Function-as-a-Service Allocation Policies Made Formal

Giuseppe De Palma[1,2], Saverio Giallorenzo[1,2], Jacopo Mauro[3], Matteo Trentin[1,2,3], and Gianluigi Zavattaro[1,2] *

[1] Alma Mater Studiorum - Università di Bologna, Italy
[2] OLAS team INRIA, France
[3] University of Southern Denmark, Denmark

Abstract. Function-as-a-Service (FaaS) is a Serverless Cloud paradigm where a platform manages the execution scheduling (e.g., resource allocation, runtime environments) of stateless functions. Recent developments demonstrate the benefits of using domain-specific languages to express per-function scheduling policies, e.g., enforcing the allocation of functions on nodes that enjoy low data-access latencies thanks to proximity and connection pooling. In this paper, we consider APP, one of the languages proposed to specify *Allocation Priority Policies* in FaaS implemented on top of the popular OpenWhisk serverless platform. The aim of our operational semantics is twofold: on the one hand, it represents the underlying substrate necessary for the application of formal analysis techniques, on the other hand, it can drive consistent implementations of APP on top of the numerous serverless platforms recently proposed.

1 Introduction

Function-as-a-Service (FaaS) is a programming paradigm of the Serverless Cloud execution model [36]. In FaaS, developers implement a distributed architecture by composing stateless functions and delegate concerns like execution runtimes and resource allocation to the platform, thus focusing on implementing the business logic. FaaS solutions include those of the main cloud [5, 13, 52] and open-source alternatives [21, 33, 47, 48].

A common denominator of these platforms is that they manage the allocation of functions over the available computing resources, also called *workers*, following opinionated policies that favour some performance principle. Indeed, effects like *code locality* [33]—due to latencies in loading function code and runtimes—or *session locality* [33]—due to the need to authenticate and open new sessions to interact with other services—can substantially increase the run time of functions.

* This work has been partially supported by the research project FREEDA (CUP: I53D23003550006) funded by the framework PRIN 2022 (MUR, Italy), the French ANR project SmartCloud ANR-23-CE25-0012, and from ICSC – Centro Nazionale di Ricerca in High Performance Computing, Big Data and Quantum Computing, funded by European Union – NextGenerationEU.

The breadth of the design space of serverless scheduling policies is witnessed by the growing literature focused on techniques that mix one or more of these locality principles to increase the performance of function execution, assuming some locality-bound traits of functions [2, 37–39, 51, 57]. Besides performance, functions can have functional requirements that the scheduler shall consider. For example, users can ward off allocating their functions alongside "untrusted" ones—common threat vectors in serverless are limited function isolation and the ability of functions to (surreptitiously) gather weaponisable information on the runtime, the infrastructure, and the other tenants [4, 8, 14, 59].

Although one can mix different principles to expand the profile coverage of a given platform-wide scheduler policy, the latter hardly suits all kinds of scenarios. This shortcoming motivated the introduction of a domain-specific, platform-agnostic, declarative language, called *Allocation Priority Policies* (APP), for specifying custom function allocation policies [15, 19]. Thanks to APP, the same platform can support different scheduling policies, each tailored to meet the specific needs of a set of related functions. We validated APP by implementing a serverless platform as an extension of the open-source Apache OpenWhisk project [19] and showing that our APP-based version does not introduce a significant overhead w.r.t. the vanilla one, while APP custom function scheduling policies can improve function performance in locality-bound scenarios.

Contributions. In this paper, we address the problem of formally defining the semantics of APP. The objective is twofold. On the one hand, to apply formal analysis techniques we need to formalise the behaviour of FaaS systems following the allocation policies prescribed by a given APP script. For instance, security analysis could require checking whether the scheduler can allocate a trusted function on a worker where an untrusted one is running (and vice versa), or liveness analysis could require checking whether the scheduler allocates a given safety-critical function on a reliable worker every time it is required. On the other hand, we see our formal semantics as a specification for driving APP implementations. In fact, OpenWhisk is only one of the many open-source platforms supporting Function-as-a-Service, and we foresee APP implemented on top of other popular platforms such as OpenFaaS, Fission, or Knative [1, 21, 46]. The consistency of several implementations of APP is fundamental to support platform-agnostic analysis of the correctness of FaaS applications.

Lessons learned from Rocco De Nicola. There is a third motivation behind this contribution: we present it to the ReoCAS Colloquium organised in honour of the 70th birthday of Rocco De Nicola. Rocco has significantly contributed to the development of formal analysis techniques of programming languages, especially in the context of Concurrent Systems (for which Rocco developed the notion of testing equivalences for processes [44]), Coordination languages (see e.g., the well-known KLAIM language [43]) and Autonomic Systems (where Rocco contributed by proposing the SCEL language [45]). Some authors of this paper collaborated with Rocco on several national and international projects, in particular, the European Project SENSORIA—Software Engineering for Service-

Oriented Overlay Computers. Rocco covered the role of leader of a work package dedicated to the definition of the formal semantics of process calculi for service-oriented computing. The task was challenging due to its contrasting objectives yet critical for the overall success of the project: the calculi had to be at the same time simple enough to capture the essence of service-oriented computing and support the formal investigation of its basic properties, but also sufficiently expressive to allow for the modelling of typical programming patterns used in the development of service-oriented systems. Under the supervision of Rocco, the Service Centered Calculus (SCC) [10] was defined as a common effort among several research units. In the context of the same work package, our Bologna unit concentrated also on other aspects, in particular, correlation-based instead of session-based communication, and defined another calculus, called SOCK—Service-Oriented Computing Kernel [32]. The SOCK calculus supported the formal investigation of the relationship between choreographies and orchestrations in Service Oriented Computing [11], but it also drove the development of the Jolie programming language [42], which is still actively developed, used in the industry (spearheaded by italianaSoftware S.r.l), taught in textbooks [26] and university courses (e.g., in Denmark, Italy, and Russia), and the subject of fruitful research [22,24,25,28–31].

