Towards a Function-as-a-Service Choreographic Programming Language: Examples and Applications

GIUSEPPE DE PALMA, Università di Bologna, Italy and OLAS team, INRIA, France
SAVERIO GIALLORENZO, Università di Bologna, Italy and OLAS team, INRIA, France
JACOPO MAURO, University of Southern Denmark, Denmark
MATTEO TRENTIN, Università di Bologna, Italy, OLAS team INRIA, France, and University of Southern Denmark, Denmark
GIANLUIGI ZAVATTARO, Università di Bologna, Italy and OLAS team, INRIA, France

1 INTRODUCTION

Choreographic Programming (CP) is a language paradigm whereby software artefacts, called choreographies, specify the behaviour of communicating participants. Choreographic programming is famous for its correctness-by-construction approach to the development of concurrent, distributed systems. In this paper, we illustrate FaaSChal, a proposal for a CP language tailored for the case of serverless Function-as-a-Service (FaaS). In FaaS, developers define a distributed architecture as a collection of stateless functions, leaving to the serverless platform the management of deployment and scaling [9]. We provide a first account of a CP language tailored for the FaaS case via examples that present some of its relevant features, including projection. In addition, we showcase a novel application of CP. We use the choreography as a source to extract information on the infrastructural relations among functions so that we can synthesise policies that strive to minimise their latency while guaranteeing the respect of user-defined constraints.

2 BACKGROUND ON SERVERLESS

We start with a brief overview of serverless computing and the platforms that support it.

Developers make a serverless application out of software units called functions, which run in short-lived environments triggered by different kinds of events. When an event such as an HTTP request, database change, file upload or scheduled trigger occurs, the FaaS platform runs an instance of the function(s) liked to that event. The platform runs the code after initialising an execution environment, which is a secure and isolated context that provides all the resources needed for the function lifecycle, typically implemented with virtual machines and containers.

We use Figure 1, which depicts a typical serverless platform architecture, to briefly introduce the components and processes behind the execution of serverless functions, useful to contextualise our contribution. The main components of a serverless platform, as shown in Figure 1, are the Controller and the Workers. The Controller can receive requests to execute functions from various media, e.g., an HTTP gateway or a publish-subscribe messaging service like the Simple Notification Service (SNS), by AWS. These media expose endpoints which entities—like Web applications, IoT devices, and databases, as well as running serverless functions—can trigger to invoke the execution of a function. The Controller handles the allocation of functions on Workers based on the latter's status (given a set of metrics like CPU and memory usage, collected by the Controller). The Controller also handles the storage/retrieval of the invocation/result from/to the caller. In particular, the scheduler determines which Worker should execute an invoked function based on factors such as current load, function requirements, and resource availability. Once it receives the request to execute a function, the Worker creates a new instance of that function, handling its execution environment lifecycle, including provisioning, scaling, and teardown.

