12. M. Abdi et al., "Palette load balancing: Locality hints for serverless functions," in *Proc. 18th Eur. Conf. Comput. Syst.*, 2023, pp. 365–380, doi: 10.1145/3552326.3567496.
13. G. De Palma et al., "Allocation priority policies for serverless function-execution scheduling optimisation," in *Proc. Int. Conf. Service-Oriented Comput. (ICSOC)*, E. Kafeza, Ed., Cham, Switzerland: Springer-Verlag, 2020, vol. 12571, pp. 416–430, doi: 10.1007/978-3-030-65310-1_29.
14. M. Fox and D. Long, "PDDL2. 1: An extension to PDDL for expressing temporal planning domains," *J. Artif. Intell. Res.*, vol. 20, pp. 61–124, Dec. 2003, doi: 10.1613/jair.1129.
15. G. De Palma et al., "A declarative approach to topology-aware serverless function-execution scheduling," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, C. A. Ardagna, Ed., 2022, pp. 337–342, doi: 10.1109/ICWS55610.2022.00056.

16. S. Burckhardt et al., "Durable functions: Semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–27, 2021, doi: 10.1145/3485510.

GIUSEPPE DE PALMA is a Ph.D. student at the Università di Bologna, 40126, Bologna, Italy. Contact him at giuseppe.depalma2@unibo.it.

SAVERIO GIALLORENZO is an assistant professor at Università di Bologna, Italy, and a member of the Institut National de Recherche en Informatique et en Automatique (INRIA) Foundations of Component-Based Ubiquitous Systems (FOCUS) team, Sophia Antipolis, INRIA. Contact him at saverio.giallorenzo2@unibo.it.

JACOPO MAURO is a professor with the Department of Mathematics and Computer Science at the University of Southern Denmark, 5230, Odense, Denmark. Contact him at mauro@imada.sdu.dk.

MATTEO TRENTIN is a Ph.D. student at Università di Bologna, 40126, Bologna, Italy, under a cotutelle with the University of Southern Denmark 5230, Odense, Denmark. Contact him at matteo.trentin2@unibo.it.

GIANLUIGI ZAVATTARO is a professor with the Department of Computer Science and Engineering at the University of Bologna, 40126, Bologna, Italy, and a member of the INRIA FOCUS team, Sophia Antipolis, INRIA, France. Contact him at gianluigi.zavattaro@unibo.it.