Exploring the cost and performance benefits of AWS Step Functions using a data processing pipeline

Mathew, Anil; Andrikopoulos, Vasilios; Blaauw, Frank J.

Published in:
2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21), December 6–9, 2021, Leicester, United Kingdom

DOI:
10.1145/3468737.3494084

Exploring the cost and performance benefits of AWS Step Functions using a data processing pipeline

Anil Mathew
a.palayiparambil.mathew@student.rug.nl
University of Groningen
Groningen, The Netherlands

Vasilios Andrikopoulos
v.andrikopoulos@rug.nl
University of Groningen
Groningen, The Netherlands

Frank J. Blaauw
f.j.blaauw@researchable.nl
Researchable B.V.
Groningen, The Netherlands

Abstract

In traditional cloud computing, dedicated hardware is substituted by dynamically allocated, utility-oriented resources such as virtualized servers. While cloud services are following the pay-as-you-go pricing model, resources are billed based on instance allocation and not on the actual usage, leading the customers to be charged needlessly. In serverless computing, as exemplified by the Function-as-a-Service (FaaS) model where functions are the basic resources, functions are typically not allocated or charged until invoked or triggered. Functions are not applications, however, and to build compelling serverless applications they frequently need to be orchestrated with some kind of application logic. A major issue emerging by the use of orchestration is that it complicates further the already complex billing model used by FaaS providers, which in combination with the lack of granular billing and execution details offered by the providers makes the development and evaluation of serverless applications challenging.

Towards shedding some light into this matter, in this work we extensively evaluate the state-of-the-art function orchestrator AWS Step Functions (ASF) with respect to its performance and cost. For this purpose we conduct a series of experiments using a serverless data processing pipeline application developed as both ASF Standard and Express workflows. Our results show that Step Functions using Express workflows are economical when running short-lived tasks with many state transitions. In contrast, Standard workflows are better suited for long-running tasks, offering in addition detailed debugging and logging information. However, even if the behavior of the orchestrated AWS Lambda functions influences both types of workflows, Step Functions realized as Express workflows get impacted the most by the phenomena affecting Lambda functions.

CCS Concepts

• **Software and its engineering → Software performance**; **Cloud computing**.

Keywords

Serverless, AWS Step Functions, AWS Lambda, Function-as-a-Service (FaaS), Serverless Cost, Serverless Performance

UCC'21, December 6–9, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-8564-0/21/12.
https://doi.org/10.1145/3468737.3494084

ACM Reference Format:
Anil Mathew, Vasilios Andrikopoulos, and Frank J. Blaauw. 2021. Exploring the cost and performance benefits of AWS Step Functions using a data processing pipeline. In *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3468737.3494084

1 Introduction

Serverless computing is a new and exciting form of utility computing that allows users to run event-driven and granularly billed applications without addressing operational logic [Baldini et al. 2017a]. According to a recent market report[1], the market size of serverless computing will reach USD 7.72 billion by the end of 2021 and grow further, making it a compelling paradigm for the deployment of cloud applications. Function-as-a-Service (FaaS) is a form of serverless computing that allows executing code in response to events without the complex infrastructure associated with building and launching microservices applications [Grogan et al. 2020].

Amazon Web Services (AWS), the pioneer in FaaS offerings, launched AWS Lambda in 2014allowing to run functions called *Lambda functions* or simply *Lambdas* for short, natively written in a set of commonly used programming languages like Node.js, Java, C#, Python, .NET, and Go. A single Lambda cannot be independently used to construct complex applications, and a typical complex enterprise serverless application contains multiple Lambdas. As Lambdas are stateless with no affinity to the underlying and overall infrastructure, they lack adequate integration and coordination mechanisms, making orchestration cumbersome [López et al. 2018]. *Orchestration* is a way to construct serverless applications with multiple steps and complex logic programmatically or graphically [Baldini et al. 2017b] by combining them into *workflows*. AWS supports workflow-based orchestrations of distributed systems with the help of AWS Step Functions (ASF)[2], which allows designing workflows as sets of states.

As ASF orchestrates Lambdas into Step Functions, all the advantages of the former [Baird et al. 2017] are available for it, like for example scalability [Albuquerque Jr et al. 2017], fine-grained billing [Van Eyk et al. 2017], and low operational costs and complexity [Kuhlenkamp et al. 2020]. At the same time, ASF-based orchestration inherits and amplifies existing issues with using Lambda. Serverless, for example, comes with the promise of cost efficiency but according to [Eivy and Weinman 2017], it has surprisingly complicated economics. To quote the authors: "The devil is in the

[1] Source: https://www.marketsandmarkets.com/Market-Reports/function-as-a-service-market-127202409.html
[2] https://aws.amazon.com/step-functions/

details, and the economic benefits of serverless computing heavily depend on the execution behavior and volumes of the application workloads". Works like [Baldini et al. 2017a; Van Eyk et al. 2018] highlight the lack of an actual cost and performance benchmark, making it difficult for customers to adopt a serverless model. The situation is even more challenging for applications realized using serverless function orchestrators such as ASF since the execution cost compounds the Lambda cost along with the charges incurred for managing the orchestration itself. The lack of a cost monitoring or estimating tool that provides fine granular billing and execution details hinders the understanding of the involved billing model even further, as the case remains to be for FaaS solutions in general [Leitner et al. 2019]. As a result, application developers are at a loss on how to design their serverless applications cost-efficiently.

Towards addressing this issue, this paper investigates how various parameters impact the cost and performance of a serverless application realized using Step Functions orchestrating Lambdas. To this effect, we conduct an *experiment-driven quantitative analysis* on how a number of identified parameters can impact a sample application in the form a data processing pipeline [Densmore 2021]. We show that in many aspects, realizing a serverless application based on ASF efficiently can be challenging and/or counter-intuitive. To the extent of our knowledge, and at the time of writing this, there are no other empirical studies that investigate the effect of various parameters on ASF with respect to cost and performance. This work therefore has implications for both the state of the art and for practitioners.

The remainder of this paper is organized as follows. We first present background details on the involved technologies in Section 2. In Section 3, we present an experimental design for evaluating ASF in terms of cost and performance, followed by the outcomes of the experiments in Section 4. Based on these observations, in Section 5, we discuss the lessons that we learned and that we think are relevant for practitioners that aim at using ASF as their function orchestrator when developing an application, while sketching our future work on the topic. Next, we present related work in Section 6. Finally, Section 7 concludes this work with a short summary.

