

SeBS-Flow: Benchmarking Serverless Cloud Function Workflows

Larissa Schmid*, Marcin Copik†, Alexandru Calotoiu†, Laurin Brandner†, Anne Kozirolek*, Torsten Hoefler†

*KASTEL, Karlsruhe Institute of Technology, Germany

†Department of Computer Science, ETH Zürich, Switzerland

*first.lastname@kit.edu †first.lastname@inf.ethz.ch

Abstract—Serverless computing has emerged as a prominent paradigm, with a significant adoption rate among cloud customers. While this model offers advantages such as abstraction from the deployment and resource scheduling, it also poses limitations in handling complex use cases due to the restricted nature of individual functions. Serverless workflows address this limitation by orchestrating multiple functions into a cohesive application. However, existing serverless workflow platforms exhibit significant differences in their programming models and infrastructure, making fair and consistent performance evaluations difficult in practice. To address this gap, we propose the first serverless workflow benchmarking suite SeBS-Flow, providing a platform-agnostic workflow model that enables consistent benchmarking across various platforms. SeBS-Flow includes six real-world application benchmarks and four microbenchmarks representing different computational patterns. We conduct comprehensive evaluations on three major cloud platforms, assessing performance, cost, scalability, and runtime deviations. We make our benchmark suite open-source, enabling rigorous and comparable evaluations of serverless workflows over time.

Implementation: <https://github.com/spcl/serverless-benchmarks>
Artifact: <https://github.com/spcl/sebs-flow-artifact>

I. INTRODUCTION

Serverless computing gained major adoption in the industry [1], [2], with 50-70% of cloud customers using serverless functions and containers [3]. In the Function-as-a-Service (FaaS) programming model, developers implement stateless functions and invoke them through a REST interface. The actual function deployment and resource scheduling becomes the responsibility of the cloud operator: Developers are no longer concerned with managing their applications and are charged only for resources used to handle function invocations. While the primitiveness of FaaS can be an important benefit [1], it is also a major drawback: a single function is insufficient to cover all use cases. Functions must be composed to build larger applications, keep the design modular, or use pre-defined and standardized functions, e.g., for machine learning inference.

Serverless workflows introduced allow to chain and aggregate multiple functions into a single application by creating a graph of functions and automating the execution of a sequence through control and data dependencies. They include control-flow components - conditions and loops - which allows them to represent full computations such as multi-stage machine learning pipelines. Developers implement functions and define the workflow structure in a cloud-specific format. Cloud operators then control the workflow invocation and orchestration, retaining the ability to optimize resource consumption,

Papers	Total	Benchmarks						Platforms					
		Micro	Webapp	Multimedia	Data Proc.	ML	Scientific	AWS	Azure	GCP	Other	Research	Artifact?
Analysis	14	7	1	4	2	4	2	8	4	3	3	3	5
Optimization	17	8	3	4	4	5	6	9	0	2	2	7	4
Application	18	1	4	1	4	1	7	15	5	5	2	3	9
Prog. Model	23	10	6	5	8	11	8	10	3	1	2	16	11

TABLE I: Analysis of 72 research papers on serverless workflows with benchmarks.

e.g., through optimized function placement, oversubscription, targeting idle resources, and co-locating functions that depend on each other [4]–[6].

Workflows have been adopted by the most popular commercial cloud platforms [7]–[9] and make up almost a third of serverless applications [10]. However, just like every FaaS platform is different [11], serverless workflows are quite distinct from each other. Not only the different APIs and incompatible graph syntax and format complicate the software development process, but also fundamentally different programming models: workflow platforms diverge in the statelessness of functions and the static nature of graph definition (Section II-A). Even though FaaS platforms might seem like the same product, they offer drastically different performance, reliability, and cost [11]–[13]. With workflows built as an orchestration of functions, their functionality and performance is affected by both orchestration service and existing differences in the underlying compute infrastructure. As such details are hidden, an information gap between developers and providers arises [14]. Thus, the software developers need to conduct extensive performance testing of the cloud services to estimate the performance of their workloads and understand platform limitations up-front, as choosing a certain platform implies significant lock-in [15], with only limited support for testing [2].

