Leveraging static analysis for cost-aware serverless scheduling policies

Giuseppe de Palma, Saverio Giallorenzo, Cosimo Laneve, Jacopo Mauro, Matteo Trentin, Gianluigi Zavattaro

▶ **To cite this version:**

Giuseppe de Palma, Saverio Giallorenzo, Cosimo Laneve, Jacopo Mauro, Matteo Trentin, et al.. Leveraging static analysis for cost-aware serverless scheduling policies. International Journal on Software Tools for Technology Transfer, 2025, 10.1007/s10009-024-00776-9 . hal-04887650

HAL Id: hal-04887650
https://hal.science/hal-04887650v1

Submitted on 15 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Leveraging Static Analysis for Cost-aware Serverless Scheduling Policies

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

Received: date / Accepted: date

Abstract Mainstream serverless platforms follow opinionated, hardcoded scheduling policies to allocate functions on the available workers. Such policies may decrease the performance of the application due to locality issues (e.g., functions executed on workers far from the data they use). APP is a platform-agnostic declarative language that mitigates these problems by allowing serverless platforms to support multiple, per-function, scheduling logics. However, defining the "right" scheduling policy in APP is far from trivial, often requiring rounds of refinement involving knowledge of the underlying infrastructure, guesswork, and empirical testing.

We propose a framework that lightens the burden on the shoulders of users by deriving cost information from the functions, via static analysis, into a cost-aware variant of APP that we call cAPP. We present a prototype of such framework, where we extract cost equations from functions' code, synthesise cost expressions through off-the-shelf solvers, and implement cAPP to support the specification and execution of cost-aware allocation policies.

Keywords Scheduling · Serverless · Cost Equations

Research partly supported by the SERICS project (PE00000014) under the MUR National Recovery and Resilience Plan (PNRR) funded by the European Union - NextGenerationEU, the research project FREEDA (CUP: I53D23003550006) funded by the framework PRIN 2022 (MUR, Italy), the French ANR project SmartCloud ANR-23-CE25-0012, and project PNRR CN HPC - SPOKE 9 - Innovation Grant LEONARDO - TASI - RTMER funded by the NextGenerationEU European initiative through the MUR, Italy (CUP: J33C22001170001).

[1] Università di Bologna, Italy E-mail: giuseppe.depalma2@unibo.it, saverio.giallorenzo2@unibo.it, cosimo.laneve@unibo.it, matteo.trentin2@unibo.it, gianluigi.zavattaro@unibo.it · [2] INRIA, France · [3] University of Southern Denmark E-mail: mauro@imada.sdu.dk

1 Introduction

Serverless, specifically Function-as-a-Service (FaaS), is a cloud-based service that lets users build applications as compositions of stateless functions, delegating all system administration tasks to the platform. Serverless has two main advantages for users: it saves them time by handling resource allocation, maintenance, and scaling, and it reduces costs by charging only for the resources used to perform work, i.e., users do not pay for running idle servers [22]. Several managed serverless offerings are available from popular cloud providers like Amazon AWS Lambda, Google Cloud Functions, and Microsoft Azure Functions, as well as open-source alternatives such as OpenWhisk, OpenFaaS, OpenLambda, and Fission. In all cases, the platform manages the allocation of function executions across the available computing resources, usually called *workers*, following opinionated platform-wide policies. However, a function can endure performance degradation depending on the worker that hosts it, e.g., due to effects like the latency to access data relative to the worker's location, called *data locality* [20].

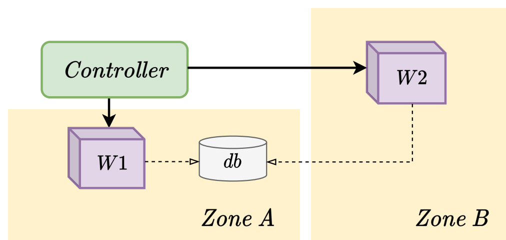
We visualise the issue by commenting on the minimal scenario drawn in Figure 1. There, we have two workers, `W1` and `W2`, located in distinct geographical *Zones A* and *B*, respectively. Both workers can run functions that interact with a database (*db*) located in *Zone A*. When the function scheduler — the *Controller* — receives a request to execute a function, it must determine which worker to use. To minimise the function run time (and, thus, the response time), the scheduler should take into account the different computational capabilities of the workers, as well as their current workloads. Moreover, when functions interact with external services, it might take into account also their latency to access them, choosing the ones that minimise it. In our example, the scheduler should find a worker that minimises the time to access the database. From Figure 1, that worker is `W1`, thanks

to its closeness to *db* (same geographic zone) which allows it to undergo lower latencies than the farther worker W2.

The APP *Language.* APP [13, 16] is a declarative language recently introduced to support the *configuration of custom function-execution scheduling policies*. The APP snippet in Figure 1 codifies the (data) locality principle of the example. In the platform, we associate the functions that access *db* with a tag, called db_query. Then, we include the scheduling rule in the snippet to specify that every function tagged db_query can run on either W1 or W2, and the strategy to follow when choosing between them is best_first, i.e., select the first worker in top-down order of appearance (hence giving priority to worker W1 if available and not overloaded).

By featuring customised function scheduling policies, APP allows one to disentangle functions from platform-specific allocation rules. However, this freedom manifests the problem of specifying the appropriate scheduling for the functions (e.g., minimise latency). Currently, APP users determine the best policy for their functions by selecting one of the strategies (e.g., the mentioned best_first) *manually* when they write the companion APP script to their functions, based on their intuitions and insights on the latter's behaviour (e.g., data access). For instance, a user can write the APP script in Figure 1 if they had knowledge about the reduced latency of worker W1 in accessing *db*. In other words, the user must know about the workers' topology and their latencies w.r.t. the external services used by their functions. However, users might not have such knowledge when writing their APP scripts. Moreover, the worker-service latency is a property that can dynamically change depending, e.g., on the state of the network connections, including traffic and congestion.

Our contribution. We propose to overcome the above limitations by letting users express latency-aware selection strategies. For instance, in the scenario in Figure 1, we expect the user to be able to express policies like the following one:




```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
    strategy: best_first
```

Fig. 1: A multi-zone serverless topology and APP script.

```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
    strategy: min_latency
```

where the strategy min_latency instructs the platform to give priority to the worker expected to endure the lowest latency w.r.t. its latency in the usage of external services (e.g., the database *db* in Figure 1).

While such high-level policies greatly alleviate the burden on users, they open a relevant question: *given a function* f *to be scheduled and a list of possible workers, how can one automatically guide the scheduling of* f *on a worker with low-latency access to* f*'s external services?*

We answer to the above question by proposing a solution consisting of three components:

1. the quantification of (an upper bound of) of the invocations done by a function to its external services, obtained through a static analysis of the function's code;
2. the periodical run-time monitoring of the latencies workers endure in contacting said external services;
3. the computation, at function scheduling time, of an upper-bound of the function-worker overall latency by combining the quantified invocations to the function's external services with the workers' expected latencies.

In other terms, we propose to use a combination of static analysis (applied on a function's code) and run-time monitoring (of the workers latencies in accessing the external services) to estimate a *cost* for executing a function on a worker, considering what and how it uses external services.

Thanks to such a quantification, we can support other meaningful scheduling policies like the following one:

```
- db_query:
  - workers:
    - wrk: W1
    - wrk: W2
  invalidate: max_latency: 300
```

In this case, we do not specify a selection strategy (using the platform's default one) to choose between the two workers, but we consider invalid any worker whose estimated latency of running the function exceeds the threshold of 300ms.

We discuss the applicability of our approach on a minimal language, called miniSL (standing for mini Serverless Language), for programming functions in serverless applications. We focus on a minimal language for two main reasons. First, it allows us to show the feasibility of our approach by concentrating on basic language constructs, abstracting away from the specific (and, in some case, idiosyncratic) constructs of the different programming languages used in serverless computing. Second, miniSL represents an abstract language for describing the behaviour of programs written in mainstream

programming languages, so that the theory developed in this article becomes directly applicable to any programming language.[1] Concretely, we define a static analysis technique that, given miniSL code, extracts a set of equations that define meaningful costs, in particular, the number and kind of external service invocations. Then, we feed the equations to off-the-shelf cost analysers (e.g., PUBS [3] and CoFloCo [18]) to compute cost expressions that quantify over-approximations of said costs.

The question we ask above focusses on a theoretical problem, i.e., how we can give an abstract estimation of the expected latencies of external service invocations done by a function scheduled on a given worker. In this article, we also address how one can concretely use our theoretical proposal, by *defining a serverless platform architecture that supports the framework and a grounding principle of the abstract estimations w.r.t. the performance of the workers*—so we can use it to select the best workers for each function under scheduling.

The serverless platform we implement supports:

1. the deployment of functions written in the miniSL language, whereupon we compute their cost equations using the technique described above;
2. the specification of scheduling policies via a dedicated scheduling policy language called cAPP, obtained by extending APP with cost-aware policies, like the `min_latency` and `max_latency` discussed above;
3. the periodical monitoring of each worker's latencies in accessing the external services possibly invoked by the deployed functions;
4. the usage of a cost equation solver when functions are scheduled to quantify the expected number of invocations to external services, so that worker selection follows the specified cost-aware policy.

We achieve such implementation by extending FunLess [15], a recent serverless platform developed for private edge cloud systems. FunLess deployments encompass heterogeneous and geographically distributed nodes, where the latencies for accessing external services could differ among workers that provide different computing and networking resources. For this reason, we expect that cost-aware scheduling policies could have a major impact on such serverless systems.