This work represents a natural continuation of this approach in the novel context of Serverless Computing and FaaS. We aim to capture the foundational properties of this new programming paradigm (in particular, aspects related to the scheduling of functions), formalise such properties to support formal verification techniques, and develop languages that are amenable to formal verification and expressive enough for programming and developing FaaS systems.

Structure of the paper. In Sec. 2, we introduce APP by presenting its syntax and discussing informally its semantics. In Sec. 3, we report the novel contribution of this paper: an operational semantics for APP. We discuss related work in Sec. 4 and draw concluding remarks in Sec. 5.

2 An Informal Introduction to APP

We dedicate this section to introduce APP [15,19]. We report and comment on its syntax and describe informally its semantics, formalised in Sec. 3.

The syntax of APP, reported in Fig. 1, draws inspiration from YAML [60], a renowned data-serialisation language for configuration files—e.g., many modern tools use the format, like Kubernetes, Ansible, and Docker. While APP scripts are YAML-compliant, in this paper, we use a slightly stylised version of the APP syntax to increase readability (e.g., we omit quotes around strings such as `*` instead of `"*"`). For simplicity, we also assume two minor variations. First, we cover the whole syntax of APP [19] except for two options, `strategy: platform` and `invalidate: overload`, which we avoid modelling within our generic reference since they capture platform-specific rules (e.g., in Apache OpenWhisk `platform` implies the usage of its hardwired strategy, based on a co-prime heuristic selection logic [19]). Second, we rename the `random` strategy option to `any`, to point out that

$$id \in Identifiers \qquad n \in \mathbb{N}$$

$$
\begin{array}{lll}
app & ::= & \overline{-tag} \\
tag & ::= & id : \overline{- block} \quad \texttt{followup} : f\_opt \\
block & ::= & \texttt{workers} : w\_opt \quad \texttt{strategy} : s\_opt \quad \texttt{invalidate} : \overline{- i\_opt} \\
w\_opt & ::= & * \mid \overline{- id} \\
s\_opt & ::= & \texttt{any} \mid \texttt{best\_first} \\
i\_opt & ::= & \texttt{capacity\_used}\ n\% \mid \texttt{max\_concurrent\_invocations}\ n \\
f\_opt & ::= & \texttt{default} \mid \texttt{fail}
\end{array}
$$

Fig. 1. APP syntax.

we model a non-deterministic worker selection instead of a uniformly-distributed one, as `random` does. This avoids introducing a more involved quantitative semantics to capture probability distributions, which we leave as future work. Looking at Fig. 1 (and for the remainder of the paper), we indicate syntactic units in *italics*, optional fragments in *grey*, terminals in `monospace`, and lists with $\overline{bars}$.

The idea behind APP's approach to the declarative specification of function scheduling is that functions have associated a tag that identifies some scheduling policies. An APP script represents: *i)* named scheduling policies identified by a *tag* and *ii)* policy *block*s that indicate either some collection of workers, each identified by a worker *id*, or the universal `*`. To schedule a function, we use its tag to retrieve the scheduling policy that includes one or more blocks of possible workers. To select the worker, we iterate top-to-bottom on the blocks. We stop at the first block that has a non-empty list of valid workers and then select one of those workers according to the strategy defined by the block (described later).

Each APP script contains a trailing `default` policy which defines the scheduling logic of non-tagged functions. Each policy can define a `followup` clause, which specifies what to do if the scheduling of the functions fails: `fail` terminates the scheduling while `default` lets the scheduler try to allocate the function following the `default` policy. Each block can define a `strategy` for worker selection—`any` selects non-deterministically one of the workers in the list and `best_first` selects the first worker in the list—and a list of constraints that `invalidate` a worker—`capacity_used` invalidates a worker if its resource occupation reaches the set threshold and `max_concurrent_invocations` invalidates a worker if it hosts more than the specified number of functions.

Fig. 2 shows an APP policy for functions tagged `f_tag`. The policy has one block which restricts the allocation of the function on the workers labelled `local_w1` and `local_w2`. Moreover, the block specifies as invalid (for hosting the function under scheduling) those workers that reach a memory consumption above 80%. Since the strategy is `best_first`, we allocate the function on the first valid

```
- f_tag:
  - workers:
    - local_w1
    - local_w2
    strategy: best_first
    invalidate:
    - capacity_used 80%
    followup: fail
```

Fig. 2. Example APP script.

worker; if none are valid, the scheduling of the function `fail`s, without trying other policies.

3 Operational Semantics of APP

Now that we informally introduced APP and its behaviour, we proceed to formalise it. We define the behaviour of APP scripts as a labelled transition system (LTS) operational semantics. In the definition of the LTS, we use these domains, structures, and functions:

$$
\begin{aligned}
f &\in \mathcal{F} & w &\in \mathcal{W} \subset \mathit{Identifiers} \\
t &\in \mathcal{T} \subset \mathit{Identifiers} & C &\in \mathcal{C} \triangleq \mathcal{W} \to \mathit{Multiset}(\mathcal{F}) \times \mathbb{N} \times \mathbb{N} \\
\mathit{reg} &\in \mathcal{F} \to \mathbb{N} \times \mathcal{T} & p &\in \mathcal{P} \triangleq \mathcal{T} \to \mathit{List}(\mathcal{B}) \\
[\![\cdot]\!] &: \mathit{app} \to \mathcal{P} & b &\in \mathcal{B} \triangleq (\mathit{List}(\mathcal{W}) \cup *) \times \mathit{s\_opt} \times \mathit{List}(\mathit{i\_opt})
\end{aligned}
$$

