FaaS Orchestration of Parallel Workloads

Daniel Barcelona-Pons
Universitat Rovira i Virgili
Tarragona, Spain
daniel.barcelona@urv.cat

Pedro García-López
Universitat Rovira i Virgili
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
pedro.garcia.lopez@ibm.com

Álvaro Ruiz
Universitat Rovira i Virgili
Tarragona, Spain
alvaro.ruiz@urv.cat

Amanda Gómez-Gómez
Universitat Rovira i Virgili
Tarragona, Spain
amanda.gomez@urv.cat

Gerard París
Universitat Rovira i Virgili
Tarragona, Spain
gerard.paris@urv.cat

Marc Sánchez-Artigas
Universitat Rovira i Virgili
Tarragona, Spain
marc.sanchez@urv.cat

Abstract

Function as a Service (FaaS) is based on a reactive programming model where functions are activated by triggers in response to cloud events (e.g., objects added to an object store). The inherent elasticity and the pay-per-use model of serverless functions make them very appropriate for embarrassingly parallel tasks like data preprocessing, or even the execution of MapReduce jobs in the cloud.

But current Serverless orchestration systems are not designed for managing parallel fork-join workflows in a scalable and efficient way. We demonstrate in this paper that existing services like AWS Step Functions or Azure Durable Functions incur in considerable overheads, and only Composer at IBM Cloud provides suitable performance.

Successively, we analyze the architecture of OpenWhisk as an open-source FaaS systems and its orchestration features (Composer). We outline its architecture problems and propose guidelines for orchestrating massively parallel workloads using serverless functions.

CCS Concepts • **Computer systems organization → Cloud computing**.

Keywords Serverless, FaaS, orchestration, event-based

ACM Reference Format:
Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. 2019. FaaS Orchestration of Parallel Workloads. In *5th Workshop on Serverless Computing (WOSC '19), December 9–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3366623.3368137

1 Introduction

Serverless Function as a Service (FaaS) is becoming a very popular model in the cloud thanks to its simplicity, billing model and inherent elasticity. The FaaS programming model is considered event-based. That is, functions can be activated (triggered) in response to specific cloud events (e.g. a change in an object store like Amazon S3). According to the Reactive Manifesto [2], existing FaaS incarnations can be considered reactive systems. They comply with the four required properties: responsiveness, resilience, elasticity and message-driven communication.

The FaaS model has also proven ideally suited for embarrassingly parallel computing tasks. For example, systems like PyWren [12, 15] or ExCamera [7] have demonstrated massively parallel computations (MapReduce jobs, video encoding) over serverless functions. Specifically, the disaggregation of compute (cloud functions) and storage (object store) services facilitated the necessary elasticity and flexibility of the system to process huge volumes of data in parallel.

Unfortunately, PyWren and ExCamera required their own *ad-hoc* external orchestration services to synchronize the parallel execution of functions. In particular, when the PyWren client launches a map job with N functions, it polls Amazon S3 until all N results appear in the S3 bucket. ExCamera also relied on an external rendezvous server to synchronize the parallel executions.

Clearly, such *ad-hoc* orchestrator services do not comply with the four requirements claimed by Amazon for a serverless service: *(i) no server management, (ii) flexible scaling, (iii) pay for value, and (iv) automated high availability* [3]. No server management implies that users do not need to provision or maintain any servers. Flexible scaling entails that the application can be scaled automatically through units of consumption (e.g., throughput, memory) rather than units of individual servers. Pay for value is to pay for the use of consumption units rather than server units. And finally, automated high availability ensures that the system must provide built-in availability and fault tolerance.

We claim that if serverless functions follow a trigger-based model, the FaaS orchestration system should also be trigger-based. This means that in a DAG (direct acyclic graph) workflow, the termination of one or many functions should trigger the next stage (function) using asynchronous events.

If the orchestration system is not trigger-based, reactive, and asynchronous, the orchestrator will require some synchronous blocking actions to wait for the termination of cloud functions, or the transition between stages. Such synchronous blocking actions imply unnecessary pull requests compared to a pure asynchronous push-based approach. This also requires an active orchestrator with increased billing for long-running workflows.