Structure of the article. We start, in Section 2, by introducing background information on serverless. Then, in Section 3, we define our minimal language, called miniSL, which includes constructs for specifying computation flow (via `if` and `for` constructs) and for service invocation (via a `call` construct).

[1] Since serverless platforms support many disparate programming languages, we see exploring the usage of miniSL as an abstract language for describing serverless functions too broad and tangential to be tackled in this article, and leave it as interesting future work.

Then, in Section 4, we describe how to exploit static analysis techniques, inspired by behavioural type systems like those by Garcia et al. and Laneve and Sacerdoti Coen [19, 28], to automatically extract a set of equations from function source codes written in miniSL that define meaningful function costs (in our case, the number of invocation to external services). One can feed these equations to off-the-shelf cost analyser (e.g., PUBS [3] or CoFloCo [18]) to compute cost expressions quantifying over-approximations of the considered costs. In Section 5, we present cAPP, our extension of APP for expressing cost-aware scheduling policies. Moving to implementation, we introduce, in Section 6, a proof of concept of the proposed framework, obtained by extending the capabilities of the serverless platform FunLess [15]. We conclude by positioning this work in Section 7 and by drawing final takeaways and future work in Section 8.

This article revises and extends previous work [12]. The most relevant novelties are the implementation of the FunLess-based serverless framework supporting cost-aware scheduling policies expressed in cAPP (Section 6), preliminaries on serverless computing (Section 2), and positioning (Section 7).

2 Preliminaries: Serverless Computing

We briefly overview serverless computing and platforms.

Modern cloud applications have access to a plethora of services that allow them to scale and be more resilient. However, also complexity and costs grow with scale, leading to the need for efficient, automatic management. Serverless computing responds to these needs by offering a service that abstracts away the underlying infrastructure and allows developers to build applications as compositions of stateless, event-driven functions that automatically scale according to user requests.

The functions that make up a serverless application run in short-lived environments, triggered by different kinds of events. These events include HTTP requests, database changes, file uploads, and scheduled timeouts. At triggering time, the provider runs the function after having initialised a dedicated execution environment; a secure and isolated context that manages all the resources needed by the function lifecycle. Depending on the implementation, these execution environments encompass Virtual Machines (VMs), containers, and dedicated interpreters/runtimes [49].

Architecture-wise, the main components that make up a typical serverless platform are the controllers and the workers, as illustrated in Figure 2.

Requests to the platform are performed by *events* that come from external sources, such as users or other systems. A variety of events are usually supported, ranging from HTTP requests for handling webhooks and web-based interactions, cloud storage events (e.g., creation, deletion, and modification of an object in the cloud storage system or database

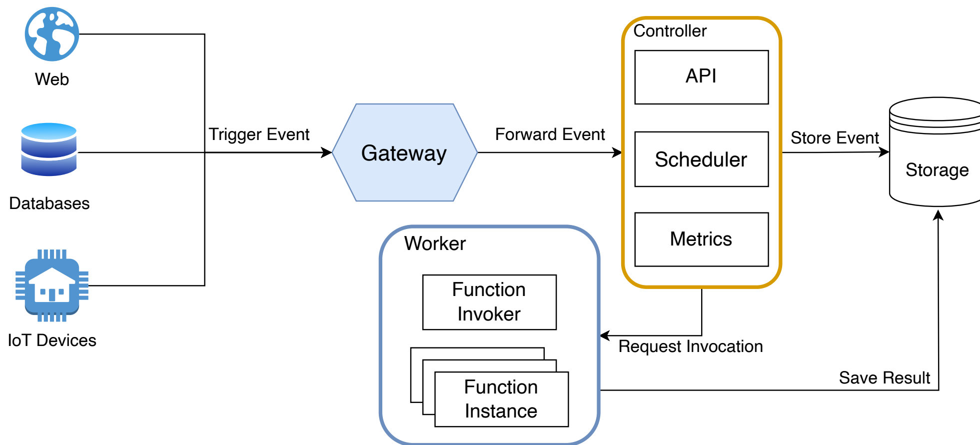


Fig. 2: Typical serverless platform architecture.

activities such as insertions, updates, or deletions of records), events triggered at predefined intervals or specific times, by messages arriving in a message queue or streams, and by custom sources or external systems via APIs.

The controller receives the requests and handles scaling decisions based on inbound traffic and system load, orchestrating the allocation of workers for function execution and managing the overall system coordination and monitoring. In particular, a component of the controller — the *scheduler* — determines which worker should execute a given function, based on factors such as current load, function requirements, and resource availability. Workers execute the functions as per controller's request, handling the execution environment lifecycle, including provisioning, scaling, and teardown.

Serverless platforms use sophisticated scheduling strategies to optimise function execution across different workers. One of these strategies regards mitigating the phenomenon of "cold start", i.e., the downtime due to waiting for the initialisation of the function's runtime environment, e.g., avoided by anticipating the initialisation of/keeping a "warm" dedicated containers/VM to reduce the function run time. To prevent bottlenecks, serverless platforms implement load balancing strategies that can range from simple round-robin methods to more complex algorithms that consider the current load, historical data, and predicted demand [49].

Serverless platforms usually adopt a layer that supports communication among controller and workers, handling messages and data transfer between components. In particular, message queues or event brokers (e.g., RabbitMQ [9], Kafka [4]) implement asynchronous communication between components, allowing decoupling and scalability. Internal APIs facilitate synchronous communication for tasks such as function deployment, status updates, and resource allocation. Monitoring tools are also used to collect metrics on resource usage, function execution times, and error rates. Metrics provide visibility into system performance, function execution, and overall health and enable debugging, troubleshooting, and performance optimisation.

Among the leading providers of serverless computing platforms, Amazon Web Services (AWS) Lambda [35] stands out as a pioneer in the field. AWS Lambda was the first publicly available serverless platform, allowing developers to pay only for the compute time consumed by their functions. Other platforms followed suit, offering similar capabilities, such as Microsoft Azure Cloud Functions [5] and Google Cloud Platform (GCP) Cloud Functions [11]. A number of open-source serverless platforms have also emerged, such as OpenWhisk [33], Knative [1], and OpenFaaS [32]. These platforms can be deployed on-premises or on the cloud, and offer a more flexible and customizable solution compared to the proprietary platforms. Among these, FunLess [15] has been recently proposed for mixed edge-cloud environments, using WebAssembly [47] (Wasm) to run functions with the aim of reducing memory and CPU footprint (thanks to the lightweight nature of Wasm) and mitigating cold-start issues (thanks to Wasm's fast startup times and efficient caching).

3 The mini Serverless Language

The mini Serverless Language, shortened into miniSL, is a minimal calculus that we propose in this article to specify the functions' behaviour in serverless computing. miniSL focuses only on core constructs to define operations to access services, conditional behaviour with simple guards, and iterations.

Function executions are triggered by events. At triggering time, a function receives a sequence of invocation parameters: for this reason, we assume a countable set of *parameter names*, ranged over by p, p'. We also consider a countable set of *counters*, ranged over by i, j, used as indexes in iteration statements. Integer numbers are represented by n; service

names are represented by h, g, $\cdots$. The syntax of miniSL is as follows (we use over-lines to denote sequences, e.g., p_1, p_2 could be an instance of $\overline{p}$):

$$
\begin{array}{lcl}
\mathsf{F} & ::= & (\overline{p})\ \texttt{=>}\ \{\ \mathsf{S}\ \} \\
\mathsf{S} & ::= & \varepsilon \mid \texttt{call}\ \mathtt{h}(\overline{\mathsf{E}})\ \mathsf{S} \mid \texttt{if}\ (\mathsf{G})\ \{\ \mathsf{S}\ \}\ \texttt{else}\ \{\ \mathsf{S}\ \} \mid \\
 & & \texttt{for}\ (i\ \texttt{in}\ \texttt{range}(\texttt{0},\mathsf{E}))\{\ \mathsf{S}\ \} \\
\mathsf{G} & ::= & \mathsf{E} \mid \texttt{call}\ \mathtt{h}(\overline{\mathsf{E}}) \\
\mathsf{E} & ::= & n \mid i \mid p \mid \mathsf{E} \sharp \mathsf{E} \mid !\mathsf{E} \\
\sharp & ::= & + \mid - \mid * \mid / \mid > \mid < \mid \\
 & & >= \mid == \mid <= \mid \&\& \mid ||
\end{array}
$$

A *function* F associates to a sequence of parameters $\overline{p}$ a statement S executed at every occurrence of the triggering event. *Statements* include the empty statement ε (which is always omitted when the statement is not empty); calls to external services by means of the `call` keyword; the conditional and iteration statements. The guard of a conditional statement could be either a boolean expression or a call to an external service which, in this case, is expected to return a boolean value. The language supports standard expressions in which it is possible to use integer numbers and counters. Notice that, in our simple language, the iteration statement considers an iteration variable ranging from 0 to the value of an expression E evaluated when the first iteration starts.

In the rest of the article, we assume all programs to be well-formed so that all names are correctly used (e.g., counters are declared before they are used). For each expression used in the range of an iteration construct, we assume that its evaluation generates an integer, and for each service invocation `call` $\mathtt{h}(\overline{E})$, we assume that h is a correct service name and $\overline{E}$ is a sequence of expressions generating correct values to be passed to that service. Calls to services include serverless invocations, which possibly execute on a different worker of the caller.