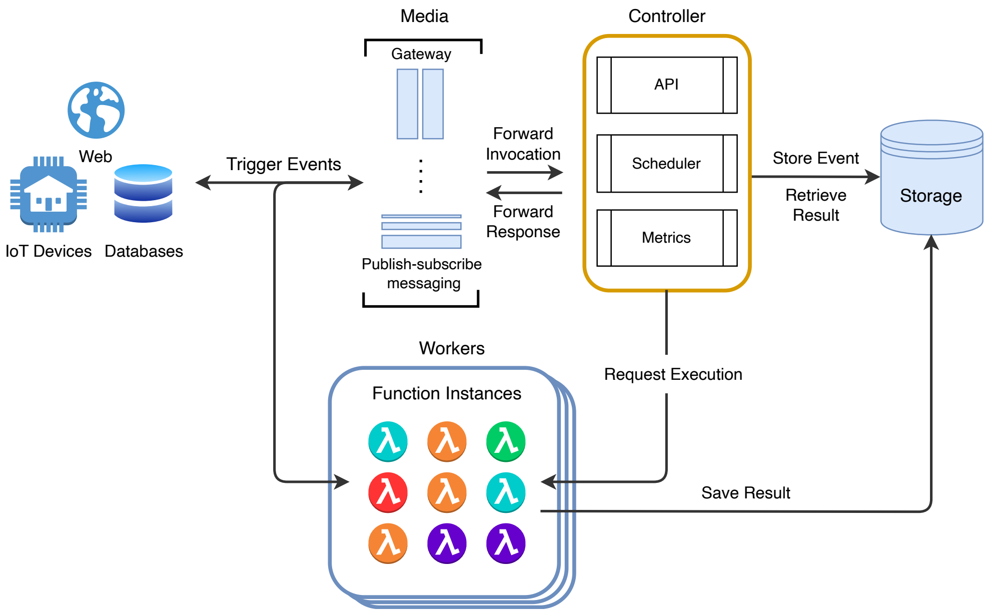

Fig. 1. A typical serverless platform architecture.

3 FAASCHAL BY EXAMPLE

We introduce FaaSChal with the example shown in Fig. 2, where a function orchestrates (a simplified version of) the training of an image AI model. The whole routine is started by a `user`, who defines the queries to extract from some databases the labels and images used in the training. The user sends to a serverless function, called `f`, the queries, which `f` uses to access two separate databases and obtain the images and labels (ordered and paired). The function then launches the training of each pair image-label in a separate function, called `g`, which finally triggers a third function, called `h`, that acts as merger/integrator of the trained weights of the model into a third database. The training process is asynchronous, i.e., the user receives a response from the orchestrator function as soon as it terminates the launching of the training of all image-label pairs.

In the choreography, we distinguish three main kinds of entities, found in the preamble at lines 1–3. We start commenting on them from line 3 upwards. The first kind is that of `services`, which are passive entitites that interact in the choreography providing labelled inbound request-response operations—enumerated within curly brackets in the preamble. For example, in Fig. 2, the service `DB1` offers a request-response operation labelled `getData`. In general, the caller can discard the response in a request-response interaction, consuming the operation in a one-way fashion. Going up, we find FaaS `stateless` functions which: *a*) must be triggered/started by some other active entity via their media endpoints, declared within brackets (e.g., `Gateway`, `SNS` and the trigger endpoint annotations), *b*) provide a request-response triggering behaviour (which the triggerer can invoke in a one-way fashion, discarding their response), *c*) after their triggering, they cannot receive other messages (but they can send outbound requests, in both one-way and request-response fashion). The last kind of entity is that of `stateful` participants, which are traditional active processes (no triggering) that can interact with the other entities.

```
 1  stateful: user
 2  stateless: f{ Gateway }, g{ SNS:"aws:sns" }, h{ SNS:"aws:sns" }
 3  services: DB1{ getData }, DB2{ getData }, DB3{ storeData }
 4  import Collections::zip as zip@f
 5  import Model::fit as fit@g
 6  import Model::integrate as int@h

 6  def main( queries@user )
 7    queries@user <-Gateway-> f do | queries@f |
 8      queries@f.labels <-> DB1: getData ▶ labels@f
 9      queries@f.images <-> DB2: getData ▶ images@f
10      for pair@f in zip(labels@f, images@f) do
11        pair@f
12          ▶ -SNS-> g ▶ fit
13          ▶ -SNS-> h ▶ int
14          ▶ -> DB3: storeData
15      end
16    end with "All training jobs started"@f
17  end
```

Fig. 2. The choreography of the serverless AI training program.

In the choreography, we can import operations (e.g., from libraries), as showcased at lines 4–6, where the stateless functions `f`, `g`, and `h` resp. `import` the operations `zip` from the `Collections` library, `fit` and `int`(egrate) from the `Model` one. Since the `import` instruction has a target function (e.g., `f`), that functionality is available/imported only at/by that function.

The statement we find at line 7 (closing at line 16) is a request-response from the `user` to the stateless function `f`, of the form

```
exp1@role1 <- MEDIUM -> role2 do [| opt_var@role2 |] ... end [ with exp2@role2 ]
```

From left to right, we evaluate the expression `exp` at `@role1` and send its value via the `MEDIUM` the function is available at (e.g., the `Gateway` at line 7 of Fig. 2) to trigger the execution of function `role2`. This function can optionally bind the data sent from `role1` to a local variable (`opt_var`) and execute the code within the block until its closure (`end`). The function sends back a response, which is empty unless specified through the suffix of the closure with the clause `with` followed by an expression at that function (`exp@role2`) evaluated to return a response—considering the body of the triggering block and the initial binding within its scope.