We propose the first **serverless workflows benchmarking suite** to support software developers and the quickly growing research activity in serverless workflows. Our work provides a baseline and benchmarking methodology for evaluating and comparing the performance of workflows on different platforms, highlighting their strengths and weaknesses. We examined 72 different research contributions to determine the similarity of their evaluation baselines (Table I). We found that publications use different applications to benchmark the performance of new ideas, do not cover the same classes of

Platform	Prog. Model	Model Flexibility	Max. Parallelism	Interface
AWS	State Machine	Static	40	JSON
Azure	Orchestrator Function	Dynamic	Unlimited	Durable Functions
Google	State Machine	Semi-dynamic	20	JSON/YAML

TABLE II: Key features of serverless workflows platforms.

workloads, and do not always compare against the same subset of platforms. Without a consistent baseline, comparing research results and establishing the most promising ideas becomes impossible [16]. Benchmarking suites and systems have been proposed for FaaS [11], [12], [17], [18], but a benchmarking suite for serverless workflows has remained an open problem. A comprehensive, consistent, platform-independent, and portable benchmarking suite will support the ongoing research work [16], [19] and enable developers to differentiate between alternative solutions. We establish a unified and portable **workflow model** to abstract away the differences between different platforms (Section III). We design the **benchmarking suite** (Section IV) and include **six workflow benchmarks** based on solutions common in research and industry (Section V). Applications are implemented in our unified workflow model, providing an identical benchmark structure for each platform. We evaluate expressiveness and overhead of our model (Section VI) and use our benchmarking suite to comprehensively evaluate the three major cloud workflow services (Section VII). We follow the FAIR principle [20] and release our benchmark suite on an open-source license, enabling automatic repetition of our experiments, allowing reproducible results, and measuring performance changes in clouds over time. We make the following contributions:

- We introduce a platform-agnostic workflow definition, automatically transcribe the application into a cloud’s proprietary presentations, and enable developers to run near identical workloads on different systems.
- We propose a benchmark suite with six real-world application benchmarks and four microbenchmarks.
- We extensively analyze performance, cost, scaling, and stability of three major cloud platforms.

II. BACKGROUND

Serverless workflows introduce multiple new challenges to the software development process due to differences in the workflows platforms (Section II-A). To model workflows, we use the formalism and semantics of Petri Nets (Section II-B).

A. Developing Serverless Workflows

While software engineers are increasingly interested in serverless applications [21], they encounter a wide range of challenges while developing them, with the first questions about the different capabilities of the platforms arising before starting the implementation [21], [22]: Workflows have been adopted by all major cloud providers, but their implementations are significantly different in capabilities (Table II). We focus on AWS Step Functions, Google Cloud Workflows, and Azure Durable Functions, as they play a leading role.

```
tasks = []
for i in range(4):
    tasks.append(context.call_activity("process", i))
res = yield context.task_all(parallel_tasks)
```

(a) Azure Durable Functions

```
"assign_array": {
  "assign": [
    {"array": [0, 1, 2, 3]}
  ],
  "process": {
    "call": "exp.exec.map",
    "args": {
      "workflow_id": "map",
      "arguments": "${array}"
    },
    "result": "res"
  }
},
"separate map-workflow":
"main": {
  "params": [ "elem" ],
  "steps": [
    {
      "map": {
        "call": "http.post",
        "args": {
          "url": "google.process",
          "body": {
            "payload": "${elem}"
          }
        },
        "result": "elem"
      },
      {
        "ret": {
          "return":
            "${elem.body}"
        }
      }
    ]
  }
},
"init": {
  "Type": "Pass",
  "Result": "States.Array(0, 1, 2, 3)",
  "ResultPath": "$.array",
  "Next": "map"
},
"map": {
  "Type": "Map",
  "ItemsPath": "$.array",
  "Parameters": {
    "payload.$":
      "$$.Map.Item.Value"
  },
  "Iterator": {
    "StartAt": "process",
    "States": {
      "process": {
        "Type": "Task",
        "Resource": "arn:proc",
        "Parameters": {
          "payload.$":
            "$$.payload"
        },
        "End": true
      }
    }
  },
  "ResultPath": "$.res",
  "End": true
}
}
```

(b) Google Cloud Workflows

(c) AWS Step Functions

Fig. 1: Workflow invoking function *process* in parallel, with inputs from zero to three and results written to *res*.