2 Background

AWS, with the debut of AWS Lambda in 2014, is the first large public cloud vendor to offer FaaS. AWS is the leader in the serverless space, with 80% of the serverless use cases choosing them as their deployment platform [Eismann et al. 2020b]. When looking into their pricing model, FaaS solutions in general feature a GB-second billing model depending on the allocated memory size and execution duration. Here the customer is also charged for each invocation and its execution duration typically in 100 ms increments. As of June 2021, for example, AWS Lambda is charging $0.20 per 1M requests and $0.0000166667 for every GB-second. This price varies based on the region where AWS Lambda is hosted[3]. In addition, in December 2020, AWS revised its Lambda pricing with precision at the level of 100 ms to 1 ms. This change decreases the price for most Lambdas, more so for short-duration functions.

Serverless solutions inspire programmers to decompose large monolith applications into fine granular functions using the process of FaaSification [Spillner 2017; Spillner and Dorodko 2017], making them more manageable to understand and reusable. FaaS offerings are biased towards simple functions that only run for a short while, use limited CPU and memory, and process relatively small amounts of data. However, functions alone are not applications but merely tasks, and to build compelling FaaS applications, the functions need to be orchestrated. [Van Eyk et al. 2017; Yan et al. 2016] envisioned this complex orchestration in the form of *workflows*. In this regard, the serverless workflow orchestration offerings by the "Big Four" public vendors i.e. AWS Step Functions (debuting on December 2016), Azure Durable Functions (June 2017), IBM Composer (October 2017), and Google Workflows (August 2020), are still relatively young technologies, with ASF being the most mature of them [López et al. 2018]. The survey by [Bocci et al. 2021] provides an overview of various research efforts focusing on these offerings.

ASF in particular provides its users with a visual interface for debugging and monitoring, and allows workflow design by defining states and transitions as finite state machines written in Amazon States Language, a custom JSON-based Domain Specific Language (DSL). The states are either a *task* or a language construct that influences the *flow* between states. AWS Lambda and other AWS services can be incorporated in ASF to build business-critical applications. When defining an ASF model, there is a mandatory start and end state, and every state must declare its successor, being a successful, transition, failed, or end state. Step Functions support two workflow types. *Standard* workflows can be used for long-running, durable, and auditable workflows, while *Express* workflows are suitable for high-volume, event-processing workloads[4]. The former have at-most-once model workflow execution semantics suitable for non-idempotent tasks and can run for up to one year. In contrast, the latter has at-least-once model semantics suitable for idempotent tasks and can run for up to five minutes. Furthermore, Express workflows support massive concurrency for workflow execution and nearly unlimited state transitions compared to Standard. Because Express executions are capped with a limit of 5 minutes, however, Express does not support any integrations with other services or implementing patterns that require the state machine to wait. Irrespective of their type, ASF workflow executions are affected by *cold starts* of their orchestrated Lambdas. This means that when a workflow is initiated for the very first time, all associated Lambdas will start with some provisioning delay called a cold start, but any subsequent workflow invocation will reuse the Lambdas and experience a warm start [Manner et al. 2018].

The pricing model varies between the two ASF workflow types. As of June 2021, executing a Standard workflow costs $0.025 per 1,000 state transitions. Since there is a minimum of three transitions for any workflow (one for the workflow to reach the start activity, one to reach its final one, and at least one intermediate state), customers need to pay at least $0.000075 for each successful Standard workflow execution, and $0.00005 for each failed execution. Express

[3] https://aws.amazon.com/lambda/pricing/

[4] https://docs.aws.amazon.com/step-functions/latest/dg/concepts-standard-vs-express.html

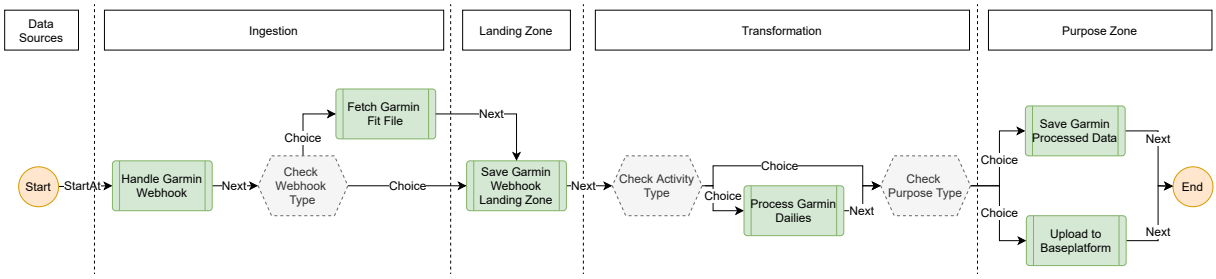

Figure 1: Garmin serverless data processing pipeline using ASF

workflows on the other hand cost the customers $1.00 per 1 million requests and the duration is priced based on the allocated memory:

- $0.00001667 per GB-second ($0.0600 per GB-hour) for the first 1,000 GB-hours
- $0.00000833 per GB-second ($0.0300 per GB hour) for the next 4,000 GB-hours
- $0.00000456 per GB-second ($0.01642 per GB-hour) beyond that.

3 Experiment Design

3.1 Application Under Test

To evaluate ASF, we will be adopting a *serverless data processing pipeline* as our test application. A *data processing pipeline* [Densmore 2021] is a type of application that enables a smooth, automated flow of data that processes data through a sequence of connected processing steps. Serverless computing supports new possibilities in designing pipelines that realize high scalability, elasticity, and performance while minimizing the cost and development effort. [Pogiatzis and Samakovitis 2021], for example, showed that a data processing pipeline constructed entirely using serverless technologies facilitated a cost effective and practical solution for sparse event processing.

The data processing pipeline to be used is one for Garmin[5] activity tracking data that was developed as part of an exploratory case study for Researchable B.V.[6] The ASF modeled application supports the processing of two possible Garmin data types, namely *Garmin Daily Summaries* and *Garmin Activity Files*[7]. The former offers a high-level view of the user's entire day, and the latter provides actual files recorded by the wearable as part of a fitness activity, including GPS coordinates, all recorded sensor and any product-specific data. Figure 1 visualizes the high-level architecture for the Garmin serverless data pipeline divided into the following activity *zones*, as advocated by [Pogiatzis and Samakovitis 2021]:

1. `Data Source Zone`: This zone consists of the integrations that the application accepts. For this work this entails accepting webhook requests from Garmin Connect.
2. `Ingestion Zone`: This zone takes care of handling the webhook requests that can either be PING/PUSH requests for Daily Summaries or Activity Files; in the latter case the file is fetched from Garmin using the provided callback URL.
3. `Landing Zone`: Data fetched/received from Garmin are stored in this zone before any processing.
4. `Transformation`: Garmin data are modified using transformation logic.
5. `Purpose Zone` : The final zone either saves the data to a serverless database or pushes the data to Baseplatform (a proprietary backend system used by Researchable).