We use $\mathcal{W}$, ranged over by w, to denote the set of workers, while $\mathcal{F}$, ranged over by f, denotes the set of functions. We use $\mathcal{C}$, ranged over by C, to denote the set of platform configurations. A configuration associates each of its workers (in $\mathcal{W}$) with a triple relating the multiset of functions ($\mathit{Multiset}(\mathcal{F})$) currently allocated on that worker, the amount of resources (in $\mathbb{N}$) used by such functions, and the maximal amount of resources (also in $\mathbb{N}$) available to that worker. Functions are tagged to associate them with a scheduling policy. We use $\mathcal{T}$, ranged over by t, to denote the set of tags and define *reg* (short for registry) as a map that associates each function with its tag and its occupancy, i.e., the amount of resources needed to host it. $\mathbb{N}$ represent the natural numbers—even considering fractional resources, we deem naturals fine-grained enough for our purpose since we can always convert these to $\mathbb{N}$ with a constant multiplying factor. We use *app* to denote the set of APP scripts that follow the grammar presented in Fig. 1.

From YAML to APP Semantics' Structures In our formal model of APP, we need to represent scripts as mathematical objects. Formally, we define a straightforward encoding $[\![\cdot]\!]$ that, given a script in *app* (which always terminates with the `default`-tagged policy as previously described), returns a policy function p (ranging over the set $\mathcal{P}$) with all `followup`s unfolded—where `default` always `fail`s. The encoding, reported in Fig. 3, inductively walks through the syntax of the APP script and translates each fragment into the corresponding mathematical object in $\mathcal{P}$. The only notable bits of the encoding regard the inclusion of the standard options for the missing parameters—for `strategy` we set it to `any` and for `invalidate` we set it to the maximal capacity of the worker, i.e., `capacity_used` 100`%`—and the static resolution of the `followup` parameter, where we concatenate the list of blocks of the tag with the blocks of the `default` one, in case the

$$[\![\overline{-\ tag}\ ::\ -\ \texttt{default:}\overline{-\ block}]\!] = [\![\overline{-\ tag}\ ::\ -\ \texttt{default:}\overline{-\ block}\ \ \texttt{followup: fail}]\!]$$

$$[\![\overline{-\ tag}\ ::\ -\ \texttt{default:}\overline{-\ block}\ \ \texttt{followup: } f\_opt]\!] = \bigcup_{t\ \in\ \overline{tag}} \left\{ [\![t]\!]_{\overline{[block]}} \right\}$$

$$[\![id : \overline{-\ block}]\!]_{\overline{b}} = [\![id : \overline{-\ block}\ \ \texttt{followup : default}]\!]_{\overline{b}} = (id, \overline{[block]} :: \overline{b})$$

$$[\![id : \overline{-\ block}\ \ \texttt{followup : fail}]\!]_{\overline{b}} = (id, \overline{[block]})$$

$$[\![\texttt{workers: } w\_opt]\!] = ([\![w\_opt]\!], \texttt{any}, \texttt{capacity\_used } 100\% :: \varepsilon)$$

$$[\![\texttt{workers: } w\_opt\ \ \texttt{strategy : } s\_opt]\!] = ([\![w\_opt]\!], s\_opt, \texttt{capacity\_used } 100\% :: \varepsilon)$$

$$[\![\texttt{workers: } w\_opt\ \ \texttt{invalidate : } \overline{-\ i\_opt}]\!] = ([\![w\_opt]\!], \texttt{any}, \overline{i\_opt})$$

$$[\![\texttt{workers: } w\_opt\ \ \texttt{strategy : } s\_opt\ \ \texttt{invalidate : } \overline{-\ i\_opt}]\!] = ([\![w\_opt]\!], s\_opt, \overline{i\_opt})$$

$$[\![*]\!] = *$$

$$[\![\overline{-\ id}]\!] = \overline{id}$$

Fig. 3. APP Syntax Encoding.

`default` option is present. Notation-wise, we introduce :: and ε to resp. indicate list concatenation and empty sequence (frequently omitted for brevity).

LTS Rules and Examples We can now present and comment on the rules of the LTS on configurations, reported in Fig. 4. The semantics has three layers: *a)* Configuration rules (we prefix their names with C), *b)* Blocks rules (prefixed with B), and *c)* Workers rules (prefixed with W). At the bottom of Fig. 4, we define the auxiliary relations **strategy** and **valid**, used by the Workers rules to resp. check if a worker can be selected according to a given strategy and if the allocation of the function does not violate any constraint on the selected worker.

Configuration Rules. The LTS in Fig. 4 has three kinds of labels, ranged over by λ: $(start, f, w)$ indicates the allocation of an instance of function f on the worker w, $(done, f, w)$ denotes the deallocation of f on w, and $(fail, f)$ traces the failure to schedule f on the current configuration. Recalling the C_0 and C_1 presented above, the labelled transition $C_0 \xRightarrow{(start, f, \texttt{w1})} C_1$ represents a reduction from the configuration C_0 to C_1 upon the allocation of f on the worker `w1`.

We focus on the two actions that can change the state of a given configuration: the allocation of a new function instance ($[C_{start}]$) and the deallocation of a function instance ($[C_{done}]$)—rule $[C_{fail}]$ tracks failed scheduling attempts, but has no effects on configurations. The rules use the function-update notation $C[\cdot \mapsto \cdot]$ to model the allocation and removal of functions on workers. Specifically, in $[C_{start}]$ we allocate the function by joining the multiset σ of allocated functions on the worker w with the new function instance f. We also update the current occupancy of the worker with the units of the function. In $[C_{done}]$, we remove the function from the worker by subtracting one instance from σ and removing its units from the current occupancy of the worker.