The most important change is the programming model, affecting the implementation of the workflows, with unknown implications to workflow performance, an important property for developers [21]. As the different implementations are all provider-specific, moving workflows from one platform to another is complicated, causing vendor lock-in [22]. Azure uses the programming model of Durable Functions [23], where the workflow definition is encoded within a regular program structure of an orchestrator. The graph of functions is expressed using a mainstream programming language such as Python, as seen in the example of mapping the elements of input *values* array to invocations of the *process* function (Figure 1a). The computation model is built on top of stateless *activity* and stateful *entity* functions. On the other hand, developers need to define their workflow using a state machine on Google Cloud Workflows and AWS Step Functions. The workflow consists of states representing computations and transitions connecting them. The main states include function invocations, while supplementary states encode control flow. State languages defined with a syntax based on JSON and YAML files can be limited, verbose, and consequently difficult to debug, with missing tool support for testing and debugging already being a problem for developers [2], [24]. The example

Platform	Compute time	Invocation	Orchestration
AWS	\$0.0000167/GBs	\$0.20 per 1M	\$0.025
GCP	\$0.0000025/GBs	\$0.40 per 1M	\$0.01 (internal), \$0.025 (external)
Azure	\$0.000016/GBs	\$0.20 per 1M	\$0.000355

TABLE III: Pricing according to vendors’ documentation [25]–[29]. Orchestration per 1000 transitions.

implementations in Figure 1 demonstrate how simple code snippets can become much more verbose when compared to a native implementation of orchestrator. In Durable Functions, implementing the same behavior requires less work and the single-source implementation is more readable and easier to debug. However, the static form of a state machine gives the cloud provider deep knowledge of the functions executed and their order, allowing for optimizations.

The programming model also has an impact on the billing system. In addition to the cost of executing functions within a workflow, cloud providers charge users for workflow orchestration. In Azure, users have to pay for the duration of the orchestration function. In AWS and Google Cloud, users are charged per each transition of the state machine. Table III shows an overview. Note that we have to estimate the orchestration cost on Azure as billing is at the granularity of complete workflows only.

With the different platform-specific implications of implementing a workflow, it is difficult for developers to predict workflow costs on a given platform. To efficiently support them during the development of serverless workflows, we need a higher-level construct for workflows to abstract away the differences between platforms, enabling evaluation of the same workflow on different platforms and therefore facilitating informed decisions about the right platform.

B. Workflow Nets

We base our model on workflow nets with data (WFD-nets) [30]. They are an extension of Petri nets, usually used for business workflows. Basing the model on Petri Nets is only one possibility among alternatives such as state machines. We opt for Petri Nets due to their advantages as modeling formalism, such as their graphical nature, formal semantics, and analysis defined. Petri nets [31] describe the flow of information and control in concurrent and asynchronous systems. A Petri net is a triple $T = \langle P, T, F \rangle$ consisting of places P , a finite set of transitions T , and a set of arcs $F \subseteq (P \times T) \cup (T \times P)$. It is a workflow net *iff* there is a single source place $start$ without incoming arcs, a single sink place without outgoing arcs, and every node is on a path from source to sink [30]. WFD-nets [30] are a tuple $\langle P, T, F, D, r, w, d, grd \rangle$, consisting of a Petri Net $N = \langle P, T, F \rangle$ and additionally containing a set D of data elements on top as well as read, write, and destroy operations on these data elements. Moreover, the guarding function $grd : T \rightarrow G_D$ can assign guards to transitions. We show an example in Figure 2 where t_1 writes data to x , while t_2 and t_3 read from x . Dynamic system properties are modeled using tokens that are routed through the net. A transition is

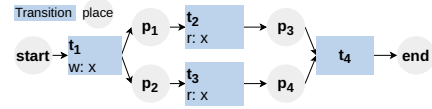


Fig. 2: WFD-net with transitions $T = \{t_1, t_2, t_3, t_4\}$ and places $P = \{p_1, p_2, p_3, p_4, start, end\}$