When the serverless data processing pipeline receives either type of Garmin data, it will require a total of nine state transitions for successful completion, including two mandatory states, i.e., start and stop. This path also includes three *Choice* states and four *Task* states[8]. A Choice state adds branching logic to a state machine that is necessary to take the appropriate path based on the data ingested. The Lambda task state is responsible for invoking AWS Lambdas within ASFs. In the following we will be focusing on running all our experiments using *Garmin Daily Summaries* only in order to have comparable results throughout all the experiments. All Lambdas have been implemented and deployed using the Node.js 14 runtime. The source code for the pipeline, together with the material required to reproduce the experiments discussed below are available on GitHub[9].

3.2 Experimental Setup

As with similar works for investigating AWS Lambda (see Section 6), our experiments are designed to investigate the performance (in terms of execution time) and cost characteristics of ASF. For this purpose, we first need to identify the parameters that are more likely to influence these two metrics.

More specifically, and as discussed in the previous section, ASF has two workflow variants, and both are priced differently. Standard workflows are charged by the total number of state transitions across all state machines, including retries, and as such the number of state transitions should be taken into account for our experiments. In contrast, Express workflows are charged only based on the number of workflow requests and their duration. The duration between state transitions has an obvious effect on performance in both cases which also needs to be investigated.

Furthermore, ASF is used to orchestrate Lambdas, and the impacts of factors affecting Lambdas like the startup type and allocated memory [Back and Andrikopoulos 2018; Cordingly et al. 2020; Grogan et al. 2020; Manner et al. 2018] are also expected to propagate

[5]https://connect.garmin.com
[6]https://researchable.nl
[7]https://developer.garmin.com/fit
[8]https://docs.aws.amazon.com/step-functions/latest/dg/concepts-states.html
[9]https://github.com/anil-rug/cost_performance_asf_data_processing_pipeline

Table 1: Experimental Setup. Startup type refers to enforced cold starts (Cold) for all Lambdas versus regularly occurring ones (Warm). Runs per configuration indicates number of executions per combination of the other configuration options.

#	Parameters		Workflow Type	Lambdas Configuration		Runs per configuration
				Startup type	Memory size (MB)	
1	Lambdas	Memory Size	Standard & Express	Cold & Warm	128, 256 & 512	90
2		Startup Type				
3	Workflow	State Transition Duration	Standard & Express	Cold & Warm	128, 256 & 512	100
4		Number of States	Standard & Express	Cold & Warm	256	30
5		Execution Duration	Standard & Express	Warm	256	10
6	Load	Payload	Standard	Warm	256	10
7		Concurrent Load	Standard	Cold & Warm	256	30/100

to the function orchestrator. Additionally, factors like payload and concurrency need to be taken into consideration too, as these parameters can vary when executing the workflow and might impact performance and cost. Table 1 summarizes the seven parameters to consider in the experiments discussed below, organized in three distinct categories: parameters related to the *effect of lambdas on ASF execution* (memory size and startup type), related to the *workflow itself* (state transition duration, number of states, total execution duration), and finally related to the *input load of the application* (payload size and concurrent load requests).

For each of these parameters under investigation we take into account the following configuration options, as shown in Table 1:

- Workflow Type: ASF has two workflow execution types, namely Standard and Express, as discussed above. In most experiments both workflow types are to be used.
- Lambdas Configuration
  - *Startup Type*: As discussed in the previous section, ASF workflows are affected by the cold starts of the Lambdas they orchestrate. As Lambdas retain their execution environment for a non-deterministic period, a guaranteed cold start has to be enforced for each consecutive run by updating all the Lambdas involved in the workflow before the ASF workflow execution is invoked. The former case is indicated by "Warm" and the latter by "Cold" in the table.
  - *Memory Size*: We have restricted the allocated memory size for orchestrated Lambdas to 128, 256, and 512 MB for our evaluation, as these sizes are more than sufficient to serve the needs of our data processing pipeline.
- *Execution Runs*: Depending on the experiment, and in order to control for performance variability, we execute multiple runs and aggregate the results in our findings. Table 1 shows the number of runs per combination of the other parameters. In experimental setup #1, for example, there are 90 runs for Type set to Standard, Startup type to Cold, and Memory size to 128 MB; then 90 more runs for 256 MB and so on, for a total of $2 \times 90 \times 2 \times 3 = 1080$ executions.

To conduct our experiments, the cost and performance for all workflow executions needs to be inspected. However, using native AWS cost monitoring services like AWS Billing[10] and Cost Explorer[11] is not sufficient for different reasons. AWS Billing does not offer granular billing details for a single execution but aggregates it monthly/daily/hourly, and AWS Cost Explorer refreshes cost data only daily. This delayed and non-granular billing fails to provide low-level cost/performance information per workflow execution. Instead, for this purpose we built a custom dashboard monitoring the duration and granular cost of workflow executions.

3.3 Experiments

The experiments to be conducted investigate the following:

3.3.1 Effect of Lambdas on ASF: For the first experiment we investigate the effect of allocated memory size and startup type to both workflow types. Experimental setups #1 and #2 in Table 1 are used for this purpose. Notice that when we discuss the warm startup type here and for the rest of the experiments, the results will also contain (some) cold starts since we are not "warming up" the ASF workflows prior to taking measurements; however, the cold startup type is ensured to consist exclusively by cold starts. We feel that comparing this average/expected versus the worst case scenario is a better foundation for understanding how ASF actually behaves in practice.

3.3.2 Workflow: In order to look into how the workflow itself affects the cost and performance of an ASF we design three experiments, focusing on a different parameter each time:

State Transition Duration: This experiment intends to evaluate the effects of intermediate transition or idle time between states on the overall cost and performance for Standard and Express workflow type executions. Experimental setup #3 in the table is used for this purpose. This experiment is similar to the previous one, but this time the logs from the workflows executions are to be used to distinguish between time spent in each state and on transitioning between them.