Within the body of the block, we first find the request-response invocations (`<->`) to the respective operations `getData` of `DB1` and `DB2` to retrieve the data (resp. `labels` and `images`) by `f`.[1]

To bind to a variable the value received by `f` as the response to the request to the databases (both for `DB1` and `DB2`), we use the forward operator ▶. The idea behind ▶, inspired by Choral [7], is to naturally support a left-to-right reading of the interactions in a choreography. Without ▶, one

[1]The operation `getData` in the example is blocking and, thus, the second call to `DB2` waits until the completion of the previous call (which might take a long time) to proceed. To increase efficiency, a simple extension of the language can include a parallel operator, like the one found in AIOCJ [2], to send the two `getData` calls in parallel, realising a join pattern.

would need to write an assignment like `var@b = data@a <- MEDIUM -> @b`, forcing the user to first parse the expression on the right[2] and then go back to the assignment of the resulting value to the `variable` on the left.

Like in Choral, users can call unary functions with ▶ in a point-free style—i.e., `exp ▶ f1 ▶ f2` is syntactic sugar for `f2(f1(exp))`—which we extend to also work as a variable assignment operator.

At lines 11–15, after the retrieval of the `labels` and `images`, `f` `zips` them together and, `for` each `pair`, it triggers a new instance of the function `g`, sending to it the pair. Note that the triggering of `g` is one-way, (represented by the communication `-MEDIUM->`), which allows us to adopt the lightweight notation found, e.g., at line 12, instead of the more complex one for request-responses we commented for `f`, above. At triggering/reception, `g` performs the training (via the `fit` operation) and then triggers function `h` to `integrate` the data into `DB3` (invoking `storeData` as a one-way operation).

A notable characteristic of `FaaSChal` is that there is no need for coordination in constructs such as loops and conditionals when the interaction concerns only `stateless` functions. Indeed, choreographic languages where processes are stateful (and usually engage in a kind of session-oriented interaction) need either the enforcement of knowledge of choice or amendments such as auxiliary communications to ensure the causality/connectedness of the actions among the processes [1, 2, 7, 10]. When conditionals/loops concern only stateless functions (which, once triggered, cannot engage in further synchronisations except for outbound request-responses) these issues do not arise. As a consequence of this triggering behaviour for `stateless` functions, at each loop at lines 11–15, we bind the identifiers `g` and `h` to resp. new function instances, i.e., at each loop `f` requests the instantiation of new copies of said functions. To further clarify the relationship between knowledge of choice and stateless functions, consider the example below

```
if exp@f then f -SNS-> g else f -SNS-> h
```

If `g` and `h` were stateful processes, we would need to inform them both on which direction the choreography shall proceed, according to the choice taken by `f` (resulting from the evaluation of `exp`). Lacking this piece of coordination, depending on the choice made by `f`, either `g` or `h` would wait for `f`'s call indefinitely (since either of them does not know that `f` selected the other branch), exposing the program to deadlocks. On the contrary, since all roles are `stateless` functions, there is no need to inform, e.g., `g` that `f` is choosing the `else` branch, because `g` is not running (it is triggered by `f`'s call) and has no risk of ending up in a deadlock state.

4 PROJECTION

We show one of the typical applications of choreographic programming, which is the generation of local code that implements the semantics of the source choreography. In particular, this section aims to provide code examples that FaaS developers can use to get a better grasp of the semantics of the example in Fig. 2. In the following, we use pseudocode inspired by Ruby and Python and annotate the code to indicate to which entity it corresponds and other information useful for deployment, e.g., the name that the FaaS platform shall bind to the function and how it shall expose the function for consumption (its `MEDIUM`).

We remind that `services` are passive entities which the other participants use as always-available operations, thus, the produced local code does not include the sources for `DB1`, `DB2`, and `DB3`.