As an example, consider the APP script in Fig. 2 and three configurations C_0, C_1, C_2. We consider an infrastructure that includes two workers: `w1` with maximal capacity 10 and `w2` with maximal capacity 20 and we assume that f takes 8 units. Let C_0 be the configuration where the workers have no functions

Workers Layer

$$[W_{first}]\frac{\textbf{strategy}(\overline{w}, s) = w \quad w \neq \bot \quad \textbf{valid}(f, w, \overline{i}, C)}{f, \overline{w}, s, \overline{i}, C \triangleright w} \qquad [W_{end}]\frac{\textbf{strategy}(\overline{w}, s) = \bot}{f, \overline{w}, s, \overline{i}, C \triangleright \bot}$$

$$[W_{next}]\frac{\textbf{strategy}(\overline{w}, s) = w \quad w \neq \bot \quad \neg\textbf{valid}(f, w, \overline{i}, C) \quad f, \overline{w} \setminus w, s, \overline{i}, C \triangleright w'}{f, \overline{w}, s, \overline{i}, C \triangleright w'}$$

Blocks Layer

$$[B_{one}]\frac{\overline{w'} = \overline{w} \cap \mathrm{dom}(C) \quad f, \overline{w'}, s, \overline{i}, C \triangleright w}{f, C, (\overline{w}, s, \overline{i}) \to w} \qquad [B_{star}]\frac{f, \overline{\mathrm{dom}(C)}, s, \overline{i}, C \triangleright w}{f, C, (*, s, \overline{i}) \to w}$$

$$[B_{first}]\frac{f, C, b_1 \to w \quad w \neq \bot}{f, C, b_1 :: \cdots :: b_n \to w} \qquad [B_{next}]\frac{f, C, b_1 \to \bot \quad f, C, b_2 :: \cdots :: b_n \to w}{f, C, b_1 :: b_2 :: \cdots :: b_n \to w}$$

Configuration Layer

$$[C_{start}]\frac{reg(f) = (n, t) \quad p(t) = \overline{b} \quad f, C, \overline{b} \to w \quad w \neq \bot \quad C(w) = (\sigma, n', m)}{C \xrightarrow{(start, f, w)} C[w \mapsto (\sigma \cup \{f\}, n' + n, m)]}$$

$$[C_{fail}]\frac{reg(f) = (\cdot, t) \quad p(t) = \overline{b} \quad f, C, \overline{b} \to \bot}{C \xrightarrow{(fail, f)} C}$$

$$[C_{done}]\frac{w \in \mathrm{dom}(C) \quad f \in \sigma \quad reg(f) = (n, \cdot) \quad C(w) = (\sigma, n', m)}{C \xrightarrow{(done, f, w)} C[w \mapsto (\sigma \setminus \{f\}, n' - n, m)]}$$

strategy relation and **valid** predicate

$$\textbf{strategy}(\overline{w}, s) = \begin{cases} w & \text{if } s = \texttt{any} \wedge w \in \{\overline{w}\} \\ w & \text{if } s = \texttt{best\_first} \wedge \overline{w} = w :: \overline{w'} \\ \bot & \text{otherwise} \end{cases}$$

$$\textbf{valid}(f, w, i_1 :: \cdots :: i_n, C) = \textbf{valid}(f, w, i_1, C) \ \wedge \ \cdots \ \wedge \ \textbf{valid}(f, w, i_n, C)$$

$$\textbf{valid}(f, w, i, C) = \begin{cases} \texttt{true} & \text{if } i = \texttt{capacity\_used } n\texttt{\%} \\ & \quad \wedge\ reg(f) = (n_f, \cdot) \wedge\ C(w) = (\cdot, n_{cur}, n_{max}) \\ & \quad \wedge\ \min(n, 100) \geq 100 * (n_{cur} + n_f)/n_{max} \\ \texttt{true} & \text{if } i = \texttt{max\_concurrent\_invocations } n \\ & \quad \wedge\ \textbf{valid}(f, w, \texttt{capacity\_used } 100\texttt{\%}, C) \\ & \quad \wedge\ C(w) = (\sigma, \cdot, \cdot) \wedge\ n \geq |\sigma| + 1 \\ \texttt{false} & \text{otherwise} \end{cases}$$

Fig. 4. strategy and **valid** functions.

allocated on them. To obtain C_1 we start from C_0 and allocate f on `w1`. C_2 is a reduction from C_1, where we schedule another time f; since `w1` is full (it cannot

$$[C_{start}]\ \frac{reg(f) = (8, \texttt{f\_tag}) \quad p(\texttt{f\_tag}) = block_f :: \varepsilon \quad [B_{first}]\ \dfrac{Eq.\ (2)}{f, C_0, block_f :: \varepsilon \to \texttt{w1}} \quad \texttt{w1} \neq \bot \quad C_0(\texttt{w1}) = (\emptyset, 0, 10)}{C_0 \xrightarrow{(start, f, \texttt{w1})} C_0[\texttt{w1} \mapsto (\emptyset \cup \{f\}, 0 + 8, 10)]} \tag{1a}$$

$$[C_{start}]\ \frac{reg(f) = (8, \texttt{f\_tag}) \quad p(\texttt{f\_tag}) = block_f :: \varepsilon \quad [B_{first}]\ \dfrac{\vdots}{f, C_1, block_f :: \varepsilon \to \texttt{w2}} \quad \texttt{w2} \neq \bot \quad C_1(\texttt{w2}) = (\emptyset, 0, 20)}{C_1 \xrightarrow{(start, f, \texttt{w2})} C_1[\texttt{w2} \mapsto (\emptyset \cup \{f\}, 0 + 8, 20)]} \tag{1b}$$

host f due to the invalidation conditions of `f_tag`), we allocate f on `w2`. Formally:

$$C_0 = \{\ \texttt{w1} \mapsto (\emptyset,\ 0,\ 10),\ \texttt{w2} \mapsto (\emptyset,\ 0,\ 20)\ \}$$

$$C_1 = \{\ \texttt{w1} \mapsto (\{f\},\ 8,\ 10),\ \texttt{w2} \mapsto (\emptyset,\ 0,\ 20)\ \}$$

$$C_2 = \{\ \texttt{w1} \mapsto (\{f\},\ 8,\ 10),\ \texttt{w2} \mapsto (\{f\},\ 8,\ 20)\ \}$$