Number of States: For this experiment we iteratively re-model and re-implement the data processing pipeline by merging individual tasks so that the workflow consists of:

[10] https://docs.aws.amazon.com/account-billing

[11] https://aws.amazon.com/aws-cost-management/aws-cost-explorer

(1) 7 + 2 states: This is the original implementation specified in Section 3.1 and it comprises of 4 Lambda task States and 3 Choice States, plus the obligatory start and end states.
(2) 4 + 2 states: 3 Choice States were removed and the Lambda States remain the same.
(3) 3 + 2 states: combined "Handling of Garmin Webhook" and "Save Garmin Webhook to Landing Zone" activities into one; hence a total of 3 intermediate states.
(4) 2 + 2 states: combined "Handling of Garmin Webhook", "Save Garmin Webhook to Landing Zone," and "Process Garmin data" Lambda States, for a total of 2 intermediate states.
(5) 1 + 2 states: combined all Lambda States into one.

The experiment is executed 30 times for each of these variants of the workflow with all Lambdas involved being allocated 256 MB of memory, as shown in Table 1, Experimental setup #4.

Execution Duration: Experiment setup #5 in Table 1 shows the configuration used to evaluate the effect of overall workflow execution duration on ASF workflows. The involved Lambda tasks have been modified by adding progressively longer sleep times to them (0/7.5/15 s to each task) to induce latency, to a maximum of 5 minutes, since this is the cap of Express workflows (see Section 2).

3.3.3 Load: In these two experiments we investigate how ASFs behave when input load changes in two ways. One, by changing the amount of incoming data to be processed by the application, and two, by doing multiple concurrent requests:

Payload: In the previous experiments the same Daily Summary single entry of size 0.9 KB has been used as an input. For this experiment the same ASF workflow will be executed using Experimental setup #6 in the table 10 times in each case for a linearly growing payload size ranging from 0.9 KB (size of a single data entry in the payload) to ~200 KB, the maximum size imposed by ASF at the time of conducting our experiments (June 2021).

Concurrent Load: In this experiment we intend to study the effect of workflows being triggered concurrently in two sub-experiments, each structured to follow a two-step process (setup #7 in Table 1). The first step is to initiate multiple instances of the workflow concurrently so that we would encounter one or more cold starts. Then, for the second step, we invoke the subsequent concurrent executions instantly after the first step to analyze how the workflow executions handle the concurrent load. For the first sub-experiment we first run 10 workflows simultaneously, followed by 20 executions. For the second one, we equalize the load between the two steps and execute 50 workflows concurrently in each step.

4 Results

In the following we present our findings with respect to the experiments described in the previous section. Their implications are discussed in the following section.

4.1 Effect of Lambdas

Memory Size: Figures 2 and 3 depict the workflow execution duration and cost, respectively, using 128, 256, and 512 MB memory size and cold/warm starts. Based on these results, we observe that:

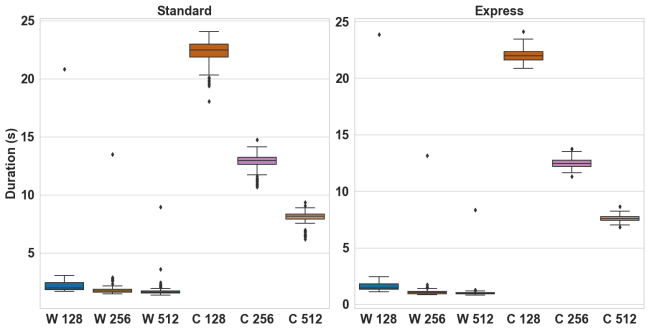

Figure 2: ASF execution duration vs allocated AWS Lambda memory for Standard & Express workflow executions (warm [W] and cold [C] starts)

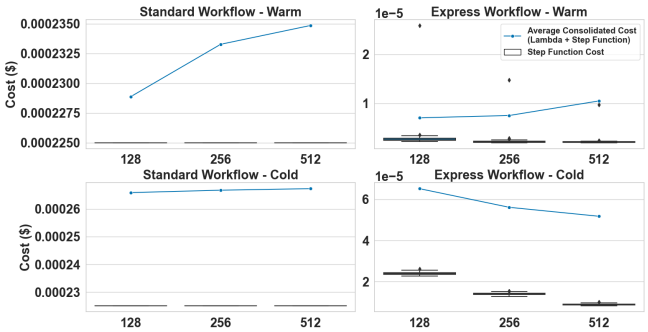

Figure 3: ASF execution cost vs Lambdas memory for Standard & Express workflow executions (warm/cold starts). Both consolidated (total) and Step Function-only costs are shown in the figures.

- With increased memory, the overall cost slightly rises by 1.35% (warm) and 0.38% (cold) on average across all memory configurations even if there is a decrease in the execution duration for Standard workflows. This is expected because the ASF state transition cost remains constant throughout all memory configurations and the Lambda cost is the only differentiating factor.
- When looking at the Express workflow, the trend shows that similarly the cost increases with more allocated memory for (mostly) warm startups. However, for cold startups the cost decreases with the increase in memory. The latter behavior is because of the drastic difference in execution duration with the change in memory configuration (22.155 s for 128 MB vs 7.635 s for 512 MB, on average).

Startup Type: Based on the same figures, but looking specifically at the startup types, we conclude that:

- The startup type of Lambdas has a significant impact on the execution duration and the overall cost of the execution. ASF executions that experience (mostly) warm starts execute on average 85.82% faster than executions that have been performed using only cold starts. This behavior is noticed for both the Standard and Express workflows.

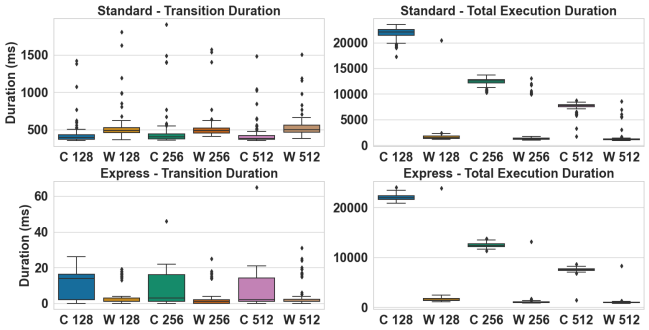

Figure 4: Transitions-only vs total duration for Standard & Express workflow executions using different Lambda memory sizes and experiencing warm [W] and cold [C] starts