We illustrate miniSL by means of three examples. As a first example, consider the code in Listing 1 representing the call of a function that selects a functionality based on the characteristic of the invoker.

```
1 ( isPremiumUser , par ) => {
2   if( isPremiumUser ) {
3     call PremiumService( par )
4   } else {
5     call BasicService( par )
6   }
7 }
```

Listing 1: Function with a conditional statement guarded by an expression.

This code may invoke either a `PremiumService` or a `BasicService` depending on whether it has been triggered by a premium user or not. The parameter `isPremiumUser` is a value indicating whether the user is a premium member (when the value is true) or not (when the value is false). The other invocation parameter `par` must be forwarded to the invoked service. For the purposes of this article, this example is relevant because if we want to reduce the latency of this function, the best node to schedule it could be the one that reduces the latency of the invocation of either the service `PremiumService` or the service `BasicService`, depending on whether `isPremiumUser` is true or false, respectively.

Consider now the following function, where differently from the previous version, it is necessary to call an external service to decide whether we are serving a premium or a basic user.

```
1 ( username , par ) => {
2   if( call IsPremiumUser(username) ) {
3     call PremiumService( par )
4   } else {
5     call BasicService( par )
6   }
7 }
```

Listing 2: Function with a conditional statement guarded by an invocation to external service.

In this case, the first parameter carries an attribute of the user (its name) but it does not indicate (with a boolean value) whether it is a premium user or not. Instead, the necessary boolean value is returned by the external service `IsPremiumUser` that checks the username and returns true only if that username corresponds to that of a premium user. Within this setting is difficult to predict the best worker to execute such a function, because the branch that will be selected is not known at function scheduling time. If the user triggering the event is a premium member, the expected execution time of the function is the sum of the latencies of the service invocations of `IsPremiumUser` and `PremiumService` while, if the user is not a premium member, the expected execution time is the sum of the latencies of the services `IsPremiumUser` and `BasicService`. As an (over-)approximation of the expected delay, we could consider the worst execution time, i.e., the sum of the latency of the service `IsPremiumUser` plus the maximum between the latencies of the services `PremiumService` and `BasicService`. At scheduling time, we could select the best worker as the one giving the best guarantees in the worst case, e.g., the one with the best over-approximation.

Consider now a function triggering a sequence of map-reduce jobs.

```
1 ( jobs , m, r ) => {
2   for(i in range(0, m)) {
3     call Map(jobs , i)
4     for(j in range(0, r)) {
5       call Reduce(jobs , i, j)
6     }
7   }
8 }
```

Listing 3: Function implementing a map-reduce logic.

The parameter `jobs` describes a sequence of map-reduce jobs. The number of jobs is indicated by the parameter `m`. The

"map" phase, which generates `m` "reduce" subtasks, is implemented by an external service `Map` that receives the `jobs` and the specific index `i` of the job to be mapped. The "reduce" subtasks are implemented by an external service `Reduce` that receives the `jobs`, the specific index `i` of the job under execution, and the specific index `j` of the "reduce" subtask to be executed — for every `i`, there are `r` such subtasks. In this case, the expected latency of the entire function is given by the sum of `m` times the latency of the service `Map` and of `m` × `r` times the latency of the service `Reduce`. Given that such latency could be high, a user could be interested to run the function on a worker, only if the expected overall latency is below a given threshold.

4 The Inference of Cost Expressions

In this section, we formalise the inference of a cost program from miniSL code. Once inferred, we can feed this program to off-the-shelf tools, such as [3, 18], to calculate the cost expression of the related miniSL code. Notice that, since these tools are designed to handle only Presburger arithmetic, we restrict our extraction only to a subset of miniSL, where the expressions conform to Presburger arithmetic constraints. Cost programs are lists of *equations* which are terms

$$f(\bar{x}) \ = \ \mathbb{e} + \sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}) \qquad [\ \varphi\ ]$$

where variables occurring in the right-hand side and in φ are a subset of $\bar{x}$ and f and f_i are (cost) function symbols. Every function definition has a right-hand side consisting of

- a *Presburger arithmetic expression* $\mathbb{e}$ whose syntax is

$$\mathbb{e} ::= \ x \ | \ q \ | \ \mathbb{e}+\mathbb{e} \ | \ \mathbb{e}-\mathbb{e} \ | \ q*\mathbb{e} \ | \ max(\mathbb{e}_1, \cdots, \mathbb{e}_k)$$

  where x is a variable and q is a positive rational number,
- a number of *cost function invocations* $f_i(\overline{\mathbb{e}_i})$ where $\overline{\mathbb{e}_i}$ are Presburger arithmetic expressions,
- the *Presburger guard* φ is a *linear conjunctive constraint*, *i.e.*, a conjunction of *constraints* of the form $\mathbb{e}_1 \geq \mathbb{e}_2$ or $\mathbb{e}_1 = \mathbb{e}_2$, where both $\mathbb{e}_1$ and $\mathbb{e}_2$ are Presburger arithmetic expressions.

The intended meaning of an equation

$$f(\bar{x}) \ = \ \mathbb{e} + \sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}) \ [\ \varphi\ ]$$

is that the cost of f is given by $\mathbb{e}$ and the costs of $f_i(\overline{\mathbb{e}_i})$, when the guard φ is true. Intuitively, $\mathbb{e}$ quantifies the specific cost of one execution of f without taking into account invocations of either auxiliary functions or recursive calls. Such additional cost is quantified by $\sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i})$. The *solution of a cost program* is an expression, quantifying the cost of the function symbol in the first equation in the list, which is parametric in the formal parameters of the function symbol.

For example, the following cost program

$$\begin{array}{ll} f(N,M) = M + f(N-1,M) & [N \geq 1] \\ f(N,M) = 0 & [N = 0] \end{array}$$

defines a function f that is invoked $N+1$ times and each invocation, excluding the last having cost 0, costs M. The solution of this cost program is the *cost expression* $N \times M$.

Our technique associates cost programs to miniSL functions following a syntax-directed approach: we define a set of (inference) rules that, following the parse tree bottom-up, gather fragments of cost programs that are then combined in a syntax-directed manner. As usual with syntax-directed rules, we use *environments* Γ, Γ', which are maps. In particular,

- Γ takes a service `h` or a parameter name p and returns a Presburger arithmetics expression, which is usually a variable. For example, if $\Gamma(\mathtt{h}) = X$, then X will appear in the cost expressions of miniSL functions using `h` and will represent the cost for accessing the service. As regards parameter names p, $\Gamma(p)$ represents values which are known at function scheduling time,
- Γ takes counters i and returns the type `Int`.

When we write $\Gamma + i : \mathtt{Int}$, we assume that i does not belong to the domain of Γ. Let C be a sum of (cost) function invocations and let Q be a list of equations. Judgments have the shape

- $\Gamma \vdash \mathsf{E} : \mathbb{e}$, meaning that the value of the *integer expression* E in Γ is represented by (the Presburger arithmetic expression) $\mathbb{e}$,
- $\Gamma \vdash \mathsf{E} : \varphi$, meaning that the value of the *boolean expression* E in Γ is represented by (the Presburger guard) φ,
- $\Gamma \vdash \mathsf{S} : \mathbb{e} \ ; \ \mathsf{C} \ ; \ \mathsf{Q}$, meaning that the cost of S in the environment Γ is $\mathbb{e} + \mathsf{C}$ given a list Q of equations,
- $\Gamma \vdash \mathsf{F} : \mathsf{Q}$, meaning that the cost of a miniSL function F in the environment Γ is given by the cost program Q (remember that a cost program is a list of equations).

We use the notation $var(\mathbb{e})$ to address the set of variables occurring in $\mathbb{e}$, which is extended to tuples $var(\mathbb{e}_1, \cdots, \mathbb{e}_n)$ with the standard meaning. Similarly $var(\sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}))$ is the union of the sets of variables $var(\overline{\mathbb{e}_0}), \cdots, var(\overline{\mathbb{e}_n})$. We use $var(\varphi)$ for Presburger guards.

The inference rules for miniSL are reported in Figure 3. They compute the cost of a program with respect to the calls to external services (whose cost is recorded in the environment Γ). Therefore, if a miniSL expression (or statement) has no service invocation, its cost is 0. Notice that in the rule [IF-EXP] we use the guard $[\ \neg\varphi\ ]$, to model the negation of a linear conjunctive constraint φ, even if negation is not permitted in Presburger arithmetic. Actually, such notation is syntactic sugar defined as follows:

$$\text{[EPS]}\quad \Gamma \vdash \varepsilon : 0 \;;\; \emptyset \;;\; \emptyset \qquad\qquad \text{[CALL]}\quad \frac{\Gamma(\mathtt{h}) = \mathbb{e} \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q}}{\Gamma \vdash \mathtt{call}\ \mathtt{h}(\overline{\mathsf{E}})\ \ \mathsf{S} : \mathbb{e} + \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q}}$$

$$\text{[IF-EXP]}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{E} : \varphi \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q} \qquad \Gamma \vdash \mathsf{S}' : \mathbb{e}'' \;;\; \mathsf{C}' \;;\; \mathsf{Q}' \qquad \mathit{if}_\ell \ \mathit{fresh} \\ \overline{w} = \mathit{var}(\varphi, \mathbb{e}', \mathbb{e}'') \cup \mathit{var}(\mathsf{C}, \mathsf{C}') \qquad \mathsf{Q}'' = \begin{bmatrix} \mathit{if}_\ell(\overline{w}) = \mathbb{e}' + \mathsf{C} & [\ \varphi\ ] \\ \mathit{if}_\ell(\overline{w}) = \mathbb{e}'' + \mathsf{C}' & [\neg\varphi] \end{bmatrix}\end{array}}{\Gamma \vdash \mathtt{if}\ (\mathsf{E})\ \ \{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}'\ \} : 0 \;;\; \mathit{if}_\ell(\overline{w}) \;;\; \mathsf{Q},\ \mathsf{Q}', \mathsf{Q}''}$$