In this document, we advocate for a *trigger-based serverless orchestration service for massively parallel workflows.* To this end, this paper presents the following contributions:

- We first compare parallel fork-join workloads with three different services: AWS Step Functions, Azure Durable Functions, and IBM/Apache Composer.
- We then analyze Apache OpenWhisk and Composer. We describe the architecture and life cycle, showing the overheads and limitations, concluding that workflow orchestration must be reactive and event-based.

2 FaaS function orchestration

Serverless FaaS is based on the event-driven programming model [6]. And event-driven systems have become the backbone of a myriad of distributed systems because of their decoupled and scalable architecture. Therefore, a logic reasoning is to use a reactive system for FaaS function composition and orchestration. We take the definition of Reactive System from the Reactive Manifesto [2]: a system is considered reactive if it fulfills all four properties described in the manifesto. First, a reactive system is responsive. The system provides rapid and consistent responses. Second, it is resilient. When a failure happens, the system remains responsive. Third, it is elastic. The system reacts to varying workload and continues to provide the same responsiveness. And fourth, it is message-driven. Component-to-component communication is solved with asynchronous message-passing, which greatly helps to make the system responsive, resilient and elastic.

The most relevant work aiming to provide reactive orchestration of serverless functions is the *Serverless Trilemma* (ST) paper [5]. The authors advocate for reactive run-time support for cloud function orchestration, and define the so-called trilemma: first, functions must be considered *black boxes*; second, function composition must obey a *substitution principle* with respect to synchronous invocation (i.e., a composition should also behave like a function); and third, invocations must not be *double-billed.* If all three properties are fulfilled, a system is considered ST-safe (*Serveless Trilemma*-safe).

As a case study of a ST-safe system, the paper also presents a solution for sequential compositions atop Apache OpenWhisk. However, the authors explicitly renounce to event-trigger function composition due to two technical problems in the OpenWhisk implementation. First, the system lacks function termination events to trigger subsequent functions. Second, the inability to transfer results from function to function in a composition (e.g., a sequence), so as to respond with the final result to the invoking client (synchronous invocation). Consequently, Baldini et al. propose a solution for ST-safe sequence composition by using an internal procedure of OpenWhisk (i.e., active ack)[1] as termination events/triggers. With extensive changes in the core using these triggers, the system is able to orchestrate sequential compositions. This same idea evolved later on into *conductor actions* [4], an extension of OpenWhisk that allows the system to chain function invocations with state machine logic. This is the technology used by Apache Composer [9].

Azure Durable Functions (ADF) is in the same line to enable function orchestration within the FaaS service. However, in this case the implementation is more of a wrapper around functions, and not a modification of the service core.

Other systems have tried to orchestrate functions without internal support. They can be classified into two types: (i) functions to orchestrate functions; and (ii) external client schedulers. In the first category (e.g., [13, 14]), the orchestration is performed inside a serverless function. However, this approach suffers *double billing*: the orchestrator function waits for the execution of the orchestrated functions (both billed at the same time). In the second type (e.g., [7, 12]), an external scheduler coordinates the functions, thereby avoiding double billing. But, in this case, the *substitution principle* is violated: compositions cannot be treated as functions since they are external to the system itself. Additionally, the external scheduler cannot be considered a serverless service.

The last category also includes previous approaches for workflow management: Apache Airflow, Oozie [11] or even BPMN workflow engines like Camunda [1] rely on a dedicated long-running stateful workflow-execution engine that manages the state machine of the orchestration. In this line, Google Cloud Composer relies on a managed Airflow deployment (using a Kubernetes cluster per client) with a minimum of three nodes to ensure fault-tolerance and responsiveness.

AWS Step Functions (ASF) is Amazon's serverless orchestration service providing fault-tolerant workflows as state machines. Unlike Google Cloud Composer, ASF is not offered as a managed cluster; it is a serverless service that bills per step transition in the workflow. Despite being serverless, ASF follows an external client-scheduler model where orchestrations are not themselves first-class functions, and therefore, not ST-safe.

[1]Active ack is a pipeline bypass strategy in OpenWhisk to minimize request-response latency.