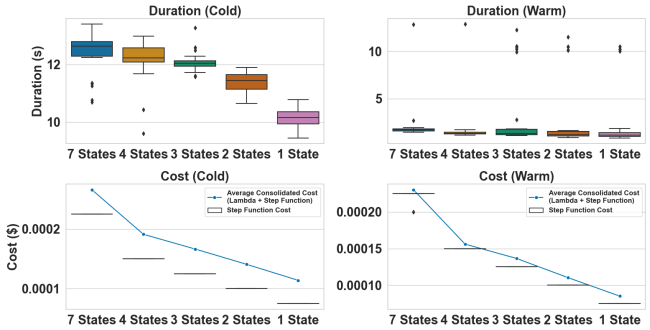

Figure 5: ASF execution duration & cost vs number of workflow states for Standard workflow executions; both the average consolidated cost (i.e. including Lambda costs) and the pure Step Function cost are plotted for the latter

- As Lambda costs are calculated based on the memory allocated for the Lambda and the duration of execution, cold starts result into a higher incurred cost overall.
- This cost difference between warm and cold starts is much more obvious for Express workflows compared to Standard workflows, as the AWS Lambda execution duration directly influences the cost of the former. In contrast, the ASF cost for the latter is basically not determined by the Lambdas execution duration. Thus an increase in Lambda execution duration caused by cold starts has a more severe effect on ASF Express workflows.

4.2 Workflow

State Transition duration: With respect to the effect of state transition duration on the workflow execution, the following can be concluded based on Figure 4:

- Express workflows have in general minimal state transition/idle time, spending less than 20 ms on average in state transition. On the other hand, for Standard workflows, the total state transition duration is on average between 350 ms to 670 ms, which is much higher when compared to Express, for an average of 3.27% (cold) and 33.1% (warm) of the total execution time spent in state transition. This considerable idle time can be attributed to the fact that Standard workflows store state information while Express workflows do not offer any state transition details (see next section for more).
- The absence of storing state transition information behavior makes Express workflow overall slightly faster than Standard workflow executions, everything else being the same.

Number of States: Figures 5 and 6 plot the experimental results for decreasing the amount of states in the same application for both ASF workflow types. Based on them we conclude that:

- Choice states do not have a severe impact on the application's performance, as indicated by the transition from 7 to 4 states in the figure. The slight difference in performance (420 ms on average) is mainly due to the transition time for Standard workflows. For Express workflows, this improvement is negligible (less than 10 ms) as this workflow does not store state information transition from one state to another.

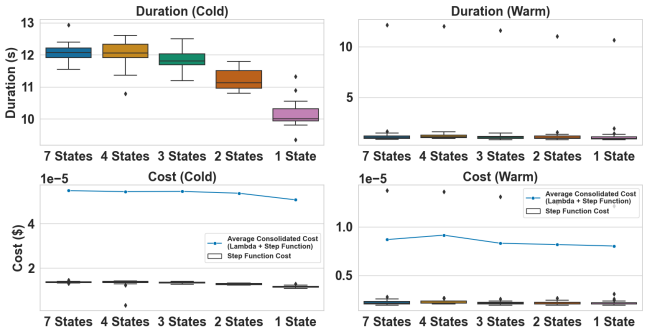

Figure 6: ASF execution duration & cost vs number of workflow states for Express workflow executions (as in the previous figure)

- Reducing states has a significant positive impact on both performance and cost when executing using Standard workflows with a cold start: decreasing the number of states from 7 to 1, for example, resulted in an improvement of 19.62% and 56.87%, respectively. In contrast, when the execution experiences mostly warm starts, the performance is alike for all cases; however, overall the cost declines with a decrease in states.
- Express workflows experience better performance but trivial cost benefits with a decrease in states for executions with a cold start. On the other hand, executing the workflow in either a distributed or monolith pattern, i.e. with too many or a single state using the Express workflow has negligible monetary and performance gains for warm starts.
- For the executions with a cold start, decreasing the Lambda states improves overall performance as the total latency incurred for cold start initialization by AWS Lambda drops.

Execution Duration: Figure 7 shows the results of the experiment to understand the effect of execution duration on Standard and Express costs. For Standard workflows, and since Standard workflow execution cost depends on step transitions, we observe that the cost does not vary with the increase in execution duration. For Express workflows, however, the step function cost keeps increasing

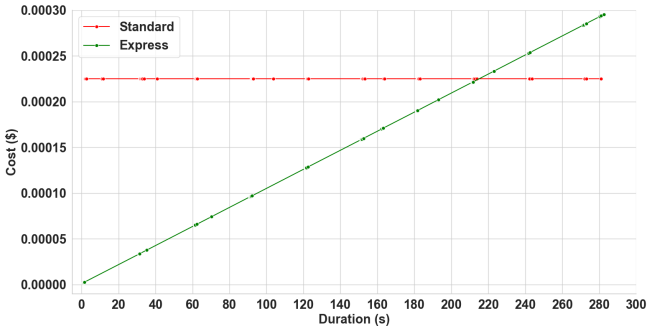

Figure 7: ASF execution duration vs total cost plot for Standard & Express workflow executions

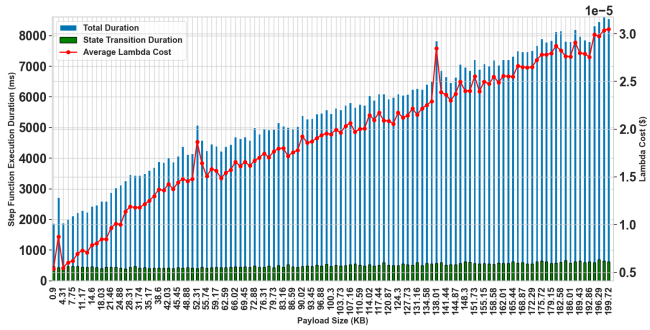

Figure 8: Payload size vs ASF execution duration (left y-axis) vs AWS Lambda cost (right y-axis) for Standard workflow executions

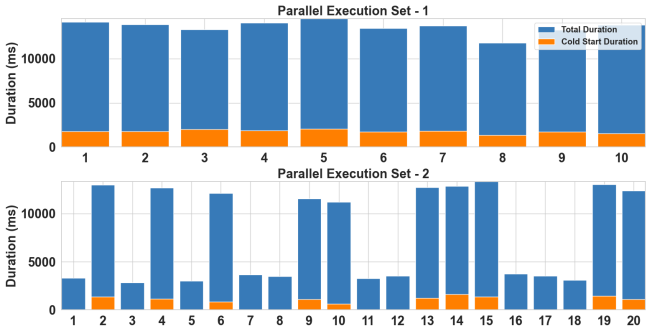

Figure 9: Sub-experiment 1 (Standard workflow executions): 10 + 20 **concurrent executions**