$$\text{[IF-CALL]}\quad \frac{\Gamma(\mathtt{h}) = \mathbb{e} \qquad \Gamma \vdash \mathsf{S} : \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q} \qquad \Gamma \vdash \mathsf{S}' : \mathbb{e}'' \;;\; \mathsf{C}' \;;\; \mathsf{Q}'}{\Gamma \vdash \mathtt{if}\ (\mathtt{call}\ \mathtt{h}(\overline{\mathsf{E}}))\ \ \{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}'\ \} : \mathbb{e} + \mathit{max}(\mathbb{e}', \mathbb{e}'') \;;\; \mathsf{C} + \mathsf{C}' \;;\; \mathsf{Q},\ \mathsf{Q}'}$$

$$\text{[FOR]}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{E} : \mathbb{e} \qquad \Gamma + i : \mathtt{Int} \vdash \mathsf{S} : \mathbb{e}' \;;\; \mathsf{C} \;;\; \mathsf{Q} \qquad \overline{w} = (\mathit{var}(\mathbb{e}, \mathbb{e}') \cup \mathit{var}(\mathsf{C})) \setminus i \\ \mathit{for}_\ell \ \mathit{fresh} \qquad \mathsf{Q}' = \begin{bmatrix} \mathit{for}_\ell(i, \overline{w}) = \mathbb{e}' + \mathsf{C} + \mathit{for}_\ell(i+1, \overline{w}) & [\ \mathbb{e}\ \geq\ i\ ] \\ \mathit{for}_\ell(i,\ \overline{w}) = 0 & [\ i\ \geq\ \mathbb{e} + 1\ ] \end{bmatrix}\end{array}}{\Gamma \vdash \mathtt{for}\ (i\ \mathtt{in}\ \mathtt{range}(\mathtt{0}, \mathsf{E}))\{\ \mathsf{S}\ \} : 0 \;;\; \mathit{for}_\ell(0,\ \overline{w}) \;;\; \mathsf{Q},\ \mathsf{Q}'}$$

$$\text{[PRG]}\quad \frac{\begin{array}{c}\Gamma \vdash \mathsf{S} : \mathbb{e} \;;\; \mathsf{C} \;;\; \mathsf{Q} \qquad \overline{w} = \mathit{var}(\overline{p}, \mathbb{e}) \cup \mathit{var}(\mathsf{C}) \\ \mathit{main}\ \mathit{fresh} \qquad \mathsf{Q}' = \mathit{main}(\overline{w}) = \mathbb{e} + \mathsf{C} \quad [\,] \end{array}}{\Gamma \vdash (\overline{p})\ \ \texttt{=>}\ \ \{\ \mathsf{S}\ \} : \mathsf{Q}',\ \mathsf{Q}}$$

Fig. 3: The rules for deriving cost expressions

– let $\neg\varphi$ (the *negation* of a Presburger guard φ) be the *list* of Presburger guards

$$\begin{aligned} \neg(\mathbb{e} \geq \mathbb{e}') &= \mathbb{e}' \geq \mathbb{e} + 1 \\ \neg(\mathbb{e} = \mathbb{e}') &= \mathbb{e} \geq \mathbb{e}' + 1 \;;\; \mathbb{e}' \geq \mathbb{e} + 1 \\ \neg(\mathbb{e} \wedge \mathbb{e}') &= \neg\mathbb{e} \;;\; \neg\mathbb{e}' \end{aligned}$$

where ; is the list concatenation operator (the list represents a *disjunction of Presburger guards*),

– let $\neg\varphi = \varphi_1 \;;\; \cdots \;;\; \varphi_m$, where φ_i are Presburger guards, then

$$\Big(f(\overline{x}) \;=\; \mathbb{e} + \textstyle\sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i})\Big)\ [\neg\varphi] \;\stackrel{\text{def}}{=}\; \Big\{ f(\overline{x}) \;=\; \mathbb{e} + \textstyle\sum_{i \in 0..n} f_i(\overline{\mathbb{e}_i}) \quad [\varphi_j] \quad \big| \quad j \in 1..m \Big\}\,.$$

We now comment on the inference rules reported in Figure 3.[2]

Rule [CALL] manages invocation of services: the cost of `call` $\mathtt{h}(E)$ S is the cost of S plus the cost for accessing the service h.

Rule [IF-EXP] defines the cost of conditionals when the guard is a Presburger arithmetic expression that can be evaluated at function scheduling time. We use a corresponding cost function, if_ℓ, whose name is fresh,[3] to indicate that the cost of the entire conditional statement is either the cost of the then-branch or the else-branch, depending on whether the guard is true or false. As discussed above, the use of the guard $\neg\varphi$ generates a list of equations.

Rule [IF-CALL] defines an upper bound of the cost of conditionals when the guard is an invocation to a service. At scheduling time it is not possible to determine whether the guard is true or false – *c.f.* the second example in Section 3. Therefore the cost of a conditional is the maximum between the cost $\mathbb{e}' + \mathsf{C}$ of the then-branch and the one $\mathbb{e}'' + \mathsf{C}'$ of the else-branch, plus the cost $\mathbb{e}$ to access to the service in the guard. However, considering that the expression $\mathit{max}(\mathbb{e} + \mathsf{C}, \mathbb{e}' + \mathsf{C}')$ is not a valid right-hand side for the equations in our cost programs, we take as over-approximation the expression $\mathit{max}(\mathbb{e}, \mathbb{e}') + \mathsf{C} + \mathsf{C}'$.

As regards iterations, according to [FOR], its cost is the invocation of the corresponding function, for_ℓ, whose name is fresh (we assume that iterations have pairwise different line-codes). The rule adds the counter i to Γ (please recall that $\Gamma + i : \mathtt{Int}$ entails that $i \notin \mathit{dom}(\Gamma)$). In particular, the counter i is the first formal parameter of for_ℓ; the other parameters are all the variables in $\mathbb{e}$, in notation $\mathit{var}(\mathbb{e})$ plus those in

[2] We omit rules for expressions E since they are straightforward: they simply return E if E is in Presburger arithmetics. We notice that no rule is defined if *E is not* in Presburger arithmetics. In fact, in these cases, it is not possible to derive cost equations.

[3] We assume that conditionals have pairwise different line-codes and ℓ represents the line-code of the if in the source code.

the invocations C (minus the i). There are two equations for every iteration: one is the case when i is out-of-range, hence the cost is 0, the other is when it is in range and the cost is the one of the body *plus* the cost of the recursive invocation of for_ℓ with i increased by 1.

The cost of a miniSL program is defined by [PRG]. This rule defines an equation for the function *main* and puts this equation as the first one in the list of equations [4]. Once inferred, we can feed this program to off-the-shelf tools, such as [3,18], which will compute the cost of the the *first function* of the list, *i.e.* the *main* function.

As an example, we apply the rules of Figure 3 to the codes in Listings 1, 2, and 3. Let $\Gamma(\texttt{isPremiumUser}) = u$, $\Gamma(\texttt{par}) = v$, $\Gamma(\texttt{PremiumService}) = P$ and $\Gamma(\texttt{BasicService}) = B$. For Listing 1, we obtain the cost program

$$
\begin{array}{rll}
main(u,v,P,B) = & if_2(u,P,B) & [\,] \\
if_2(u,P,B) = & P & [\,u=1\,] \\
if_2(u,P,B) = & B & [\,u=0\,]
\end{array}
$$

Notice that the parameters of the *main* function include, initially, the values corresponding to the parameters of the corresponding miniSL function and then those corresponding to the other variables occurring in the cost equations.
For Listing 2, let $\Gamma(\texttt{username}) = u$, $\Gamma(\texttt{par}) = v$, $\Gamma(\texttt{IsPremiumUser}) = K$, $\Gamma(\texttt{PremiumService}) = P$ and $\Gamma(\texttt{BasicService}) = B$. Then the rules of Figure 3 return the single equation

$$
main(u,v,K,P,B) = K + max(P,B) \qquad [\,]
$$

For 3, when $\Gamma(\texttt{jobs}) = J$. $\Gamma(\texttt{m}) = m$, $\Gamma(\texttt{r}) = r$, $\Gamma(\texttt{Map}) = M$ and $\Gamma(\texttt{Reduce}) = R$, the cost program is

$$
\begin{array}{rll}
main(J,m,r,M,R) = & for_2(0,m,r,M,R) & [\,] \\
for_2(i,m,r,M,R) = & M + for_4(0,r,R) + & \\
& for_2(i+1,m,r,M,R) & [\,m \geq i\,] \\
for_2(i,m,r,M,R) = & 0 & [\,i \geq m+1\,] \\
for_4(j,r,R) = & R + for_4(j+1,r,R) & [\,r \geq j\,] \\
for_4(j,r,R) = & 0 & [\,j \geq r+1\,]
\end{array}
$$

The foregoing cost programs can be fed to automatic solvers such as PUBS [3] and CoFloCo [18]. The evaluation of the cost program for Listing 1 returns $max(P,B)$ because u is unknown. On the contrary, if u is known, it is possible to obtain a more precise evaluation from the solver: if $u = 1$ it is possible to ask the solver to consider $main(1,v,P,B)$ and the solution will be P, while if $u = 0$ it is possible to ask the solver to consider $main(0,v,P,B)$ and the solution will be B. The evaluation of $main(u,v,K,P,B)$ for Listing 2 gives the expression $K + max(P,B)$, which is exactly what is written in the equation. This is reasonable because, statically, we are not aware of the value returned by the invocation of IsPremiumService. Last, the evaluation of the cost program for Listing 3 returns the expression $m \times (M + r \times R)$.