Without going too much into the details of the pseudocode, we notice the most salient features linked to serverless function programming. First, in the code of functions, note the presence of a `main` procedure, which is the one canonically invoked by the platform to execute the behaviour

[2]Left-to-right: take the `data` from `a`, send it via `MEDIUM`, instantiate `b` and return the data sent by `a` to `b` as the expression's result.

```
# user code
def main( queries )
  Gateway.invoke( "f", queries )
end
```

```
# deploy as f, trigger GATEWAY
def main( queries )
 labels = DB1.getData( queries.labels )
 images = DB2.getData( queries.images )
 for pair in labels.zip( images ) do
  triggerFn( "g", "aws:sns", pair )
 end
 return { code: 200, body: "All training jobs started" }
end
```

```
# deploy as g, trigger SNS
import Model::fit as fit

def main( _ )
 return triggerFn( "h", "aws:sns", fit( _ ) )
end
```

```
# deploy as h, trigger SNS
import Model::integrate as int

def main( _ )
  return DB3.storeData( int( _ ) )
end
```

Fig. 3. Projections of the example from Fig. 2.

of the function. Second, we find the automatic injection of FaaS platform auxiliary functionalities (one can make these functionalities platform-agnostic by providing different implementations of the same API parametrised w.r.t. specific deployments) provided to trigger the functions, e.g., `Gateway.invoke` and `triggerFn` resp. found in the `user`'s and functions' code. To keep the example lightweight, we did not introduce distinct syntaxes for one-ways and request-responses—e.g., one can provide the same API parametrised to either send a request and wait for a response to return it or return immediately.

5 APPLICATIONS: THE CASE OF FUNCTION SCHEDULING POLICIES

The scheduling of functions, i.e., the allocation of functions over the available workers, can substantially influence their performance. Indeed, effects like *code locality* [8]—due to latencies in loading function code and runtimes—or *session locality* [8]—due to the need to authenticate and open new sessions to interact with other services—can substantially increase the run time of functions. Usually, serverless platforms implement opinionated policies that favour some performance principle tailored for one or more of these locality principles. Besides performance, functions can have functional requirements that the scheduler shall consider. For example, users might want to 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.

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 De Palma et al. [3–6] to introduce a YAML-like declarative language used to specify scheduling policies to govern the allocation of serverless functions on the nodes that make up a cluster, called *Allocation Priority Policies* (APP). Thanks to APP, the same platform can support different scheduling policies, each tailored to meet the specific needs of a set of related functions.

As an example of an application of FaaSChal, we introduce a variant of APP. We extract locality principles that emerge from the choreography—e.g., the loop where `f` spawns many `g`s and `h`s presents a locality linked to the time it takes `f` to contact SNS and issue the call. Then, given a description of the infrastructure topology and possible user-defined constraints on the allocation of functions, we synthesise an APP script that strives to orient the scheduling of functions to minimise their latency of execution, while guaranteeing the respect of the constraints imposed by the user.

5.1 The APP Language

To define function-specific policies, APP assumes the association of each function with a tag. In our examples, we directly use the function's reference name as the tag, but the relation can be one-to-many to specify a policy shared among a set of functions. Then, APP associates a tag to a policy, so that, at runtime, the scheduler of the platform can pair each function with its APP policy and follow the latter's scheduling logic.

In the APP variant we showcase, we assume to have the nodes of the cluster associated with a label, i.e., several nodes can share the same label, e.g., `group1`. In an APP script, users can specify a sequence of blocks (each identified by YAML's list unit `-`) associated with a tag. Each block indicates on which nodes the scheduler can allocate the function. At function invocation, the scheduler tries to allocate the function following the logic in the first block, passing to the next only if none of the machines specified in that block can host the function, and so on (exhausting all blocks causes the invocation's failure). In APP, these nodes take the name of `workers`, which is also the keyword used in the scripts to specify the label of the nodes for that block. Besides workers, APP lets users specify the strategy the scheduler shall use to select among the indicated workers (e.g., choose at random, for load-balancing) and when a worker becomes invalid (e.g., setting a maximal threshold of concurrent functions running on it). The variant we present does not use these options—but it is valid APP code nonetheless, since, when omitted, APP uses default strategy and invalidation rules. The only additional element in our variant's syntax is that of `affinity`. This option accepts a list of tags, where each tag can be prefixed by a `!`. For instance, if we have a tag `a` with `affinity: b, !c`, it means that function `a` is affine with `b`-tagged functions and anti-affine with `c`-tagged ones. Schedule-wise, "anti-affine" means that we cannot allocate the function we want to schedule on a worker that contains instances of any of its anti-affine functions—from the example, we cannot allocate an instance of `a` on workers hosting instances of `c`. Complementarily, "affine" means that we can allocate the function under scheduling only on workers that host at least one instance of each of its affine functions—from the example, we can allocate an instance of `a` only on workers which have at least one instance of `b` running on them. These (anti-)affinity constraints are useful to specify e.g., security concerns (like separating the execution of trusted from untrusted functions to avoid possible security risks) and performance (like the allocation of functions on the same worker to let them reuse a pool of connections to a database). In APP, `affinity` constraints are not symmetric, i.e., if we set `f` affine with `g` it does not imply that `g` is affine with `f` (but one can symmetrise the relation by adding the complementary constraint).