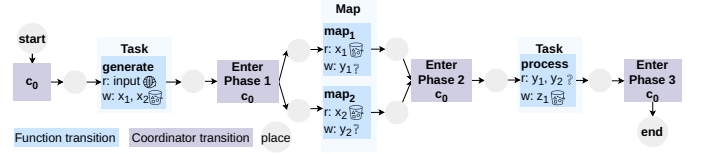


Fig. 3: Workflow using our model based on WFD-nets.

enabled if tokens are in all its input places $\bullet t = \{p | (p, t) \in F\}$. When it fires, it removes the token(s) from its input place(s) and routes them to its output place(s) $t \bullet = \{p | (t, p) \in F\}$. In our example, t_1 will be enabled if there is a token in $start$ and put tokens to p_1 and p_2 , which will enable t_2 and t_3 .

The platforms orchestrating serverless workflows that impose time limits on execution and schedule functions. Moreover, it is important to model how in- and output data is passed between functions. Modeling both of these is currently not supported by WFD-nets.

III. SERVERLESS WORKFLOWS MODEL

We define a model for serverless workflows that allows developers to implement and analyze a workflow application independent of the platform it will run on, alleviating provider lock-in. The model should encode the control flow and task parallelism, and clearly display the flow of data between functions, aiding developers in detecting scalability bottlenecks and errors, e.g., inconsistent or missing data. Therefore, we define our model on top of WFD-nets [30] (cf. Section II-B) and extend them to be able to express the orchestration by the platform and how data is passed between functions.

A. Transitions

The set of transitions T is composed of two types, the coordinators C and serverless functions SF , $T = C \cup SF$. Figure 3 shows an example with $C = \langle c_0, EnterPhase1c_0, EnterPhase2c_1 \rangle$ and $SF = \langle generate, map_1, map_2, process \rangle$.





A *function* transition $sf \in SF$ represents the execution of a serverless function. All function transitions that can run in parallel without any precedence dependencies and their immediate predecessor and successor places make up a workflow phase. There are different possible token routing constructs within one phase of the workflow: A *task* phase is a sequential routing, consisting of one function transition only. For parallel routing, there are two alternatives: First, a *parallel* phase can consist of any number of sub-phases that will be executed concurrently. Second, the *map* phase. Similar to the parallel phase, it can consist of any number of sub-phases, but each sub-phase is executed concurrently on

different elements of an input array. Figure 3 shows an example: The `map` functions compute $y_i = \text{map}(x_i)$ simultaneously for all i . A `switch` phase uses conditional routing based on values of data by annotating guarding functions to transitions.

The first transition of a workflow in our model is always a *Coordinator* $c \in C$ that initializes the workflow and schedules functions for execution. Additional coordinator transitions take place between phases, meaning that the coordinator awaits the termination of the currently running functions and afterwards schedules the functions of the next phase, explicitly modeling the orchestration of the workflow by the platform. For readability, we do not show the coordinator transitions when they can be skipped while preserving the control flow between function transitions, i.e., whenever a sequential phase is the next phase. This is because the sequential function already serves the purpose of the AND-join otherwise realized by the coordinator transition. In Figure 3, this means we can leave out all coordinator transitions after the initial c_0 transition.

B. Resource Annotations

Data labeling functions indicate the required inputs and provided outputs of a transition. However, for the performance of serverless workflows, it is important to know where the data resides and how it is provided. Therefore, we extend the notation of WFD-nets by annotating how the data is passed using the following resource annotations:

-  **Object storage.** Data is saved in cloud storage in the same region. While providing high capacity, it suffers from limited I/O bandwidth and high latency.
-  **Invocation Payload.** Protocols such as HTTP and gRPC can transfer small input data. However, the exact size limit is subject to the protocol and platform.
-  **Transparent.** The type of transmission used when returning a payload is up to the provider and can change given the payload size.
-  **Reference.** Some functions only need the reference to an object in the object storage rather than the object itself.

We annotate data location in workflows using the respective icon and show an example in Figure 3. The function `generate` receives a payload via an invocation payload and stores its output on the object storage. The `map` functions each receive an element of the array, process it, and return their resulting elements y_1 and y_2 through a protocol decided by the cloud provider. Once both `map` functions have returned, the `process` function receives y_1 and y_2 as input and, finally, uploads the final result z of the workflow to the object storage.