Since we combine miniSL and our inference system for estimating costs of functions interacting with external services, one might wonder how relevant the approach is, i.e., how common are serverless functions that call external services, and what is their structure? While a systematic study is out of the scope of this article, we started this process by analysing a comprehensive repository of illustrative serverless functions[5] for different platforms (AWS, Azure, OpenWhisk, etc.). Our analysis reveals that 50% (65/130) of these functions follow patterns that one can represent using miniSL by abstracting away structured data and internal computation and estimate their cost w.r.t. the flow of external calls, such as HTTP invocations to external services.

5 From APP to cAPP

We now present the new language cAPP for expressing cost-aware function scheduling policies, by extending the already available language APP. We start by briefly introducing the APP syntax and constructs, reported in Figure 4, as found in its first incarnation by De Palma et al. [16] and then discussing the new constructs we introduce to handle cost-aware scheduling policies.

5.1 The APP Language

APP scripts are collections of tagged scheduling policies. The main, mandatory component of any policy (identified by a *policy_tag*) are the `workers` therein, i.e., a collection of labels that identify on which workers the scheduler can allocate the functions. The assumption is that the environment running the APP script establishes a 1-to-1 association so that each worker has a unique, identifying label. A *policy* associates to every function a list of one or more *block*s, each including

- the `worker` clause stating on which workers the function can be scheduled;
- the `strategy`, an optional parameter that defines the scheduling followed to select one of the workers of the block;
- the `invalidate` condition, optional as well, which determines when a worker cannot host a function.

When a selected worker is invalid, the scheduler tries to apply the selection strategy and allocate the function on the rest of the available workers in the block. If none of the workers of a block is available, the scheduling moves to the next block. The last clause, `followup`, encompasses a whole policy and

[4] Given that miniSL functions are anonymous, we use the default name *main* for the corresponding cost function.

[5] "A collection of ready-to-deploy Serverless Framework services" at `https://github.com/serverless/examples`.

```
policy_tag ∈  Identifiers ∪ {default}   worker_label ∈ Identifiers
n          ∈  ℕ
app        ::= tag‾
tag        ::= policy_tag : ‾- block followup?‾
block      ::= workers: [ * | ‾- wrk: worker_label‾ ]
               (strategy: [ random | platform | best_first
                          | min_latency ])?
               (invalidate: [ capacity_used : n%
                            | max_concurrent_invocations: n
                            | overload
                            | max_latency: n
                            ])?
followup   ::= followup: [ default | fail ]
```

Fig. 4: The APP syntax and, in red, the cAPP extension.

defines what to do when no *block*s of the policy managed to allocate the function. When set to `fail`, the scheduling of the function fails; when set to `default`, the scheduling continues by following the (special) `default` policy.

The `strategy` parameter supports the following values: `platform` that applies the default selection strategy of the serverless platform; `random` that allocates functions stochastically among the workers of the block following a uniform distribution; `best-first` that allocates functions on workers based on their top-down order of appearance in the block. The options for the `invalidate` parameter are: `overload` that invalidates a worker based on the default invalidation control of the platform; `capacity_used` that invalidates a worker if it uses more than a given percentage threshold of memory; `max_concurrent_invocations` that invalidates a worker if a given number of function invocations are already currently executed on the worker.

We close this section by extending the example presented in Figure 1 to illustrate APP, reported below.

```
db_query:
  - workers:
    - wrk: W1
    - wrk: W2
    strategy: best_first
    invalidate: capacity_used: 50%
  followup: fail
```

Recalling the example, we consider some functions that need to access a database. To reduce latency (as per data locality principle), we want to run those functions on the workers within the same zone of the database (W1). If that option is not valid, then we run the functions on workers located further away (W2).

In the code, at the first line, we define the *policy tag*, which is db_query. The functions accessing the database have the same tag (not shown in the example) so we link them to this policy. Then, the keyword `workers` indicates a list of *worker_labels*, which identify the worker in the proximity of the database, W1, and the farther one, W2. Finally, we define three parameters: the `strategy` used by the scheduler to choose among the listed worker labels, the policy that `invalidates` the selection of a worker label, and the `followup` policy in case all workers are `invalidated`. In the example, given the `best_first` `strategy`, we first prefer W1 and then W2, and we `invalidate` the scheduling on each of them if the worker corresponding to the chosen label has `capacity_used` at more than `50%`. Since there are no subsequent blocks, in case all workers of the blocks are invalidated, we proceed with the `followup` instruction, which specifies to `fail` the request for function execution.

The interested reader can find more examples and tutorials on APP in publications by De Palma et al. [13, 14, 17].

5.2 Cost-aware policies with cAPP

To support the scheduling of functions based on costs we propose two extensions to APP. The first one is a new selection strategy named `min_latency`. Such a strategy selects, among some available workers, the one which minimises a given cost expression. The second one is a new invalidation condition named `max_latency`. This condition invalidates a worker in case the corresponding cost expression is greater than a given threshold.

We dub cAPP the cost-aware extension of APP and illustrate its main features by showing examples of cAPP scripts that target the functions in Listings 1–3.

```
- premUser:
 - workers:
     - wrk: W1
     - wrk: W2
   strategy: min_latency
```

Listing 4: cAPP script for Listings 1 and 2.

Listing 4 defines a cAPP tagged `premUser` that we will associate to both the functions at Listing 1 and 2. In this script, we specify to follow the logic `min_latency` to select among the two workers, W1 and W2 listed in the `workers` clause, and prioritises the one for which the solution of the cost expression is minimal.

To better illustrate the phases of the `min_latency` strategy, we depict in Figure 5 the flow, from the deployment of the cAPP script to the scheduling of the functions in Listings 1 and 2. When the cAPP script is created, the association between the functions code and their cAPP script is specified by tagging the two functions with the comment `// tag:premUser`. In this phase, assuming the scheduling policy of the cAPP script requires the computation of the functions cost (because the strategy is `min_latency`), the

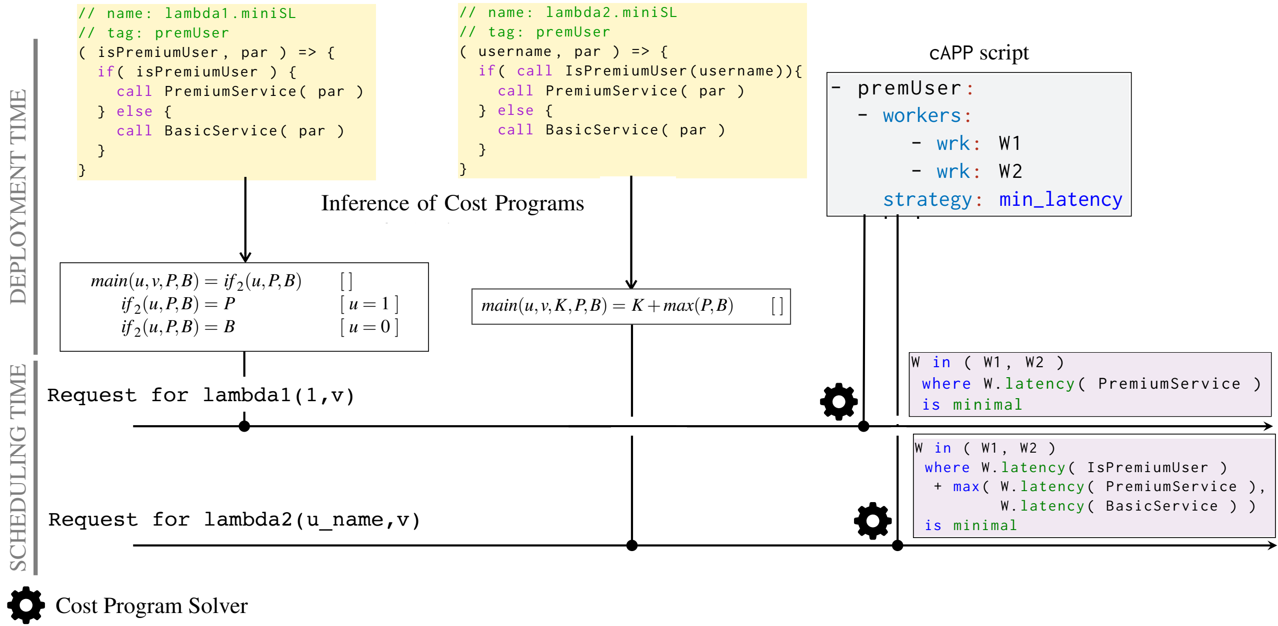


Fig. 5: Flow followed, from deployment to scheduling, of the functions at Listings 1 and 2.

code of the functions is used to infer the corresponding cost program. When the functions are invoked, i.e., at scheduling time, we can compute the solution of the cost program, given the knowledge of the invocation parameters. The knowledge of the invocation parameters allows for a more precise analysis. For instance, for the function in Listings 1, called `lambda1`, it is possible to invoke the cost analyser with either $main(1,v,P,B)$ or $main(0,v,P,B)$ where P represent the cost of `PremiumService`, B the cost of `BasicService` and the first parameter is the value of the `isPremiumUser` parameter.