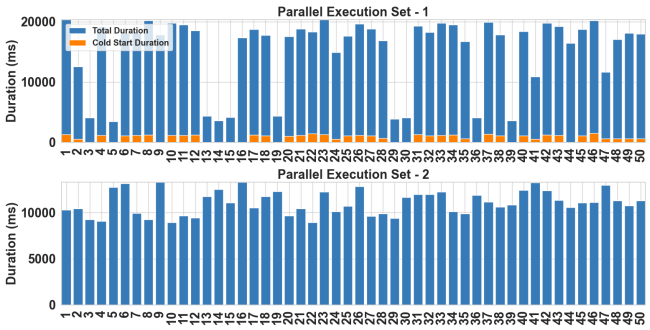

Figure 10: Sub-experiment 2 (Standard workflow executions): 50 + 50 **concurrent executions**

linearly with an increase in step execution duration, to the point that it overtakes the cost for the execution of the same workflow in Standard for longer running Step Functions. For the Garmin Daily Summaries workflow, this tipping point is at around 215 s total execution time, after which it becomes more cost efficient to execute the same workflow in Standard, everything else being the same.

4.3 Load

Payload: Figure 8 has been plotted based on the experiment results to understand the effect of payload size on workflow execution. Below are the conclusions that can be drawn from the graph:

- With the increase in payload size, both overall ASF execution duration and AWS Lambda costs increase proportionally.
- The presence of outliers (spikes in the figure) indicates that the relative AWS Lambda costs are much lower even with higher tasks duration. On further analysis, it is observed that roughly 50% of the time is actually used for state transition. This behavior can be corroborated using the Effect of State Transition Duration Experiment (Section 4.2), where it is shown that Standard executions incorporate higher state transition time.

Concurrent Load: Figures 9 and 10 plot the results of our experiments to understand the effect of concurrent requests on workflow execution. As expected, the performed experiments show that ASF can be executed concurrently and at scale, and that the effects of AWS Lambda are propagated to the ASF execution. At the same time, and with more concurrent workflow executions, several runs take longer than average to finish as spinning up new Lambda instances to serve all the requests introduces latency and thus negatively impacts performance. With respect to the two sub-experiments using Experimental setup #7 in the table:

- During the initial step of the first sub-experiment, all the workflows underwent some cold start duration. The cold start can be seen in the graph as the orange bars. In the subsequent executions, where 20 workflows were triggered concurrently, 10 workflows ran without any cold start duration, i.e. reused the previous warm Lambdas, with the remaining 10 showing cold starts of varying duration, indicating that new Lambdas needed to be initiated. The executions that did not have any cold start associated with them executed much faster (73.191% on average) than those that had, making them more performant for both workflows and economical for Express workflow.
- During the initial execution set in the second sub-experiment (50 workflows), only some Lambdas experienced a cold start (orange bars in the figure). The chart shows that the executions that experienced a cold start took (much) more time than the executions that did not have a cold start — 14.041 s on average to be precise. Despite the ASF being triggered concurrently, some of the executions from the first 50 workflows do not have a cold start duration because they reused existing warmed-up Lambdas,

despite being invoked concurrently in our load. Furthermore, the subsequent execution set of the next 50 workflows did not have any cold start at all, showing that multiple instances of the Lambdas were warm and ready for reuse without starting a new Lambda instance. Surprisingly, the execution duration for the second set was considerably higher when compared to both the first set of 50 workflows and the previous sub-experiment.

5 Discussion & Future Work

5.1 Implications

In this experimental evaluation, we performed a total of 7 experiments to understand the effects of various parameters on ASF with the support of an application realized as a Step Function. We will now discuss the implications of these results for practitioners and researchers when considering adopting ASF for FaaS orchestration.

First, when examining the effects of AWS Lambda memory allocation on ASF, we observed that *cold start latency decreases with an increase in Lambda memory size, thus resulting in ASF executions with better performance.* This finding complied with the observations from e.g. [Wang et al. 2018] and implies that AWS allocates CPU power proportionally to the defined memory size, suggesting that the more the CPU power, the environment boot-up becomes faster. Contrary to expectations, however, and even if the increase in memory size reduces execution duration, the *workflow execution cost appears to be inversely proportional and actually increasing for warm starts in both Standard and Express workflows.* This result may be explained by the fact that the execution duration for warm Lambdas varies insignificantly, and the cost of higher memory Lambdas surpasses the slight decrease in execution duration. A similar trend is also seen for the Standard workflow with a cold start, but the disparity between the different memory sizes is not so drastic. On the contrary, Express workflow executions with cold start show a downwards trend in cost with increased memory, signifying that the workflow execution duration significantly impacts ASF cost.

Our experiment for evaluating the effects of Lambda startup type confirms that cold and warm starts have a substantial impact on workflow execution. It is seen however that *cold starts impact the cost for Express workflows considerably more than for Standard workflows as the execution duration directly controls the Express workflows execution cost.* So, if an application is expected to have a considerable amount of cold starts, due to e.g. having intermittent load, Express workflows would not be the recommended.

Another important finding is that *ASF Express workflows have significantly less state transition times when compared to Standard workflow execution.* A possible explanation for this might be that Express workflows persist state transitions in memory while the Standard workflow stores the state details to a disk. The persistence of states on disk makes Standard workflow more fault-tolerant, and if a state fails, then that particular state needs to be restarted. However, for the Express workflow, the entire state machine needs to be restarted in case of failure. This will incur additional unexpected costs as reported also by practitioners [Leitner et al. 2019]. Another possible explanation for this latency is that Standard workflows store each state's input and output data using the execution history and provide a visual debugging interface. Express workflows do not have additional overhead as logs are confined to Amazon CloudWatch[12], and logs are stored using a nonblocking behavior.

When looking at the cost viewpoint on the number of states, Standard workflows are heavily influenced by it as pricing is based on state transitions. *With the decrease in the number of states, the Standard workflow cost decreases proportionally.* If the workflow is executed using only cold starts, the workflow with the least number of transitions is more performant and runs with the least duration. On the contrary, if the execution encounters mostly warm starts, the execution durations are similar. At the same time, *most of the conducted experiments hugely advocate using Express workflows for short-lived tasks.* Tasks that require a prolonged duration however show that Express workflows can be pricier than Standard workflows, which is influenced mostly by the number of state transitions. A Standard workflow with the least possible number of states seems therefore a viable option if a user wants to take advantage of the detailed execution history and visual console for debugging. However, this reduction of states impacts flexibility and granularity and increases coding effort, reducing developer happiness. On the other hand, if the user adopts an Express workflow, it is not necessary to worry about the number/type of states as they can be used judiciously. However, this happens at the expense of losing access to the rich information available for Standard executions.