5.2 Extraction of Locality Principles and Generation of APP Scripts

Briefly, we define the extraction of locality principles from a choreography by attributing `data locality`—the principle that the closer the function is to the data the lower its latency, proportional to faster access to the data repository—to all functions that access a database (we omitted this information from Fig. 2, but it is simple to annotate `services` accordingly); `call locality` comes from interactions among functions, in particular the repeated ones, which can benefit from running on machines with faster access to the medium that accepts/delivers the call; `code locality` comes from the re-use of loaded code in a worker's memory (i.e., avoiding fetching and loading times). On the right of Fig. 4, we find an example of the extracted localities from Fig. 2.

The last ingredient is the infrastructure topology and the constraints that users might want to impose on the functions. We report on the right of Fig. 4 an example of such a schema. In the example, the writing `( a, b ): N` indicates that `a` and `b` have a connection speed (symmetric) of `N` (we can

```
# Extracted localities from choreo
data locality:
  ( f, DB1 ),
  ( f, DB2 ),
  ( h, DB3 ),
call locality:
  ( f, g, SNS, 1:n ) # n = n. of labels/imgs
  ( g, h, SNS, 1:1 )
code locality:
  ( g, h ) # Model
```

```
# Infrastructure topology (speed)
( DB1, group1 ): 100
( DB2, group2 ): 80
( DB2, group1 ): 20
( DB3, group2 ): 50
( SNS, group1 ): 50
( SNS, group2 ): 50

# User-defined constraints
anti-affine: ( f, g ), ( g, g ), ( h, g )
```

Fig. 4. Left: locality principles extracted from the example from Fig. 2. Right: infrastructure topology and user-defined function scheduling constraints.

```
f:
 - workers: group1
   affinity: f, !g
 - workers: group1
   affinity: !g
```

```
g:
 - workers: *
   affinity: !f, !g, !h
```

```
h:
 - workers: group2
   affinity: h, !g
 - workers: group2
   affinity: !g
```

Fig. 5. APP script generated from the localities, topology, and user-defined constraints from Fig. 4

abstract away the unit of measure, as long as all items use the same), e.g., `( DB1, group1 ): 100` means that the machines in group1 have a (fast) connection of 100 with DB1. Note that the absence of infrastructural pairs are as important as the present ones, e.g., the fact that there is no couple `( DB3, group1 )` in the schema means that no machine in group1 can reach DB3.

For compactness, we understand the specification of user-defined (anti-)affinity constraints of the schema as symmetric, e.g., if `(a, b)` are anti-affine, we read this constraint as "neither a can run on a worker where a b is running nor b can run on a worker where an a is running". Above, we set f and g anti-affine to avoid running f on a worker loaded with g (which performs heavy computations to train the model) and vice versa. Similarly, we avoid placing more instances of the function g on the same worker and placing the functions g and h together.

Given the ingredients from Fig. 4, we can obtain the APP script reported in Fig. 5. In the script, we have two subsequent blocks for the allocation of function f. In both blocks, we try to allocate f on group1 because this is the only group of machines that can access both DB1 and DB2. Considering affinities, in the first block, we try to allocate f with other instances of the same function to exploit connection pooling to DB1 and DB2. Following the user-defined constraints, we set a negative affinity with function g (written !g). In the second block, f can run on a worker without another instance of the same function running on it (this item avoids the problem of self-affinity, which would prevent the allocation of an initial f), yet we preserve the anti-affinity with g.