If the invocation is `lambda1`$(1,v)$ (first horizontal line in In Figure 5) then the cost program (represented by the intersection point on the left) and the corresponding cAPP policy to implement the expected scheduling policy are retrieved. At this point, a cost analyser is used to solve the cost programs (depicted by the gear). In this case, since the cost expression is P, which is `PremiumService`, the scheduling amounts to (*i*) estimating the latencies to access to `PremiumService` from the considered workers and (*ii*) choosing the worker that minimises the foregoing latency. This computation is highlighted in the rightmost grey window corresponding to the request `lambda1`$(1,v)$.

When the request is `lambda2`(u_name,v), the corresponding cost function is $main(u_name,v,K,P,B)$, where K is the cost of the service `IsPremiumUser`. In this case, the cost expression is $K+max(P,B)$ Since `lambda2.miniSL` has the same tag as `lambda1.miniSL`, the selected cAPP script is the same. Therefore the scheduling amounts to minimize the latencies from the workers `W1` and `W2` to the services `IsPremiumUser`, `PremiumService` and `BasicService` according to the expression $K+max(P,B)$. This is highlighted in the rightmost grey window corresponding to the request `lambda2`(u_name,v).

The controller needs also to be aware of the possibility of invalidating a worker when the latency to access a service exceeds a certain threshold. In particular, when `max_latency` is used in the `invalidate` clause, workers are not selected if the computed latency is above the given value. To illustrate this item, let us consider the cAPP code for the map-reduce function in Listing 5.

```
- mapReduce :
  - workers:
    - wrk: W1
    - wrk: W2
  strategy: random
  invalidate:
    max_latency: 300
```

Listing 5: cAPP script for Listing 3.

As visualised in Figure 6, starting from the (top-most) deployment phase box where we tag the function (`//tag: mapReduce`), the cost program is computed, obtaining the associated cost expression. Then, when a request for the function is received, the execution of the cAPP policy is triggered, which selects one of the two workers `W1` or `W2` at `random` and checks their validity following the logic shown at the bottom of Figure 6, i.e., the cost program is solved and the parameters `m` and `r` are replaced with the `latency` to contact the `Map` and `Reduce` services from the selected worker, and

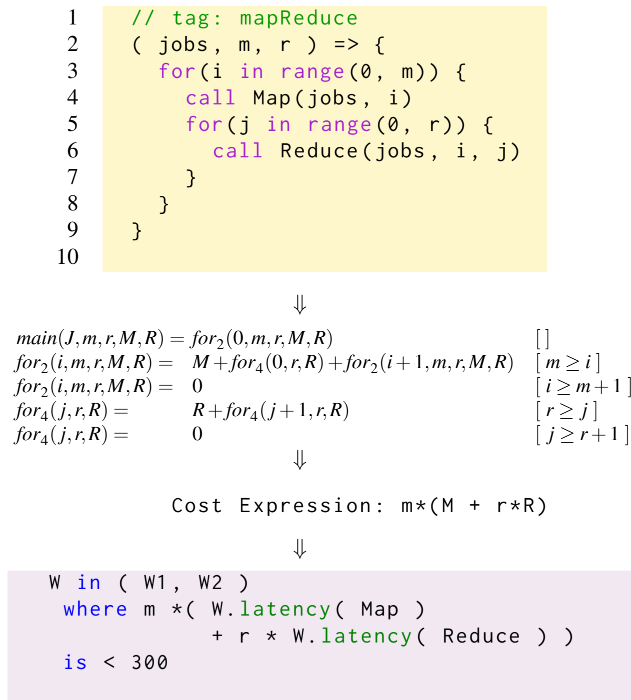


Fig. 6: The map-reduce function, its cost analysis, and scheduling invalidation logic.

possibly invalidate it if the computed value is greater than `300`.

6 Implementation

We now describe the implementation of a prototype serverless framework that allows to use cAPP to express cost-aware function scheduling policies.

6.1 The FunLess Platform

To develop the prototype, we rely on FunLess [15], i.e., a FaaS platform designed for mixed edge-cloud environments, using WebAssembly [47] (Wasm) to run functions. This approach offers several advantages: enhanced security through Wasm's inherent isolation mechanisms, reduced memory and CPU footprint by eliminating the need for container runtimes and orchestrators, and mitigated cold-start issues thanks to Wasm's fast startup times and efficient caching. Moreover, FunLess ensures a consistent function development and deployment environment across diverse hardware and software architectures, making it adaptable to various edge-cloud scenarios and providing flexible deployment options, either through existing containerization solutions or simpler setups, leveraging Wasm's portability and lightweight nature.

FunLess is composed of two kinds of services built with Elixir and Rust (the Core and the Workers), on top of the BEAM virtual machine, a Database (Postgres), and a monitoring system (Prometheus). The platform's architecture is shown in Figure 7, with the yellow highlighted components being the ones we have added or modified to support cAPP.

Core. The central management component of FunLess is the Core. It exposes an HTTP REST API for users to interact with the platform and handle the lifecycle of functions — creation, storage, scheduling, and invocation. When a function is uploaded to the platform, it is stored in the *Postgres* database and broadcasted to the available Workers, which will cache it locally to reduce cold-start times during invocation. The Core is also responsible for scheduling function executions. It uses real-time metrics collected by *Prometheus* to select the Worker with the highest amount of available memory. This results in a balanced workload distribution in case of workers with similar resources. Communication between the two components leverages the BEAM's lightweight distributed messaging system.

Workers. The workers are the components responsible for executing functions as directed by the Core. Workers use the Wasmtime [46] runtime, a WebAssembly engine that supports the WebAssembly System Interface (WASI) [48]. Each Worker caches function binaries locally upon receiving them from the Core. When a function is invoked, it first checks its cache for the required binary: if the binary is present, it is loaded and executed immediately; if not, the Worker requests the binary from the Core, which sends it back for execution. Each Worker's maximum cache size is configurable, and when the cache exceeds its limit, the least recently used functions are evicted. Workers are designed to abstract away the specifics of the Wasm runtime, allowing for future flexibility in supporting different or multiple runtimes. This design ensures that functions can be executed across different hardware architectures, making FunLess versatile for various deployment environments, from cloud servers to low-power edge devices.

6.2 Extending FunLess to support cAPP

To correctly handle cAPP-based scheduling policies in FunLess, several additions had to be made to the platform, both in terms of deployment and implementation.

Firstly, we implemented a miniSL-to-Wasm compiler, to produce binaries that would be compatible with FunLess' Workers. As discussed in the previous section, the cost-analysis that we perform on miniSL functions considers the invocations that such functions perform on external services. Moreover, we expect to monitor, at run time the latencies of the worker-external service invocations. To this

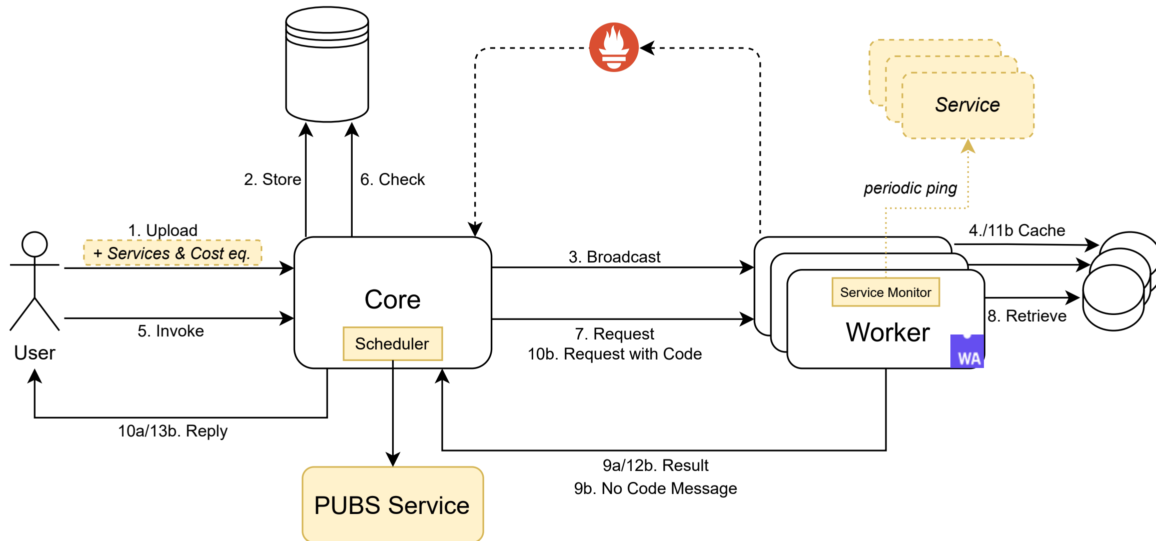


Fig. 7: Schema of the FunLess architecture, extended to support cAPP.

aim, we extended FunLess' internal data structures to include information for each function, e.g., the URL and HTTP verb associated to the external services the function can invoke. This metadata is then to be used by the Workers to actually perform service calls to monitor, at run time, the invocation latencies. By default, each Worker sends a HEAD request to all services every 10 seconds and caches the response time. The latencies for all services are exposed as metrics by each Worker, allowing Prometheus to collect them along with standard information (e.g., memory usage).