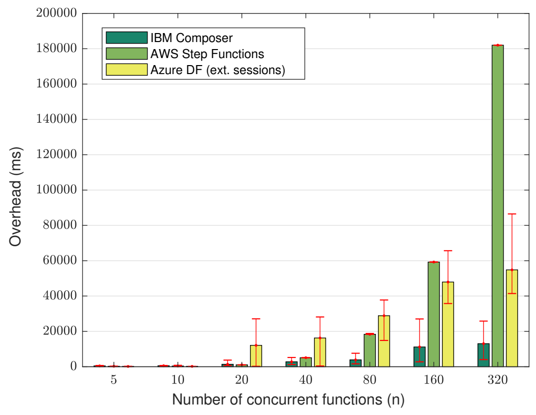

Figure 1. Parallelism overhead

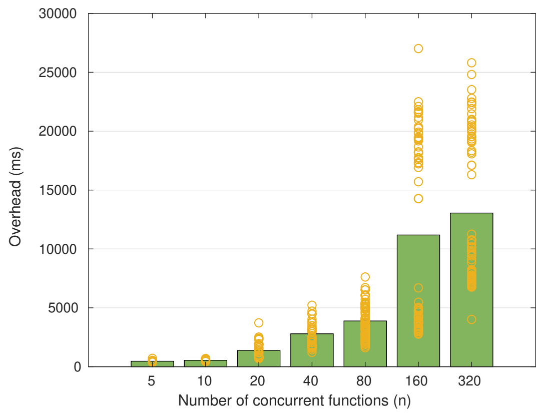

Figure 2. Variability in Composer's parallel executions

3 Benchmarking parallel fork-join using FaaS orchestration systems

In this section, we benchmark the overhead added by different FaaS orchestrators offered by cloud providers. We pursue to assess the performance of these services when managing big amounts of parallel tasks in a workflow. Our candidates are Step Functions at AWS, Azure Durable Functions at Microsoft Azure, and Apache OpenWhisk Composer at IBM Cloud.

A previous work [10] already studied the overhead of these services for sequences and parallel execution in a fork-join manner. In the present work, we focus on the latter. We update the data of the benchmark and extend it with up to 320 parallel tasks in the orchestration (the previous work goes up to 80). Additionally, we include Apache OpenWhisk Composer to the experiment.

For this experiment, we (programmatically) define a workflow in each service with a single parallel stage conformed by *n* parallel instances of the same task, with *n* ranging from 5 to 320, and doubling each time. This task has a fixed duration of 20 seconds. Consequently, any execution of the experiment should ideally last 20 seconds, irrespective of the parallelism or environment. To put it in another way, in an ideal system with no overhead, the execution time of the *n* concurrent tasks should match that of a single task. Therefore, we compute the overhead of the orchestration system by subtracting the fixed time of a single task, namely 20 seconds, from the total execution time. Note that this overhead is not only a consequence of the system itself (i.e., implementation-wise), but it also depends on the type of resources provisioned for each service at every cloud infrastructure.

Figure 1 shows the overhead evolution when increasing parallelism for the three services. ASF keeps growing exponentially up to the 320 parallel functions. However, the results are very stable, meaning that the behavior is actually implementation-related, and not a problem with resources. It is important to notice here that past 80 tasks, the overhead surpasses the fixed task time. Analyzing timestamps, we see that when task 86 is scheduled, number 1 has already completed, and the FaaS worker is reused. Therefore, the maximum concurrency achieved in ASF with this experiment is around 85.

We also see that ADF seems unstable, like in the previous paper. However, we do not see a clear exponential overhead. We see a lot of variability, which can be explained with restrictions or difficulties to obtain resources past the 20 concurrent functions. This makes us think that the increases in overhead are not strictly limited by the implementation, but by resource availability or provisioning. In this service, we also see functions that are not scheduled until some have already finished.

Since Apache Composer added support for parallel executions in January 2019, we also include it in Figure 1. As a first insight, the orchestration overhead with Composer grows more smoothly than with the other systems. This makes it the best solution for high degrees of parallelism (40–320).

Overall, out of the three, Composer has the best support for parallel task execution. However, we notice high variability in the results. We explore further this behavior in Figure 2, where we report the individual total execution times from many executions. As can be seen in this figure, variability increases with the amount of parallelism. Indeed, for 160 parallel tasks onward, variability exhibits bimodality, in the sense that some executions incur a low overhead of ≈ 4.5 seconds, while in another group of executions, it climbs up to 20 seconds. This suggests that, although all tasks are scheduled within the 20 seconds of the fixed task duration, limited resources lead to throttling the execution of some tasks until a bulk of them have finished. We completed this experiment by trying to identify the maximum degree of parallelism supported by Composer. Concretely, we observed a maximum degree of parallelism of 450. The concurrent execution of a greater number of functions resulted in system errors.