IV. WORKFLOWS BENCHMARK SUITE

We now present the design and implementation of SeBS-Flow¹. To enable reliable and fair comparison of various workflow platforms, we need to execute the same benchmark implementation on many platforms. However, the platforms exhibit vast differences in the programming model and API

¹An extended definition and discussion of benchmarks can be found in the Master thesis [32].

<pre>"process_names": { "type": "map", "array": "customers", "root": "shorten", "next": "list_emails", "states": { "shorten": { "type": "task", "func_name": "short" } } }</pre> <p>(a) <i>Map Statement.</i></p>	<pre>"send_if_enough_data": { "type": "switch", "cases": [{ "var": "data.length", "op": ">=", "val": 1048576, "next": "send_large" }, { "var": "data.length", "op": ">=", "val": 1024, "next": "send" }], "default": "log" }</pre> <p>(c) <i>Switch Statement.</i></p>
<pre>"process_10": { "type": "repeat", "func_name": "process", "count": 10 } }</pre> <p>(b) <i>Repeat Statement.</i></p>	

Fig. 4: Workflow definition language: a portable specification of control-flow and data dependencies.

of their workflow services (Section II-A). Thus, we define a platform-agnostic workflow definition (Section IV-A) based on our workflow model (Section III). Then, we propose platform-specific generators that transcribe workflows to the respective proprietary definition of the desired platform (Section IV-B). We add the workflow representation and implementation to a serverless benchmark suite (Section IV-C).

A. Platform-Agnostic Workflow Definition

Our workflow model encodes the application as Petri Net (cf. Section III). Every phase has a `type`, relating to one of the available routing constructs (cf. Section III-A). Coordinator transitions encode the order of phases, represented by the `next` field of phases that describes the consecutive step in the workflow. The workflow will terminate if the `next` field is not set. Each phase receives the output payload of the previous function as input. We encode the different phases as follows: *Task*. A `task` executes a single serverless function.

Map. The `map` phase concurrently executes the given states one after another on each element of the given array and returns an array again. The phase can define `common_parameters` from the running variable that will be passed in addition to the array element. Listing 4a shows an example with the `process_names` phase: for each element of `customers`, the function `short` is executed concurrently. Only after all functions have terminated, the coordinator will transition to the next phase, which in this case is `list_emails`.

Loop. The `loop` phase is similar to `map` but traverses the given input array sequentially. Thus, `loop` encodes tasks that cannot be parallelized due to existing dependencies.

Repeat. A `repeat` phase executes a function a given number of times. This syntactic sugar eases the modeling of a chain of tasks. Listing 4b presents an example where the function `process` is invoked 10 times, and the return payload of the i^{th} invocation is passed onto the $i+1^{\text{th}}$ execution.

Switch. The `switch` phase dynamically decides the next phase at runtime depending on the given condition. Listing 4c presents

a simple `switch` phase where different functions are executed depending on the running variable `data.length`.

Parallel. This higher-level phase executes sub-workflows, consisting of any of the phases, concurrently.

B. Platform-Specific Transcription

We map the six phases building a serverless workflow to different features of the modeling language on each platform.

1) *AWS:* The most notable difficulty when transcribing our definition to the state machine definition of AWS Step Functions is the `loop` phase. Step Functions do not inherently support sequential array iteration. Their official documentation suggests using an additional serverless function that iterates over a given range [33], which is inefficient. Thus, we use the AWS `map` state and configure it to traverse the given array sequentially, yielding the semantics of a `loop`. A downside of this approach is that the input to each function is the same, i.e., consecutively executed functions can observe the results of computations of their predecessors only if uploaded to the object storage.

2) *Google Cloud:* Google Cloud Workflows do not natively support a `task` type. Instead, the recommended approach for invoking Cloud Functions [34] is to create a state performing a POST request and providing the trigger URL of the desired function as input. However, this requires additional states for each `task` and `map` to parse the HTTP response of a function and assign results. Moreover, the parallel `map` execution accepts only other workflows and not states, which requires creating another sub-workflow, even if it contains only a single function to be invoked. Finally, there is no mechanism for passing additional arguments to a `map` function, which is necessary for us to track measurements. As a workaround, the input array is zipped together with an array consisting of the additional parameter passed by the benchmarking infrastructure.