The most obvious finding to emerge from the investigation is that *as the payload size grows, so will the workflow execution duration.* This increase in duration is because the Lambdas need to perform more complicated processing and handle more extensive data. Another reason for this increased duration can be attributed to the larger payloads being transitioned between states as the state transition time gradually increases with a larger payload size. Identifying the proportional impact of these two performance degradation points is however left for future work. *Concurrent execution of ASF workflows impacts both performance and cost*, mainly due to the cold and warm starts of AWS Lambdas. If Lambdas are experiencing concurrent invocations, AWS will initialize them simultaneously as expected. Our experiment shows, however, that a sudden spike in requests in a short period negatively impacts the overall performance. Moreover, the number of concurrent requests can be limited by the type of workflow, as discussed in Section 2. If the application expects massive ingestion of data simultaneously and results at near real-time latency, then ASF adopters must look into managing the concurrency for AWS Lambdas by investigating the reserved or provisioned concurrency options[13] instead.

One final point that emerges by looking across the various experiments is that the two ASF workflow types behave effectively as two different systems instead of two variants of the same orchestration system. Especially the differences in state transition management and their attached pricing model hint heavily at different implementations. Verifying this supposition e.g. experimentally is however left as the task for further research.

5.2 Limitations

Our experiments were performed using Lambdas deployed only on the Node.js 14 Lambda runtime environment. Lee et al. [Lee

[12] https://aws.amazon.com/cloudwatch

[13] https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency

et al. 2018] report fluctuations in the cost and performance of FaaS functions across different languages and FaaS platforms. While we did not investigate whether the performance of different languages/runtime environments can impact ASF, we believe that our primary conclusions are applicable irrespective of the programming language. This study can be repeated using another runtime environment to confirm this hypothesis in the future.

Furthermore, and although the current work is based on using a single Garmin serverless data processing pipeline application, the findings suggest that these observed results are not constrained to this particular use case. These experimental result trends should be reproducible when used on different ASF-realized use cases like media processing, microservice orchestration, and even machine learning serverless applications. Hence, these experiments need to be extended to examine more serverless applications developed using ASF as part of future work. Apart from assessing different use cases, the effects of other service integrations must be evaluated.

Finally, a natural progression of this work is to utilize the identified list of experiments and extend it to other FaaS orchestration systems like Azure Durable Functions, Google Workflows, and IBM Cloud Composer. Then the results from those evaluations can be compared to our observations, enabling serverless adopters to select from a broader range of vendors and finally more informed decisions while selecting a particular FaaS vendor and its corresponding FaaS orchestration system.

6 Related Work

Previous studies like [Eismann et al. 2020a; Elgamal 2018; Villamizar et al. 2016] provide various analyses and techniques to perform and evaluate cost and performance modeling of serverless solutions, but they are primarily focused on tuning AWS Lambda and equivalent FaaS offerings. Works by [Back and Andrikopoulos 2018; Hellerstein et al. 2018; Malawski et al. 2017; Shahrad et al. 2019; Wang et al. 2018] provide their insights into how Lambda memory size, cold starts, and communication overhead influence overall performance and cost. Other approaches such as [Malawski et al. 2020] and [Kuhlenkamp et al. 2019] are looking into the suitability of AWS Lambda and other FaaS offerings for implementing data pipelines. However, our research focuses primarily on ASF and the side effects of implementing a serverless solution using ASF to coordinate multiple Lambdas into flexible workflows that are easy to debug and modify.

From works that discuss specifically ASF, Lin et al. [Lin and Khazaei 2020] use ASF and AWS Lambdas to evaluate their proposed Probability Refined Critical Path Greedy algorithm (PRCP) to optimize cost and performance of Lambdas defined using ASF workflows. However, this research concentrates on AWS Lambdas and does not consider the state transition duration and workflow type. Moreover, most of the work presented does not reflect the latest billing model for AWS Lambda which rounds up the duration to the nearest millisecond with no minimum execution time. In [López et al. 2018] the authors compared three primary function orchestration systems: IBM Composer, ASF, and Azure Durable Functions. The authors concluded that ASF is the most mature one as it was the first to be available on the market and the most performant project for short and long-running orchestrations. The investigation also states that ASF has limited programmability constructs than other services, like a lack of parallel programming support, callback pattern, and nested workflows. However, in the past few years, ASF has advanced drastically by offering all these constructs, and even introducing the Express workflow type.

The work of [Bharti et al. 2021] modeled several sequential composition workflows like reflexive, fusion, async, chaining, and client-based scheduling using ASF and IBM Cloud Function Sequences. The study shows that the compositions implemented in IBM Cloud Function Sequences perform better than ASF workflows for both language runtimes, i.e., Python and Node.js. However, the authors also conclude that the cold start problem is more prominent for the IBM offering when compared to ASF.

[Wen and Liu 2021] performed the first empirical study on characterizing and comparing the leading FaaS orchestration systems, i.e., ASF, Azure Durable Functions, Alibaba Serverless Workflow[14], and Google Cloud Composer[15]. The authors compared their characteristics from six dimensions: orchestration, data payload limit, parallelism support, execution time limit, reusability, and supported development languages. Furthermore, they measured the performance of these orchestration systems under varied experimental settings: activity complexity, data-flow complexity, and function complexity using sequence and parallel applications scenarios. Based on their findings, ASF has been recommended by the authors for activity-intensive sequence, data flow-intensive sequence (or parallel), and function-sensitive sequence (parallel) tasks when considering the total execution time and orchestration overhead.

The mentioned investigations show a thriving interest in ASF serverless workflows, their performance benefits, and their ability to enable users to implement function composition. However, even with increasing interest, the lack of in-depth information regarding ASF and its workflow types hinders its adoption. Furthermore, the performed studies focus on ASF Standard workflow without considering the consequences the Express workflow can have on the application's performance. Finally, another vital aspect that these studies fail to consider is the cost impact for the workflow execution. Therefore, our findings and implications serve as baselines and suggestions for developers in adopting ASF workflow and choosing the most suitable type of workflow based on cost and performance.

7 Conclusion

In this work we performed an in-depth experimental study concentrating on the performance and cost effects on both ASF Standard and Express workflows with the help of an ASF-realized application. More specifically, we looked at various parameters and their impact on the cost/performance tradeoff by performing experiments on a serverless data processing pipeline use case. In particular, we investigated the effects of allocated memory size and cold/warm starts of orchestrated Lambdas, number of state transitions and state transition duration, total execution duration, payload size, and concurrent executions.