We should note here that, although Composer uses OpenWhisk's conductor actions (which makes it ST-safe), the implementation of parallel compositions is not integrated in the FaaS platform. To achieve this kind of orchestration, Composer needs a Redis key-value store to maintain intermediate results and synchronize the execution. Furthermore, each parallel composition spawns a function that remains blocked during the parallel stage, which breaks the *double billing* property.

Discussion All platforms offering orchestration of parallel tasks incur in some overheads to schedule and maintain the fault-tolerance capability of the execution. However, these overheads vary from system to system. To wit, ASF is very stable, but the penalty increases greatly with high parallelism. This makes it the "go to" for simple workflows with few parallel tasks, but not a good solution for *fork-join*-like workloads. ADF scales better, but its high variability becomes a problem with high degrees of parallelism. Composer is the best solution available for *fork-join*-like compositions. Out of the three, it is the only approach to achieve high levels of concurrency with suitable performance, while its variability could be mitigated with better resources.

4 Analysis of the internal architecture of Apache OpenWhisk and Composer

We have seen that production orchestration services have problems to offer parallel task compositions. However, Composer is the closest to a fitting solution. Fortunately, both, OpenWhisk and Composer, are open-source projects, and we dedicate this section to analyze their architecture and mechanisms to orchestrate function compositions.[2] We identify the parts in their design that thwart an elastic, reactive management of compositions.

4.1 OpenWhisk architecture

OpenWhisk is a FaaS system that enables users to code and deploy event handlers (Actions) to the cloud, and register them to respond to certain events. It is based on the following main components:

- *Controller:* Its main task is to handle user requests. In particular, the *controller* performs the authentication and authorization of every request. It can be viewed as the "orchestrator" of the system, since it decides the path that each request will eventually take.
- *Invoker:* It is responsible for processing Action invocations. Each *invoker* manages several Docker containers, whose purpose is to isolate the invocations from each other and thus, support multi-tenancy.

[2] OpenWhisk and Composer are both Apache projects initiated by IBM and available on GitHub [8, 9]. The data from Section 3 corresponds to IBM Cloud, and their deployment might differ from the architecture we detail.

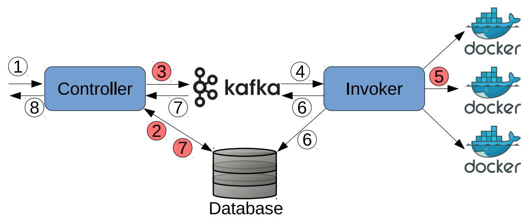

Figure 3. OpenWhisk life cycle.

- *Apache Kafka:* Its only function is to allow logged communication between the *controller* and the *invokers*. The rationale behind its usage is to tolerate heavy loads and system crashes. Since there exists the possibility that there are not enough resources to run an Action immediately, Kafka acts as a "*buffer*" where to hold invocations until they are ready to be executed.
- *Database (CouchDB):* It stores the source code, logs, results and metadata of the Actions, as well as authentication information.

4.2 OpenWhisk simple life cycle

Fig. 3 depicts an overview of the Apache OpenWhisk life cycle (red background circles indicate synchronous requests): (1) The user makes an invocation request to the controller via a REST API call. (2) The controller checks the database for authentication and authorization and obtains the source code to execute. (3) To get the Action invoked, the controller choses an invoker and publishes the invocation message to its Kafka topic. The user can perform this invocation either *asynchronous* or *synchronously*. (4) The chosen invoker receives the invocation message from its topic. (5) Then, it selects the *appropriate* Docker container, injects the code, and runs it. The invoker makes the decision of either reusing an existing "hot" container, starting a paused "warm" container, or launching a new "cold" container. This step can be synchronous. (6) The invoker forwards the response back from the container to the controller through Kafka. At the same time, it asynchronously stores the result to the database for fault tolerance. (7) The waiting controller receives the result either via Kafka (faster) or by polling the database (safer, but slower) in the event of a Kafka failure. (8) The controller replies with the result back to the user in case of synchronous invocation.