3) *Azure:* Azure uses the dynamic model of Durable Functions instead of state machines. There, we upload our workflow definition together with the function code. The user-provided orchestrator parses the definition as input, decodes our definition, and executes it by spawning new function executions.

C. Benchmark Suite

We follow standard design practices to build a new benchmark suite: it should be relevant, extensible, easy to use, and reproducible [11], [35]–[37]. Our suite is relevant as we include applications representing a variety of workloads in the industry and academia (Section V). The implementation is based on an abstract workflow definition and can be extended to new platforms by implementing a single interface that transcribes our model definition to the new platform. To fulfill the two remaining criteria, we build our implementation upon SeBS [11], an established benchmark suite for FaaS: Benchmarks must be easy to deploy and execute to ensure their self-validation [35]. Integration into a maintained and up-to-date platform helps integrating new developments of serverless platforms continuously and avoids pushing this task to the end user. SeBS-Flow is multi-platform, supports automatic deployment of functions to the cloud, and integrates with

Benchmark	#functions	Parallelism	Critical path	Download [MB]	Upload [MB]
Video	4	2	3	238.83	7.48
Trip Booking	7	1	4/7	0.0	0.0
MapReduce	9	5	4	0.02	0.04
ExCamera	16	5	6	302.07	17.49
ML	3	2	2	7.82	3.91
1000Genome	19	12	4	273.54	3.47

TABLE IV: Key features of different benchmarks.

services like storage and cloud logging, allowing developers to focus on the actual implementation rather than specifics of cloud providers, which can be time-consuming [38], [39].

Serverless functions need cloud-managed storage to access data and retain state across invocations. To that end, SeBS automatically manages object storage instances and provides functions with a multi-cloud API. To create realistic workflow representations of web applications, we need to support low-latency data stores other than object storage. We chose NoSQL key-value storage for this task and extended SeBS with a high-level interface for creating, modifying, retrieving, and deleting items. The interface supports a partition and an optional sorting key. Each benchmark function can use multiple tables managed by the benchmark suite. We map the tables to DynamoDB on AWS, CosmosDB on Microsoft Azure, and Firestore in Datastore mode on Google Cloud.

We collect timestamps for *start* and *end* of each function, its *requestID*, and a *containerID* to detect container reuse by using the temporary filesystem and global variables. The runtime of a phase is defined by the *start* of its earliest function and the *end* of the latest one. All collected values are sent to a Redis [40] instance deployed in the same cloud region. We chose an in-memory cache as it provides sub-millisecond latencies, reducing the risk of distorting the performance measurements.

V. BENCHMARK APPLICATIONS

In SeBS-Flow, we implement six benchmarks covering real-life workloads. Also, we implement four microbenchmarks used in the evaluation: function chain, object storage performance, parallel invocations (Section VII-C1), and selfish detour (Section VII-C2). The selected benchmarks cover various domains that use workflows (Table IV), and correspond to previous findings on the characterization of workflow use cases [10], [41] regarding control-flow, number of functions, parallel invocations of the same functions, and longer runtimes: While 33% of workflows include complex control-flow, 50% are sequential, which we cover with Trip Booking and function chain microbenchmark. 72% of workflows use less than ten different functions, 52% involve parallel invocations of the same function, and 25% contain functions with a runtime of over a minute, which is also included in our benchmark suite. We visualize only one of the benchmarks here, but provide figures for the other benchmarks in the supplementary material.

a) *Video Analysis:* The benchmark detects objects in a video, and parallelizes the sequential benchmark in vSwarm [42] (Figure 5). Functions decode video frames and apply the Faster R-CNN model [43]. The `decode` function first downloads the video, decodes F frames, and then uploads $N = \lceil \frac{F}{B} \rceil$ batches of size B . N parallel `detect` functions

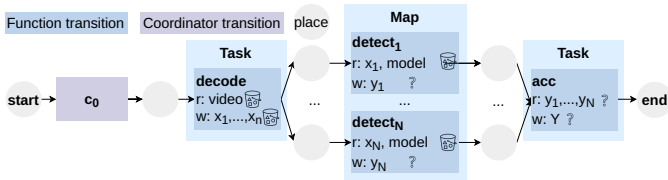


Fig. 5: The Video Analysis benchmark.