We then extended the FunLess scheduler to allow scheduling decision based on cAPP.

For modularity purposes, we did not bind the implementation to rely on a specific cost analyser but we allow instead the administrator to choose the one that best fits the platform's needs. For show the feasibility of our approach, we used a containerised version of PUBS [3] and invoke it using simple API requests. This allows the Core to contact this service to calculate the correct upper bound for each function's cost equations and estimate the latencies for all the available Workers.

The extension to FunLess was written in Elixir (as the rest of the platform) and required around 1k lines of code.[6]

6.3 Implemented Case Studies

We have performed a qualitative evaluation of our cAPP-based extension of FunLess by verifying that the expected Workers (i.e., the ones we simulate having the lowest latency accessing the services used by the given function) are being targeted during scheduling.[7]

We deployed our platform on a local Kubernetes configuration — we use kind[8], which is a tool for running local Kubernetes clusters — using two Worker nodes and one Core node. We have tested the scenario depicted in Figure 5 by implementing *PremiumService*, *BasicService* and *IsPremiumUser*. These services are configured to simulate different latency towards different Workers by delaying their response depending the host performing the HTTP requests. We performed 100 function invocations for each use case and noted the amount of times the "correct" Worker (i.e. the one with the lowest predicted latency, using the `min_latency` strategy) was targeted. We then did the same using FunLess' default scheduling policy, and compared the results.

Additionally, we tested the behaviour of the `max_latency` invalidate option using the map-reduce use case from Figure 6. Also in this case, we performed 100 invocations and noted the amount of times the "incorrect" worker (i.e., the one with the excessive latency) was selected, and then compared them with the default scheduling policy. We also extended this test case by changing the latencies after 50 invocations, so that the lowest-latency worker would not stay the same during the entire test.

Summing up, this gave us four separate test cases to compare the behaviour of cAPP-based scheduling with that of FunLess' default scheduler:

[6] https://github.com/funlessdev/funless/tree/miniSL

[7] A quantitative performance evaluation is left as a future work and outside the scope of this work since it would also require the adoption of a fully-fledged programming language to use FaaS benchmarks.

[8] https://kind.sigs.k8s.io/

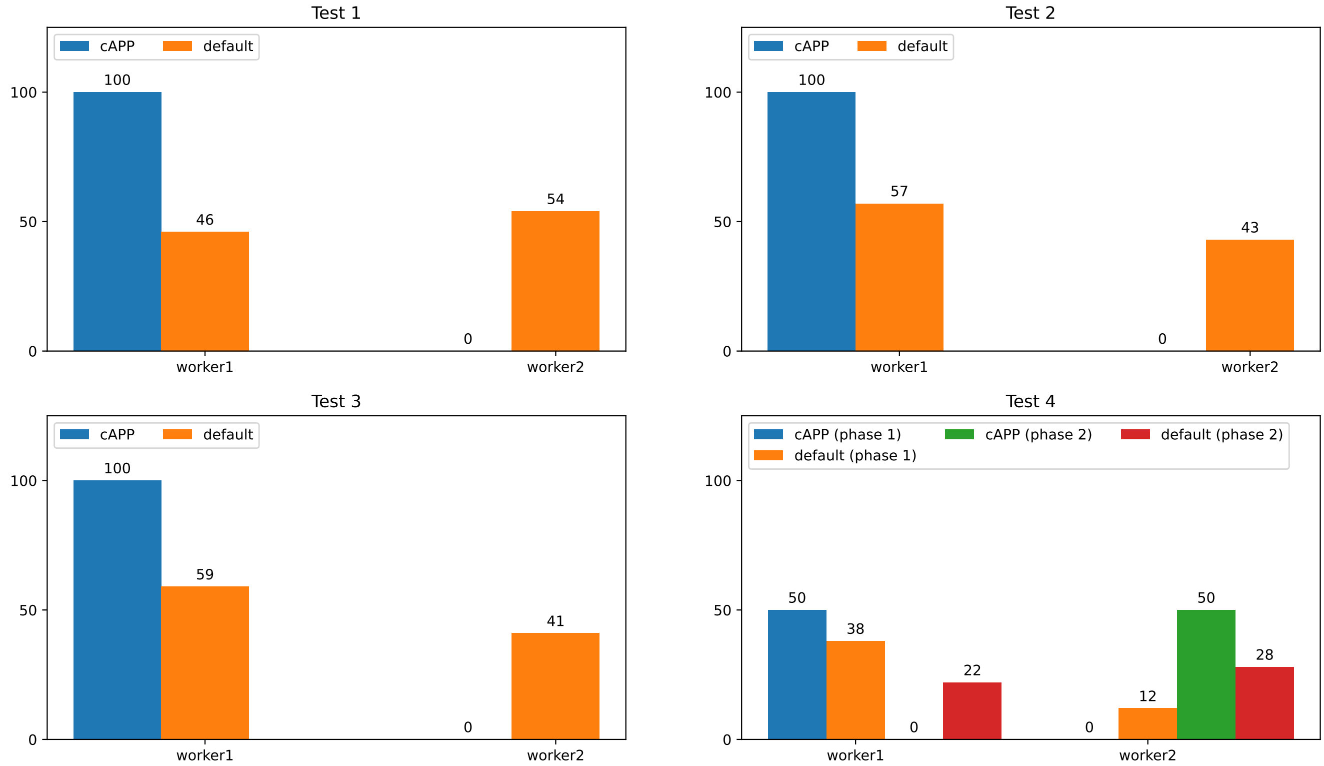


Fig. 8: Experimental results for each test case. Number of invocations is shown on the y-axis.

1. *PremiumService*/*BasicService*, where *isPremiumUser* is a boolean parameter.
2. *PremiumService*/*BasicService*, where *IsPremiumUser* is a service to be called.
3. *Map*/*Reduce*, where latencies are unchanged during all invocations.
4. *Map*/*Reduce*, where latencies are flipped after 50 invocations and having a break of 30 seconds between the two phases to allow Prometheus to receive updated latency information from the Workers.

In each of these test cases, *worker1* had a latency of 300ms towards all services, while *worker2* had a latency of 700ms. This was flipped in the last 50 invocations of Test 4.

The cAPP script used in Test 1 and Test 2 are the ones in Figure 5 and 3 with the only change that the maximum latency was set to 2000ms (otherwise 300ms would have invalidated both workers).

The results are shown in Figure 8. It can be seen, in all experiments, how cAPP-based scheduling always chooses the "correct" worker, while the default scheduling policy tends to balance the workload between the two available. Specifically, in Test 4 (bottom right plot), cAPP switches from *worker1* to *worker2* during the second phase of the test, when the latencies are flipped between the two. This shows that, even with dynamic latency changes, the scheduler can still adapt and choose the optimal worker without any changes to the cAPP script or the function definitions.

A limiting factor here is of course that the Core needs to get updated information from Prometheus, and therefore it strongly depends on the latency monitoring interval to perform optimal decisions in a timely manner. Too long an interval would result in a long period of suboptimal decisions, where the Core bases its policy on old latencies.

7 Related Work

To the best of our knowledge, this is the first work that uses cost equations of functions to govern serverless scheduling.

In general, there is a growing literature focused on techniques that mix one or more locality principles to increase the performance of function execution, assuming some locality-bound traits of functions [2, 6, 7, 10, 21, 23–26, 29, 30, 34, 36–43]. Some of these works focus on applying static analysis techniques for optimising serverless and cloud computing. For instance, Wang et al. [45] use static control and data flow analysis to enhance performance modelling of serverless functions, achieving accurate predictions. Obetz et al. [31] use service call graphs for static analysis of serverless applications, enabling various program analysis applications. Looking at the infrastructure underlying serverless, Garcia et al. [19] present a static analysis technique for computing upper bounds of virtual machine usage in cloud environments, using a technique similar to the one presented in Section 4.

The inference of cost equations and their computation with cost analyzers has been also used for estimating the computational time of programs in an actor model [27] and for analyzing updates of smart contracts balances due to transfers of digital assets [28].

Static-time techniques are also proposed in the field of Implicit Computational Complexity where type inference is used to derive (computational) costs of programs in a direct way, without resorting to cost analyzers. Similar to our approach, the techniques are applied to restricted languages where the cost analysis is decidable (e.g., *loop programs* as in [8]). It is worth to notice that, when such techniques are applied to cAPP, the resulting costs are less precise than those computed with cost analysers. One simple example is Listing 1, when computed according to [8], whose cost is $max(P,B)$ because, in loop programs, conditionals are always nondeterministic.

Besides static analysis, other works used dynamic runtime analyses to visualise measure resource costs [44]. These tools operate by injecting instructions into a program or modifying its runtime to instrument real-time monitoring for collecting information about the behaviour of the program. Contrary to static analyses, dynamic ones requires modifying the runtime of the platform to collect the data needed by the analysis. Moreover, it requires the execution of the programs/functions over an exhaustive set of inputs, which makes the application of the technique more impractical (and could provide a partial "view" of the cases).

8 Conclusion

We introduce a framework that lightens the burden on the shoulders of users by deriving cost information from the functions, via static analysis, into a cost-aware variant of APP that we call cAPP. To show the feasibility of the approach, we present a prototype of such framework where we extract cost equations from functions' code, synthesise cost expressions through off-the-shelf solvers, and implement cAPP to support the specification of cost-aware allocation policies.