We illustrate the rules by considering the allocation of an instance of function f on a worker, where p is the policy function obtained from the encoding of the considered APP script in Fig. 2, and the configuration is C_0. The reduction, reported in Eq. (1a), happens via rule $[C_{start}]$. There, f has tag `f_tag` and p maps it to $block_f$ (cf. Sec. 3). The third premise uses the Blocks rules ($[B_{first}]$, derived in Eq. (2)) to find a valid (non-$\bot$) worker (`w1`). We update C_0 by allocating f on `w1`. In Eq. (1b), we want to allocate f given configuration C_1. Since `w1` has insufficient capacity (it has 8/10 of its capacity occupied by an instance of f) rule $[C_{start}]$ selects the second worker (`w2`)—obtaining configuration C_2.

Blocks Rules. The rules in the blocks layer embody the logic of block allocation unfolding. Briefly, we pick the first block ($[B_{first}]$), and check if we can find an eligible worker for the allocation within that block—either given a list of workers ($[B_{one}]$) or the universal ($[B_{star}]$). If that is not the case, we continue by unfolding the list of blocks following their order of appearance ($[B_{next}]$). Notice that, in rule $[B_{star}]$, we abuse the list notation $\overline{\text{dom}(C)}$ to indicate the transformation of the domain of C into a list of workers—the transformation is idempotent, i.e., the transformations of the same set always result in the same sequence of elements.

Continuing our examples, Eq. (2) shows the evaluation of $block_f$ started in Eq. (1a)—on the left of the operator $\to$. We replaced $block_f$ with its components. *Nomen omen*, the rule $[B_{first}]$ successfully finds a valid worker from the first (and only) block, using the rule $[B_{one}]$. In that rule, we find the candidate workers at the intersection between those provided by C_0 and the ones listed in the block (`w1 :: w2`). Then, we use the selection rules ($[W_{first}]$, derived in Eq. (3a)) to find a valid worker. We omit to report the reduction for the second scenario on C_1 since it is similar to the previous one at blocks level. However, we report in Eq. (3b) the upper layers of that reduction in order to illustrate rule $[W_{next}]$.

$$[B_{first}]\ \frac{[B_{one}]\ \dfrac{\mathtt{w1} :: \mathtt{w2} = \overline{w} \cap \{\mathtt{w1}, \mathtt{w2}\} \qquad [W_{first}]\ \dfrac{\textit{Eq. (3a)}}{f, \mathtt{w1} :: \mathtt{w2}, s, \overline{i}, C_0 \triangleright \mathtt{w1}}}{f, C_0, (\overline{w}, s, \overline{i}) \to \mathtt{w1}} \qquad \mathtt{w1} \neq \bot}{f, C_0, (\underbrace{\mathtt{w1} :: \mathtt{w2}}_{\overline{w}}, \underbrace{\mathtt{best\_first}}_{s}, \underbrace{\mathtt{capacity\_used}\ 80\%}_{\overline{i}}) :: \varepsilon \to \mathtt{w1}} \quad (2)$$

$$[W_{first}]\ \frac{\textbf{strategy}(\mathtt{w1} :: \mathtt{w2}, \mathtt{best\_first}) = \mathtt{w1} \qquad \mathtt{w1} \neq \bot \qquad \textbf{valid}(f, \mathtt{w1}, \overline{i}, C_0)}{f, \mathtt{w1} :: \mathtt{w2}, \mathtt{best\_first}, \underbrace{\mathtt{capacity\_used}\ 80\%}_{\overline{i}}, C_0 \triangleright \mathtt{w1}} \quad (3a)$$

$$[W_{next}]\ \frac{\textbf{strategy}(\overline{w}, s) = \mathtt{w1} \qquad \mathtt{w1} \neq \bot \qquad \neg\textbf{valid}(f, \mathtt{w1}, \overline{i}, C_1) \qquad [W_{first}]\ \dfrac{\vdots}{f, \overline{w} \setminus \mathtt{w1}, s, \overline{i}, C_1 \triangleright \mathtt{w2}}}{f, \underbrace{\mathtt{w1} :: \mathtt{w2}}_{\overline{w}}, \underbrace{\mathtt{best\_first}}_{s}, \underbrace{\mathtt{capacity\_used}\ 80\%}_{\overline{i}}, C_1 \triangleright \mathtt{w2}} \quad (3b)$$

Workers Rules. The last stratum of the APP LTS semantics are the Workers rules, which evaluate strategies s and invalidation policies $\overline{i}$ to find (if any) a valid worker to allocate the function. The logic of this layer is to exhaustively try to find a valid worker among the ones listed (in a block). Specifically, we either find the first worker **valid** (following the strategy of the block) and use that one ($[W_{first}]$) or we go through the remaining workers in the list ($[W_{next}]$), either returning the first **valid** worker or reporting failure otherwise ($[W_{end}]$).

Completing our examples, Eq. (3a) ends the derivation from Eq. (1a) and Eq. (2) via rule $[W_{first}]$. There, the **strategy** relation (left-most premise) interprets the `strategy` of the block and finds a worker (`w1`) while the **valid** predicate (right-most premise) checks if the worker can support the allocation of the function—e.g., considering the list of `invalidate` options of the block.

Eq. (3b) closes the derivation on C_1 and shows the logic of workers selection with invalid workers. Since `w1` cannot host f ($\neg$**valid**$(\cdots)$), in the right-most premise, we remove it from the candidate workers and find `w2` with rule $[W_{first}]$.

4 Related Work

To the best of our knowledge, this is the first presentation of a formal model to reason on the semantics of serverless function scheduling.