Since g has no "favourite" group in the infrastructure (both group1 and group2 reach the same speed w.r.t. the only infrastructural locality it presents, SNS), the policy for g allocates the function on any available worker *. Following the anti-affinity constraints specified by the user, we mark g anti-affine with f (as per the symmetric interpretation of the anti-affinity constraints above), itself, and h (similarly to the anti-affinity with f).

Finally, we specify that function `h` can only run on machines of `group2` since these are the only ones that can reach `DB3`. Following the user-defined constraints, we have `h` anti-affine with `g` and `h` affine with itself to exploit connection pooling.

Note that the synthesis of the APP script does not include all extracted localities (cf. Fig. 4). For instance, `call locality` did not influence the script, due to the fact that `group1` and `group2` have the same speed w.r.t. `SNS`. Another example is the `code locality` between `g` and `h`, which share the same code dependency (`Model`, cf. Fig. 2) but which we could not indicate as affine in the APP script due to the user-defined anti-affinity constraints (which have priority over the extracted localities).

6 CONCLUSION

We illustrate `FaaSChal`, a language proposal for exploring the design space of applying CP to FaaS programming. Besides showcasing relevant features of a CP language for FaaS, we provide application examples that target projection and the management of the scheduling of functions.

In the future, we plan to conduct a formal investigation into the expressiveness and limitations of `FaaSChal`, considering other use cases taken from realistic FaaS architectures and deepening the analysis of the interplay between `services`, `stateful` participants, and `stateless` functions. Another interesting direction is to formally analyse the processes of extraction of locality principles from choreographies and the mechanisation of the synthesis of APP scripts.

REFERENCES

[1] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On Global Types and Multi-party Sessions. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6722)*, Roberto Bruni and Jürgen Dingel (Eds.). Springer, 1–28. https://doi.org/10.1007/978-3-642-21461-5_1

[2] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2017. Dynamic Choreographies: Theory And Implementation. *Logical Methods in Computer Science* 13, 2 (2017). https://doi.org/10.23638/LMCS-13(2:1)2017

[3] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. 2022. 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*, Claudio Agostino Ardagna, Nimanthi L. Atukorala, Boualem Benatallah, Athman Bouguettaya, Fabio Casati, Carl K. Chang, Rong N. Chang, Ernesto Damiani, Chirine Ghedira Guegan, Robert Ward, Fatos Xhafa, Xiaofei Xu, and Jia Zhang (Eds.). IEEE, 337–342. https://doi.org/10.1109/ICWS55610.2022.00056

[4] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. 2023. Custom Serverless Function Scheduling Policies: An APP Tutorial. In *Joint Post-proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022) (Open Access Series in Informatics (OASIcs), Vol. 111)*, Gokila Dorai, Maurizio Gabbrielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmermann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:16. https://doi.org/10.4230/OASIcs.Microservices.2020-2022.5

[5] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. 2023. Formally Verifying Function Scheduling Properties in Serverless Applications. *IT Professional* 25, 06 (nov 2023), 94–99. https://doi.org/10.1109/MITP.2023.3333071

[6] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, and Gianluigi Zavattaro. 2020. 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)*, Eleanna Kafeza, Boualem Benatallah, Fabio Martinelli, Hakim Hacid, Athman Bouguettaya, and Hamid Motahari (Eds.). Springer, 416–430. https://doi.org/10.1007/978-3-030-65310-1_29

[7] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. *ACM Trans. Program. Lang. Syst.* 46, 1, Article 1 (jan 2024), 59 pages. https://doi.org/10.1145/3632398

[8] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16).*

[9] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing.* Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley.

[10] Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. 2013. Amending Choreographies. In *Proceedings 9th International Workshop on Automated Specification and Verification of Web Systems, WWV 2013, Florence, Italy, 6th June 2013 (EPTCS, Vol. 123)*, António Ravara and Josep Silva (Eds.). 34–48. https://doi.org/10.4204/EPTCS.123.5