Specifically, we demonstrate that one can over-approximate, at scheduling time, the overall latency endured by the invocation of a function f when running on a given worker and use this information to govern its scheduling.

To achieve this result, we present a proposal for an extension of the APP language, called cAPP, to make function scheduling cost-aware. The extension adds new syntactic fragments to APP so that programmers can govern the scheduling of functions towards those execution nodes that minimise their calculated latency (e.g., increasing serverless function performance) and avoids running functions on nodes whose execution time would exceed a maximal response time defined by the user (e.g., enforcing quality-of-service constraints). The main technical insights behind the extension include the usage of inference rules to extract cost equations from the source code of the deployed functions and exploiting dedicated solvers to compute the cost of executing a function, given its code and input parameters. We have demonstrated the feasibility of our proposal by implementing a serverless platform that schedules functions following cAPP scripts. The implementation was obtained by extending the open-source FunLess [15] serverless platform and exploiting the PUBS [3] cost equations solver.

In future work, we will address several key questions that remain open. Specifically, we aim to investigate the scalability and performance of our approach by examining how it would work with more complex examples and evaluating its execution times under varied computational conditions.

Since determining the exact cost of a function is, in principle, undecidable, as future work, we will focus on exploring models and techniques that can make this problem more tractable in practical scenarios. This may include the development of heuristics and over-approximation methods that work effectively for the majority of cases, while ensuring that these approaches remain computationally efficient. Additionally, we are considering architectural solutions to complement these techniques, such as the inclusion of caching systems to store and reuse previously computed costs for repeated function invocations. These systems could significantly reduce overhead by calculating the actual cost of a function only once, avoiding redundant computations.

To further enhance system reliability, we propose integrating timeouts for particularly challenging cost calculations, paired with sensible default strategies to maintain responsiveness. This would ensure the system remains functional even in scenarios where exact costs cannot be computed within a reasonable time frame.

Moreover, we intend to explore the incorporation of user-provided inputs or hints, which could guide our models to more accurate estimations in specific contexts. Finally, we plan to evaluate the effectiveness of our approach by testing it against standard benchmarks, measuring how closely our over-approximations align with actual costs and identifying areas for further refinement.

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). DOI 10.1145/3185768.3186294
3. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: M. Alpuente, G. Vidal (eds.) Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings, *Lecture Notes in Computer Science*, vol. 5079, pp. 221–237. Springer (2008). DOI 10.1007/978-3-540-69166-2_15. URL https://doi.org/10.1007/978-3-540-69166-2_15

4. Apache Software Foundation : Apache Kafka. https://kafka.apache.org/ (2024). Accessed: Jul 2024
5. Azure, M.: Microsoft azure functions. https://azure.microsoft.com/ (2022)
6. Banaei, A., Sharifi, M.: Etas: predictive scheduling of functions on worker nodes of apache openwhisk platform. The Journal of Supercomputing (2021). DOI 10.1007/s11227-021-04057-z
7. Baresi, L., Quattrocchi, G.: Paps: A serverless platform for edge computing infrastructures. Frontiers in Sustainable Cities **3**, 690660 (2021)
8. Ben-Amram, A.M., Kristiansen, L.: On the edge of decidability in complexity analysis of loop programs. International Journal of Foundations of Computer Science **23**(7), 1451–1464 (2012)
9. Broadcom: Rabbitmq. https://www.rabbitmq.com/ (2024). Accessed: Jul 2024
10. 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)
11. Cloud, G.: Google cloud functions. https://cloud.google.com/functions/ (2022)
12. De Palma, G., Giallorenzo, S., Laneve, C., Mauro, J., Trentin, M., Zavattaro, G.: Serverless scheduling policies based on cost analysis. In: M.H. ter Beek, C. Dubslaff (eds.) Proceedings of the First Workshop on Trends in Configurable Systems Analysis, TiCSA@ETAPS 2023, Paris, France, 23rd April 2023, *EPTCS*, vol. 392, pp. 40–52 (2023)
13. 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). DOI 10.1109/ICWS55610.2022.00056
14. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Custom Serverless Function Scheduling Policies: An APP Tutorial. In: G. Dorai, M. Gabbrielli, G. Manzonetto, A. Osmani, M. Prandini, G. Zavattaro, O. Zimmermann (eds.) Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022), *Open Access Series in Informatics (OASIcs)*, vol. 111, pp. 5:1–5:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2023). DOI 10.4230/OASIcs.Microservices.2020-2022.5
15. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Funless: Functions-as-a-service for private edge cloud systems. In: IEEE International Conference on Web Services, ICWS 2024. IEEE (2024)
16. 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). DOI 10.1007/978-3-030-65310-1_29
17. De Palma, G., Giallorenzo, S., Mauro, J., Zavattaro, G.: Allocation priority policies for serverless function-execution scheduling optimisation. In: E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, H. Motahari (eds.) 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). DOI 10.1007/978-3-030-65310-1_29
18. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: J. Garrigue (ed.) Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings, *Lecture Notes in Computer Science*, vol. 8858, pp. 275–295. Springer (2014). DOI 10.1007/978-3-319-12736-1_15. URL https://doi.org/10.1007/978-3-319-12736-1_15
19. Garcia, A., Laneve, C., Lienhardt, M.: Static analysis of cloud elasticity. Sci. Comput. Program. **147**, 27–53 (2017)
20. 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)
21. 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). DOI 10.1145/3477132.3483541
22. 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 (2019)
23. 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)
24. 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)
25. Kotni, S., Nayak, A., Ganapathy, V., Basu, A.: Faastlane: Accelerating function-as-a-service workflows. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21), pp. 805–820. USENIX Association (2021)
26. Kuntsevich, A., Nasirifard, P., Jacobsen, H.A.: A distributed analysis and benchmarking framework for apache openwhisk serverless platform. In: Proc. of Middleware (Posters), pp. 3–4 (2018)
27. Laneve, C., Lienhardt, M., Pun, K.I., Román-Díez, G.: Time analysis of actor programs. J. Log. Algebraic Methods Program. **105**, 1–27 (2019)
28. Laneve, C., Sacerdoti Coen, C.: Analysis of smart contracts balances. Blockchain: Research and Applications **2**(3), 100020 (1–22) (2021)
29. Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., Sukhomlinov, V.: Agile cold starts for scalable serverless. In: Proc. of HotCloud 19. USENIX Association, Renton, WA (2019)
30. Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 57–70 (2018)
31. Obetz, M., Patterson, S., Milanova, A.L.: Static call graph construction in AWS lambda serverless applications. In: C. Delimitrou, D.R.K. Ports (eds.) 11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019. USENIX Association (2019). URL https://www.usenix.org/conference/hotcloud19/presentation/obetz
32. OpenFaaS: Openfaas. https://www.openfaas.com/ (2022)
33. OpenWhisk, A.: Apache openwhisk. https://openwhisk.apache.org/ (2022)
34. Sampé, J., Sánchez-Artigas, M., García-López, P., París, G.: Data-driven serverless functions for object storage. In: Middleware, Middleware '17, pp. 121–133. ACM (2017). DOI 10.1145/3135974.3135980. URL https://doi.org/10.1145/3135974.3135980
35. Services, A.W.: Introducing aws lambda. https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/ (2022)
36. Shahrad, M., Balkind, J., Wentzlaff, D.: Architectural implications of function-as-a-service computing. In: Proc. of MICRO, pp. 1063–1075 (2019)
37. 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)
38. Shillaker, S., Pietzuch, P.: Faasm: Lightweight isolation for efficient stateful serverless computing. In: Proc. of USENIX ATC, pp. 419–433. USENIX Association (2020)
39. Silva, P., Fireman, D., Pereira, T.E.: Prebaking functions to warm the serverless cold start. In: Proc. of Middleware, Middleware '20, pp. 1–13. ACM, New York, NY, USA (2020). DOI 10.1145/3423211.3425682
40. 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)
41. Solaiman, K., Adnan, M.A.: Wlec: A not so cold architecture to mitigate cold start problem in serverless computing. In: 2020 IEEE International Conference on Cloud Engineering (IC2E), pp. 144–153 (2020). DOI 10.1109/IC2E48712.2020.00022
42. 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 (2020). DOI 10.14778/3407790.3407836
43. Suresh, A., Gandhi, A.: Fnsched: An efficient scheduler for serverless functions. In: WOSC@Middleware, pp. 19–24. ACM (2019). DOI 10.1145/3366623.3368136
44. 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)
45. Wang, R., Casale, G., Filieri, A.: Enhancing performance modeling of serverless functions via static analysis. In: J. Troya, B. Medjahed, M. Piattini, L. Yao, P. Fernández, A. Ruiz-Cortés (eds.) Service-Oriented Computing - 20th International Conference, ICSOC 2022, Seville, Spain, November 29 - December 2, 2022, Proceedings, *Lecture Notes in Computer Science*, vol. 13740, pp. 71–88. Springer (2022). DOI 10.1007/978-3-031-20984-0_5
46. Wasmtime. `https://wasmtime.dev/` (2023)
47. Webassembly. `https://webassembly.org/` (2023)
48. Webassembly system interface. `https://wasi.dev/` (2023)
49. Wen, J., Chen, Z., Jin, X., Liu, X.: Rise of the planet of serverless computing: A systematic review. ACM Trans. Softw. Eng. Methodol. **32**(5), 131:1–131:61 (2023). DOI 10.1145/3579643. URL `https://doi.org/10.1145/3579643`