Looking at the literature on serverless, we can spot proposals that might adopt similar formal models to encode and complement desirable properties of function scheduling, otherwise implemented via ad-hoc, platform-wide policies. For instance, some works present serverless architectures that enable the efficient composition of functions co-located on the same host [3,49,55]. Here, APP can help to make these policies explicit and parametric, expressed in terms of strategies and invalidation policies. Another example regards security, e.g., Datta et al. [14] present a serverless platform where developers can constrain the information flow among functions to avoid attacks due to container reuse and data exfiltration. In

this case, APP can complement flow policies with constraints that restrict the co-tenancy of functions and their flows of communication.

Previous proposals tackled the definition of formal models that capture the semantics of function execution—while we focus on the semantics of function scheduling. One such model is the Serverless Kernel Calculus [23, 27], where the authors propose a core formal programming model for serverless computing that combines concepts from both the λ- and the π-calculus and study its relation with a model of the underlying processes that execute the functions. Another proposal by Jangda et al. [34], also drawing inspiration from the λ-calculus and process algebras, focusses on the execution details of serverless platforms, e.g., providing primitives to express cold/warm starts, persistence, and transactions. We see these models as inspiration for future work, e.g., where one can pair the execution and the scheduling layers and reason on their interactions.

Finally, future work regards extending APP to capture existing serverless function scheduling policies [2, 9, 12, 35, 37–39, 51, 53, 56–58], to demonstrate that APP is a bridging technology that lets users choose the best-fitting policies given their function profiles. The version of APP discussed in the present paper includes the `any` and `best_first` worker selection policies and the `capacity_used` and `max_concurrent_invocations` invalidation conditions, but both the language and its implementation have been designed to be easily extended with other selection policies or invalidation conditions. For instance, we have already implemented the extensions tAPP [18] and aAPP [16] respectively supporting policies based on topological information or (anti-)affinity among functions.

5 Conclusion and Future Work

We started covering the ground for a rigorous treatment of FaaS scheduling. More precisely, we have defined a formal semantics of APP, a platform-agnostic language for FaaS scheduling policies. We define the operational semantics as a labelled transition system on system configurations describing the functions that are allocated for execution on each worker in the system.

One can use this operational semantics to perform an analysis of properties related to the allocation of functions on workers. For instance, as advocated in [16], we expect to usages of our semantics for reachability analysis (e.g., detecting whether the scheduler can allocate a critical function on an untrusted worker). In general, one can define the properties of the FaaS systems by using temporal logics and then check their validity by using model-checking techniques.

Another application of our APP semantics is defining observational equivalences or pre-orders by exploiting techniques inspired, e.g., by bisimulation [41] or testing [44] semantics. This line of work could be useful to study refinement hierarchies among APP script, according to which a script is a refinement of another one when it guarantees (at least) the same set of observable properties.

Directions for future work include capturing configurations whose workers can change, i.e., when workers (dis)appear while the platform is running. In such dynamic scenarios, the configuration space becomes infinite, which could have

repercussions on the complexity/decidability of the analysis techniques. We also plan to define a timed or probabilistic semantics for APP, e.g., to specify the expected execution time of functions on workers, or in order to model random allocation policies which tries to balance the workload distribution. One such model could support quantitative analysis, e.g., to estimate the completion time of a FaaS application or the distribution of workers' load over time, i.e., it would allow us to quantitatively reason on policies, e.g., whether they could lead to bad performance or underutilisation of workers.

Another direction for future work comes from the neighbouring field of microservices—the state-of-the-art style for cloud architectures [20]. In microservices, state-of-the-art proposals for scheduling include *affinity* properties that drive the allocation of microservices. Affinity can be positive, to indicate the preference/need to deploy a microservice on a machine where other, affine microservices run. This case covers contexts where the microservices can increase their performance by sharing pools of resources, libraries/runtime environments, and efficient communication channels within the host machine. Negative affinity is too an interesting property; when we specify that a microservice is anti-affine with others, we mean that the scheduler cannot deploy that microservice on a machine where any of the anti-affine ones run. The negative case can help in cases where, e.g., for security reasons, we prevent the deployment of microservices from different trust tiers, as well as avoiding contention due to the co-location of microservices that require the same pool of resources.

Proposals in this direction to use as inspiration are by Baarzi and Kesidis [7], who present a framework for the deployment of microservices that infers and assigns affinity and anti-affinity traits to microservices to orient the distribution of resources and microservices replicas on the available machines; Sampaio et al. [50], who introduce an adaptation mechanism for microservice deployment based on microservice affinities (e.g., the more messages microservices exchange the more affine they are) and resource usage; Sheoran et al. [54], who propose an approach that computes procedural affinity of communication among microservices to make placement decisions. Moreover, looking at the industry, Azure Service Fabric [6] provides a notion of *service affinity* that ensures the placement of replicas of a service on the same nodes as those of another, affine service. Another example is Kubernetes, which has a notion of *node affinity* and *inter-pod (anti-)affinity* to express advanced scheduling logic for the optimal distribution of pods [40]. Overall, the mentioned work proves the usefulness of affinity-aware deployments at lower layers than FaaS (e.g., VMs, containers, microservices).

As a final line of future work, we mention our interest in using APP as function scheduling language also in other serverless platforms, besides the already considered OpenWhisk. In particular, we have recently developed FunLess [17], a new platform specifcally tailored for private edge-cloud systems, and we are currently implementing and validating an extension of FunLess with APP.