4.3 Composer overview

Composer is a programming model for the composition of OpenWhisk Actions [9]. It *programmatically* allows the creation of compositions and workflows, based on sequences, conditions and parallels via a Node.js API. Composer is made of three main parts:

- *Client:* It offers a rich API to end users that allows to express a variety of compositions and deploy them to Apache OpenWhisk. Importantly, the client is not involved at all in the execution of the composition, which happens entirely on OpenWhisk.

- *Apache OpenWhisk conductor actions [4]:* This *ad-hoc* code was designed to give OpenWhisk's *controller* a native support for compositions. Particularly, conductor actions manage the series of user-defined Actions to invoke at runtime as a state machine.
- *Secondary actions:* These intermediary functions are responsible for the transitions between the execution of the user-defined Actions. After each Action runs, a secondary action is invoked, which determines the next Action to be run in the workflow.

4.4 Composer simple life cycle

Fig. 4 shows an overview of the Composer logic life cycle. To simplify, we explain the procedure of a composition with a single user-defined Action. This diagram intentionally omits details covered in Fig. 3 (e.g., Apache Kafka).

The life-cycle is the following: (1) The user invokes the composition. (2) The controller checks the database to obtain the secondary action code for the composition. (3) The controller publishes the invocation message. The invoker, in turn, receives it, injects the secondary action and (4) the chosen Docker container runs it. In particular, it runs the composition's initialization and returns the next user Action to run. (5) The invoker forwards the result to the controller, which uses it to proceed to the next Action. (6) The controller fetches the user Action code from the database and (7) publishes the invocation message. The invoker, in turn, receives it, injects the code and (8) the chosen Docker container runs the actual user Action and returns the result. (9) The invoker forwards the result to the controller, which maps the output of the current invocation to the parameters of the next activation of the secondary action. (10) The controller checks the database again for the secondary action code and (11) publishes the invocation message. The invoker, in turn, receives it, injects the code and (12) the chosen Docker container runs the secondary action, which uses the output of the last user Action to determine the next one to run (if any) and returns it. (13) The invoker forwards the result to the controller, which analyzes the next Action to run. (14) In this case, as this is the last step and the composition was called synchronously, the controller returns the result to the user.

4.5 Composer parallel execution life cycle

To orchestrate compositions, Composer takes advantage from the integration of conductor actions [4] into the OpenWhisk core. However, this integrated logic only works for sequences with conditions. Simply put, *there is no support for running user-defined Actions in parallel in a composition.* An early version of Composer enables this feature with an external user-provided Redis instance.

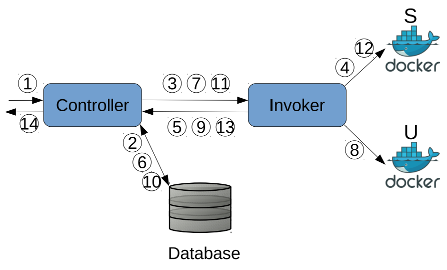

Figure 4. Composer life cycle.

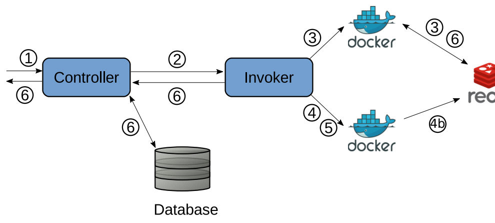

Figure 5. Parallel life cycle in Composer.

The life-cycle of this type of composition is depicted in Figure 5: (1) A Redis instance must be provided by the user and be accessible for Composer and OpenWhisk. (2) To run N tasks in parallel, Composer deploys a main secondary action and exactly N compositions. Each composition is made of two secondary actions that wrap the current user-defined Action as explained in the previous subsection. (3) The main action runs and blocks by means of a blocking pop call to an empty Redis list. (4) The first of the two secondary actions runs and initializes its corresponding composition. (5) Once this secondary action terminates, the actual user logic is run. Since there are N independent compositions, the user code is concurrently run N times. (4b) After the user code is executed, the last secondary action in each composition independently takes over and decrements an *atomic* Redis counter. The last action to update the counter also unlocks the main action by pushing an element to the empty Redis list. (6) The main action resumes the execution, retrieves the result of the composition and returns it to the user.

4.6 Discussion

The basic Composer architecture works well for the simple compositions it was initially designed (sequences and conditions). It also presents good performance numbers for parallel function orchestration compared to AWS Step Functions and Azure Durable Functions.