Our experiments show that Standard workflows are suitable for long-running tasks with few states, as they are priced by the number

[14] https://www.alibabacloud.com/product/serverless-workflow
[15] https://cloud.google.com/composer

of state transitions. So, even if a workflow takes months or milliseconds to execute, the price of workflow execution remains the same. Hence, workflows that expect an enormous volume of executions and have a considerable amount of transitions would not be ideal for executing using Standard workflows. We then observed that Express workflows with its billing similar to AWS Lambda (execution run, memory consumed, and total execution duration) is well suited for short-lived tasks even if the workflow has many state transitions. There is however an application-specific tipping point after which Express stops being more cost-efficient than Standard. Another interesting finding is that Express workflows execute faster than Standard workflows, as states are not persisted in Express workflows and state transitions happen in memory. On the other hand, Standard workflows persist the state on the disk and support history execution, which allows for better development support. When looking at Lambdas, cold starts have a massive impact on the step function execution. This effect impacts the performance of all workflow executions but has a much more noticeable influence on ASF Express workflows when coming to their cost. Increasing the allocated memory of the involved Lambdas improved the performance of the workflow execution, but the overall execution cost ended up being higher in some cases. Finally, we saw that the workflow execution duration increases with the increase in payload size, and therefore incoming application load needs to be taken also into consideration.

Acknowledgments

This work was supported by ITEA3 and RVO under grant agreement No. 17038 VISDOM (https://visdom-project.github.io/website).

The authors would like to thank the reviewing program committee members for their insightful comments and suggestions in improving this paper.

References

Lucas F Albuquerque Jr, Felipe Silva Ferraz, RF Oliveira, and SM Galdino. 2017. Function-as-a-service x platform-as-a-service: Towards a comparative study on FaaS and PaaS. In *ICSEA*. 206–212.

Timon Back and Vasilios Andrikopoulos. 2018. Using a microbenchmark to compare function as a service solutions. In *European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer, 146–160.

Andrew Baird, George Huang, Chris Munns, and Orr Weinstein. 2017. Serverless architectures with aws lambda. *Amazon Web Services* (2017).

Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017a. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

Ioana Baldini, Perry Cheng, Stephen J Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017b. 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*. 89–103.

Urmil Bharti, Deepali Bajaj, Anita Goel, and SC Gupta. 2021. Sequential Workflow in Production Serverless FaaS Orchestration Platform. In *Proceedings of International Conference on Intelligent Computing, Information and Control Systems*. Springer, 681–693.

Alessandro Bocci, Stefano Forti, Gian-Luigi Ferrari, and Antonio Brogi. 2021. Secure FaaS orchestration in the fog: how far are we? *Computing* (2021), 1–32.

Robert Cordingly, Hanfei Yu, Varik Hoang, David Perez, David Foster, Zohreh Sadeghi, Rashad Hatchett, and Wes J Lloyd. 2020. Implications of Programming Language Selection for Serverless Data Processing Pipelines. In *2020 IEEE DASC/PiCom/CBDCom/CyberSciTech*. IEEE, 704–711.

James Densmore. 2021. *Data Pipelines Pocket Reference Book*. O'Reilly Media, Inc.

Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. 2020a. Predicting the Costs of Serverless Workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. 265–276.

Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020b. Serverless applications: Why, when, and how? *IEEE Software* 38, 1 (2020), 32–39.

Adam Eivy and Joe Weinman. 2017. Be wary of the economics of "Serverless" Cloud Computing. *IEEE Cloud Computing* (2017).

Tarek Elgamal. 2018. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 300–312.

Jake Grogan, Connor Mulready, James McDermott, Martynas Urbanavicius, Murat Yilmaz, Yalemisew Abgaz, Andrew McCarren, Silvana Togneri MacMahon, Vahid Garousi, Peter Elger, et al. 2020. A Multivocal Literature Review of Function-as-a-Service (FaaS) Infrastructures and Implications for Software Developers. In *European Conference on Software Process Improvement*. Springer, 58–75.

Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

Jörn Kuhlenkamp, Sebastian Werner, Maria C Borges, Karim El Tal, and Stefan Tai. 2019. An evaluation of faas platforms as a foundation for serverless big data processing. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*. 1–9.

Jörn Kuhlenkamp, Sebastian Werner, and Stefan Tai. 2020. The ifs and buts of less is more: a serverless computing reality check. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 154–161.

Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 442–450.

Philipp Leitner, Erik Wittern, Josef Spillner, and Waldemar Hummer. 2019. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. *Journal of Systems and Software* 149 (2019), 340–359.

Changyuan Lin and Hamzeh Khazaei. 2020. Modeling and Optimization of Performance and Cost of Serverless Applications. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 615–632.

Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. 2018. Comparison of faas orchestration systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 148–153.

Maciej Malawski, Kamil Figiela, Adam Gajek, and Adam Zima. 2017. Benchmarking heterogeneous cloud functions. In *European Conference on Parallel Processing*. Springer, 415–426.

Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2020. Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. *Future Generation Computer Systems* 110 (2020), 502–514.

Johannes Manner, Martin Endreß, Tobias Heckel, and Guido Wirtz. 2018. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 181–188.

Antreas Pogiatzis and Georgios Samakovitis. 2021. An Event-Driven Serverless ETL Pipeline on AWS. *Applied Sciences* 11, 1 (2021), 191.

Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1063–1075.

Josef Spillner. 2017. Transformation of Python Applications into Function-as-a-Service Deployments. (May 2017). arXiv:1705.08169 [cs.DC]

Josef Spillner and Serhii Dorodko. 2017. Java Code Analysis and Transformation into AWS Lambda Functions. (Feb. 2017). arXiv:1702.05510 [cs.DC]

Erwin Van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. 2017. The SPEC cloud group's research vision on FaaS and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*. 1–4.

Erwin Van Eyk, Lucian Toader, Sacheendra Talluri, Laurens Versluis, Alexandru Uță, and Alexandru Iosup. 2018. Serverless is more: From paas to present cloud computing. *IEEE Internet Computing* 22, 5 (2018), 8–17.

Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. 2016. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 179–182.

Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 133–146.

Jinfeng Wen and Yi Liu. 2021. An Empirical Study on Serverless Workflow Service. *arXiv preprint arXiv:2101.03513* (2021).

Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. 2016. Building a Chatbot with Serverless Computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs* (Trento, Italy) *(MOTA '16, Article 5)*. Association for Computing Machinery, New York, NY, USA, 1–4.