References

1. Knative. https://knative.dev/ (2023)

2. Abad, C.L., Boza, E.F., Eyk, E.V.: Package-aware scheduling of faas functions. In: Proc. of ACM/SPEC ICPE. pp. 101–106. ACM (2018). https://doi.org/10.1145/3185768.3186294
3. Akkus, I.E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., Hilt, V.: {SAND}: Towards {High-Performance} serverless computing. In: 2018 Usenix Annual Technical Conference (USENIX ATC 18). pp. 923–935 (2018)
4. Alpernas, K., Flanagan, C., Fouladi, S., Ryzhyk, L., Sagiv, M., Schmitz, T., Winstein, K.: Secure serverless computing using dynamic information flow control. Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–26 (2018)
5. Azure, M.: Microsoft azure functions. https://azure.microsoft.com/ (11 2022)
6. Azure, M.: Microsoft azure functions. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-overview (11 2022)
7. Baarzi, A.F., Kesidis, G.: Showar: Right-sizing and efficient scheduling of microservices. In: Proceedings of the ACM Symposium on Cloud Computing. pp. 427–441 (2021)
8. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al.: Serverless computing: Current trends and open problems. In: Research advances in cloud computing, pp. 1–20. Springer (2017)
9. Baresi, L., Quattrocchi, G.: Paps: A serverless platform for edge computing infrastructures. Frontiers in Sustainable Cities **3**, 690660 (2021)
10. Boreale, M., Bruni, R., Caires, L., Nicola, R.D., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: SCC: A service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4184, pp. 38–57. Springer (2006). https://doi.org/10.1007/11841197_3
11. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Ciancarini, P., Wiklicky, H. (eds.) Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Bologna, Italy, June 14-16, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4038, pp. 63–81. Springer (2006). https://doi.org/10.1007/11767954_5
12. Casale, G., Artač, M., Van Den Heuvel, W.J., van Hoorn, A., Jakovits, P., Leymann, F., Long, M., Papanikolaou, V., Presenza, D., Russo, A., et al.: Radon: rational decomposition and orchestration for serverless computing. SICS Software-Intensive Cyber-Physical Systems **35**(1), 77–87 (2020)
13. Cloud, G.: Google cloud functions. https://cloud.google.com/functions/ (11 2022)
14. Datta, P., Kumar, P., Morris, T., Grace, M., Rahmati, A., Bates, A.: Valve: Securing function workflows on serverless computing platforms. In: Proceedings of The Web Conference 2020. pp. 939–950 (2020)
15. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: A declarative approach to topology-aware serverless function-execution scheduling. In: IEEE International Conference on Web Services, ICWS 2022, Barcelona, Spain, July 10-16, 2022. pp. 337–342. IEEE (2022). https://doi.org/10.1109/ICWS55610.2022.00056
16. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Formally verifying function scheduling properties in serverless applications. IT Prof. **25**(6), 94–99 (2023). https://doi.org/10.1109/MITP.2023.3333071
17. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Funless: Functions-as-a-service for private edge cloud systems. CoRR **abs/2405.21009** (2024). https://doi.org/10.48550/ARXIV.2405.21009

18. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: An openwhisk extension for topology-aware allocation priority policies. In: Castellani, I., Tiezzi, F. (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, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14676, pp. 201–218. Springer (2024). https://doi.org/10.1007/978-3-031-62697-5_11
19. De Palma, G., Giallorenzo, S., Mauro, J., Zavattaro, G.: Allocation priority policies for serverless function-execution scheduling optimisation. In: Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14-17, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12571, pp. 416–430. Springer (2020). https://doi.org/10.1007/978-3-030-65310-1_29
20. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering, pp. 195–216. Springer (2017). https://doi.org/10.1007/978-3-319-67425-4_12
21. Fission: Fission. https://fission.io/ (11 2022)
22. Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J., Montesi, F.: Self-reconfiguring microservices. In: Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday. pp. 194–210. Springer (2016). https://doi.org/10.1007/978-3-319-30734-3_14
23. Gabbrielli, M., Giallorenzo, S., Lanese, I., Montesi, F., Peressotti, M., Zingaro, S.P.: No More, No Less - A Formal Model for Serverless Computing. In: Nielson, H.R., Tuosto, E. (eds.) COORDINATION 2019 - 21th International Conference on Coordination Languages and Models. Coordination Models and Languages, vol. LNCS-11533, pp. 148–157. Springer International Publishing, Kongens Lyngby, Denmark (Jun 2019). https://doi.org/10.1007/978-3-030-22397-7_9, part 3: Exploring New Frontiers
24. Gabbrielli, M., Giallorenzo, S., Lanese, I., Zingaro, S.P.: A language-based approach for interoperability of iot platforms. In: 51st Hawaii International Conference on System Sciences, HICSS 2018, Hilton Waikoloa Village, Hawaii, USA, January 3-6, 2018. pp. 1–10. ScholarSpace / AIS Electronic Library (AISeL) (2018)
25. Gabbrielli, M., Giallorenzo, S., Montesi, F.: Service-oriented architectures: From design to production exploiting workflow patterns. In: Distributed Computing and Artificial Intelligence, 11th International Conference, DCAI 2014, Salamanca, Spain, June 4-6, 2014. pp. 131–139. Springer (2014). https://doi.org/10.1007/978-3-319-07593-8_17
26. Gabbrielli, M., Martini, S., Giallorenzo, S.: Programming Languages: Principles and Paradigms, Second Edition. Undergraduate Topics in Computer Science, Springer (2023). https://doi.org/10.1007/978-3-031-34144-1
27. Giallorenzo, S., Lanese, I., Montesi, F., Sangiorgi, D., Zingaro, S.P.: The Servers of Serverless Computing: A Formal Revisitation of Functions as a Service. In: de Boer, F.S., Mauro, J. (eds.) Recent Developments in the Design and Implementation of Programming Languages. OpenAccess Series in Informatics (OASIcs), vol. 86, pp. 5:1–5:21. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/OASIcs.Gabbrielli.5
28. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F.: Lemma2jolie: A tool to generate microservice apis from domain models. Sci. Comput. Program. **228**, 102956 (2023). https://doi.org/10.1016/j.scico.2023.102956

29. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S.: Jolie and LEMMA: model-driven engineering and programming languages meet on microservices. In: Damiani, F., Dardha, O. (eds.) Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12717, pp. 276–284. Springer (2021). https://doi.org/10.1007/978-3-030-78142-2_17
30. Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Unwerawattana, N.: Jot: A jolie framework for testing microservices. In: Jongmans, S., Lopes, A. (eds.) Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13908, pp. 172–191. Springer (2023). https://doi.org/10.1007/978-3-031-35361-1_10
31. Giallorenzo, S., Montesi, F., Safina, L., Zingaro, S.P.: Ephemeral data handling in microservices with tquery. PeerJ Comput. Sci. **8**, e1037 (2022). https://doi.org/10.7717/peerj-cs.1037
32. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: A calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4294, pp. 327–338. Springer (2006). https://doi.org/10.1007/11948148_27
33. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with openlambda. In: 8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16) (2016)
34. Jangda, A., Pinckney, D., Brun, Y., Guha, A.: Formal foundations of serverless computing. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–26 (2019)
35. Jia, Z., Witchel, E.: Boki: Stateful serverless computing with shared logs. In: Proc. of ACM SIGOPS SOSP. pp. 691–707. ACM, New York, NY, USA (2021). https://doi.org/10.1145/3477132.3483541
36. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.C., Khandelwal, A., Pu, Q., Shankar, V., Menezes Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud programming simplified: A berkeley view on serverless computing. Tech. Rep. UCB/EECS-2019-3, EECS Department, University of California, Berkeley (02 2019)
37. Kehrer, S., Scheffold, J., Blochinger, W.: Serverless skeletons for elastic parallel processing. In: 2019 IEEE 5th International Conference on Big Data Intelligence and Computing (DATACOM). IEEE. pp. 185–192 (2019)
38. Kelly, D., Glavin, F., Barrett, E.: Serverless computing: Behind the scenes of major platforms. In: 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). pp. 304–312. IEEE (2020)
39. Kotni, S., Nayak, A., Ganapathy, V., Basu, A.: Faastlane: Accelerating function-as-a-service workflows. In: Proc. of USENIX ATC. pp. 805–820. USENIX Association (2021)
40. Kubernetes: Assign pods to nodes using node affinity. https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes-using-node-affinity/ (11 2022)
41. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)

42. Montesi, F., Guidi, C., Zavattaro, G.: Composing services with JOLIE. In: Fifth IEEE European Conference on Web Services (ECOWS 2007), 26-28 November 2007, Halle (Saale), Germany. pp. 13–22. IEEE Computer Society (2007). https://doi.org/10.1109/ECOWS.2007.19
43. Nicola, R.D., Ferrari, G., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng. **24**(5), 315–330 (1998). https://doi.org/10.1109/32.685256
44. Nicola, R.D., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. **34**, 83–133 (1984). https://doi.org/10.1016/0304-3975(84)90113-0
45. Nicola, R.D., Loreti, M., Pugliese, R., Tiezzi, F.: A formal approach to autonomic systems programming: The SCEL language. ACM Trans. Auton. Adapt. Syst. **9**(2), 7:1–7:29 (2014). https://doi.org/10.1145/2619998
46. OpenFaaS: faasd. https://github.com/openfaas/faasd/
47. OpenFaaS: Openfaas. https://www.openfaas.com/ (11 2022)
48. OpenWhisk, A.: Apache openwhisk. https://openwhisk.apache.org/ (11 2022)
49. Sabbioni, A., Rosa, L., Bujari, A., Foschini, L., Corradi, A.: A shared memory approach for function chaining in serverless platforms. In: 2021 IEEE Symposium on Computers and Communications (ISCC). pp. 1–6. IEEE (2021)
50. Sampaio, A.R., Rubin, J., Beschastnikh, I., Rosa, N.S.: Improving microservice-based applications with runtime placement adaptation. Journal of Internet Services and Applications **10**(1), 1–30 (2019)
51. Sampé, J., Sánchez-Artigas, M., García-López, P., París, G.: Data-driven serverless functions for object storage. In: Middleware. pp. 121–133. Middleware '17, ACM (2017). https://doi.org/10.1145/3135974.3135980
52. Services, A.W.: Introducing aws lambda. https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/ (11 2022)
53. Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., Bianchini, R.: Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In: Proc. of USENIX ATC. pp. 205–218 (2020)
54. Sheoran, A., Fahmy, S., Sharma, P., Modi, N.: Invenio: Communication affinity computation for low-latency microservices. In: Proceedings of the Symposium on Architectures for Networking and Communications Systems. pp. 88–101 (2021)
55. Shillaker, S., Pietzuch, P.: Faasm: Lightweight isolation for efficient stateful serverless computing. In: Proc. of USENIX ATC. pp. 419–433. USENIX Association (2020)
56. Silva, P., Fireman, D., Pereira, T.E.: Prebaking functions to warm the serverless cold start. In: Proc. of Middleware. pp. 1–13. Middleware '20, ACM, New York, NY, USA (2020). https://doi.org/10.1145/3423211.3425682
57. Smith, C.P., Jindal, A., Chadha, M., Gerndt, M., Benedict, S.: Fado: Faas functions and data orchestrator for multiple serverless edge-cloud clusters. In: 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC). pp. 17–25. IEEE (2022)
58. Sreekanti, V., Wu, C., Lin, X.C., Schleier-Smith, J., Gonzalez, J.E., Hellerstein, J.M., Tumanov, A.: Cloudburst: Stateful functions-as-a-service. Proc. VLDB Endow. **13**(12), 2438–2452 (Jul 2020). https://doi.org/10.14778/3407790.3407836
59. Wang, L., Li, M., Zhang, Y., Ristenpart, T., Swift, M.: Peeking behind the curtains of serverless platforms. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 133–146 (2018)
60. YAML: YAML specification. https://yaml.org/spec/ (11 2022)