The innovative aspect of Composer is to delegate composition/orchestration to serverless functions. This brings major serverless benefits to orchestration like flexible scaling and pay per use, which cannot be found in dedicated workflow engines such as Airflow.

But we consider that the solution proposed for parallel executions is not reactive. First, it involves an external Redis instance for users to implement the fork-join model. But most importantly, a parallel stage needs a dedicated function blocked, idle, on a request to that external Redis store. Such a blocking call is enough to break the reactive definition, and it may affect the system's elasticity since it holds waiting resources. But even more, it may incur on double billing, since the blocked function is billed for the same time as the actions in the parallel stage.

Additionally, Composer is not designed for long-running parallel workflows that may be found in typical Big Data pipelines. The blocked function waiting on Redis will be effectively active all the time. Imagine the case where there must be a long idle wait between two steps in a composition, or a dependency on an external procedure or event (e.g., a user input or something to appear in a database). In those cases, Composer is likely to misuse resources and provoke interferences to the FaaS runtime. The effects of such drawback may be seen in the variability observed in Section 3.

We advocate for a more reactive solution to function orchestration. Instead of blocking the orchestrator function waiting for results in the Redis store, non-blocking event-based function sleep mechanisms should be available. A suspend primitive like the one presented in [10] could be used to build reactive function orchestrators.

5 Conclusions

Cloud providers are offering their services to create compositions of their FaaS platforms. AWS Step Functions, Azure Durable Functions, and OpenWhisk Composer, offer similar features but with very different approaches and programming models. These differences make them perform differently. In this paper we focused on the orchestration of serverless functions for parallel executions, and we have seen that none of the platforms solves it with suitable performance.

Composer is the best solution available as of today. Its idea of using serverless functions for the orchestration logic is indeed a good decision to achieve fitting elasticity. However, its implementation of parallel executions fails to fulfill a reactive scheme by adding a synchronous external connection. Furthermore, the system falls short on allowing long-running workflows (some steps would be billed for idle time and waste resources); a direct consequence of the orchestration system being so tightly integrated in the FaaS infrastructure itself.

After this study, we believe the next step towards elastic orchestration of serverless functions requires a reactive event-based design, avoiding double billing and blocking external invocations.

We advocate for a novel suspend primitive for functions that may be triggered by custom events (including function termination events). The ability to suspend a function could then enable the creation of third-party orchestrators not directly tied to the underlying FaaS platform. This approach obviously requires runtime support in the FaaS system and rich trigger-based mechanisms for activating functions. But such novel primitives could enable the creation of decoupled orchestration services based on serverless functions. This would be extremely useful for Big Data pipelines involving parallel workloads and long-running compositions.

Acknowledgments

This work has been partially supported by the EU Horizon 2020 programme under grant agreement No 825184 and by the Spanish Government through project TIN2016-77836-C2-1-R. Marc Sánchez-Artigas is a Serra Húnter Fellow.

References

[1] 2013. Camunda Open Source Workflow Platform. https://camunda.com/.

[2] 2014. Reactive Manifesto. https://www.reactivemanifesto.org/.

[3] 2019. Amazon AWS Serverless Definition. https://aws.amazon.com/serverless/.

[4] 2019. Apache Conductor Actions. https://github.com/apache/openwhisk/blob/master/docs/conductors.md.

[5] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. 89–103.

[6] Opher Etzion, Peter Niblett, and David C Luckham. 2011. *Event processing in action.* Manning Greenwich.

[7] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation NSDI 17)*. 363–376.

[8] Apache Software Foundation. 2019. Apache OpenWhisk. https://github.com/apache/openwhisk.

[9] Apache Software Foundation. 2019. Apache OpenWhisk Composer. https://github.com/apache/openwhisk-composer.

[10] Pedro García-López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona-Pons, Álvaro Ruiz, and David Arroyo-Pinto. 2018. Comparison of FaaS Orchestration Systems. In *WoSC4 - UCC Companion*. 148–153.

[11] Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. 2012. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 4:1–4:10.

[12] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. ACM, New York, NY, USA, 445–451.

[13] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. *CoRR* abs/1803.06354 (2018). http://arxiv.org/abs/1803.06354

[14] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. in press. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems* (in press).

[15] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the 19th International Middleware Conference Industry (Middleware '18)*. 1–8.