compute Y_i , all detections with confidence $p > 0.5$. Finally, detections are accumulated in `acc`, returning the final payload Y . We used $F = 10$ frames and batch size $B = 5$, yielding two parallel functions in the map phase.

b) *Trip Booking*: The benchmark represents web applications, and it mocks a common example of reserving a hotel, car rental, and flight [44], [45]. The workflow is a pipeline of functions mocking the reservation system by storing trip data in a shared NoSQL database. It implements the SAGA pattern of long-running transactions [46] where a failure triggers the reversal of prior changes. For testing, we simulate failure in the last `confirm` function, which is followed by three consecutive functions to reverse the booking.

c) *MapReduce*: We base our example on prior implementations [42], [47] and perform the standard problem of word counting. First, the `split` function partitions the input text into N batches. N parallel `map` functions count how often each word occurs in their text chunk next. Next, `shuffle` flattens the resulting array $Y_i | i < M$. Finally, M reducers count the total occurrences of their respective word in parallel, yielding Z_i . The benchmark has two parameters: the number of mapping functions N , and the total number of words W . We set $N = 3$ and $W = 5000$, containing $M = 5$ different words. MapReduce frameworks typically execute fully in parallel. However, the available workflow primitives necessitate the `shuffle` function, not relying on the array Y_i itself but flattening it to enable the desired level of parallelism in reduce.

d) *ExCamera*: ExCamera [48] uses interdependent video-processing tasks to encode videos in parallel. A video with M total frames is processed in chunks of N frames by $\frac{M}{N} = T$ parallel functions. First, each frame is encoded, yielding one key frame and $N - 1$ interframes. Decode decodes all N frames again, calculating the final state. The final state from the first frame of the chunk is used for reencoding the other frames, resulting in one final state and $N - 2$ interframes. We derive our implementation from the original description of ExCamera [48] and the available implementation [49]. We use $M = 30$ total frames and a chunk size of $N = 6$, resulting in five parallel functions.

e) *Machine Learning*: This workload represents a typical training pipeline: It starts with `gen` generating a dataset, with the number of samples N and the number of features M as input. Then, we `train` K different classifiers C_i in parallel. We generate $N = 500$ samples and $M = 1024$ features, and train $K = 2$ classifiers: a Support Vector Machine [50], and a Random Forest [51], creating two concurrent functions.

This is a scientific workflow that identifies mutational overlaps using data from the 1000 Genomes project. It consists of five tasks and three phases: First, N individuals functions parse the data for their chunk of the input file of size M and then upload their results to the cloud storage. While `individuals_merge` merges the results to one, `sifting` computes the SIFT scores. In the last phase, `mutation_overlap` measures the overlap in SNP variants and `frequency` measures the frequency of mutation overlapping, both by population P . The benchmark has the number of lines as input M , number of parallel `individuals` functions N , and number of populations P as input variables. We use $M = 1250$ lines, $N = 5$ parallel individuals function, and $P = 6$ populations.

VI. EVALUATION OF WORKFLOW MODEL

By reviewing existing literature on serverless workflows, we evaluate whether our model is general enough to express applications of workflows and if our transcription to the platform-specific representations adds overhead compared to the native implementation. We do so by using the meta-search engine Google Scholar to find peer-reviewed publications containing the keywords *cloud*, *orchestration*, and *serverless workflow* or *serverless DAG*. We exclude papers that are not in English, do not use a workflow benchmark, or are published before 2017, the year of the first serverless workflows in the cloud. This results in 72 papers analyzed papers (cf. Table I, p. 1 for their categorization). We provide the complete list of papers and analysis results in the supplementary material.

A. Expressiveness of our Model

We analyze the workflow benchmarks used in the literature and evaluate whether our model can represent the control flow within the workflows without adding unnecessary dependencies between their tasks. Out of the 72 papers, 14 did not provide sufficient detail on the workflows used and their dependencies to judge if we can express them. In two papers, benchmarks are not presentable by our model, as they introduce new programming models to support communication between functions and load-balanced orchestration. Benchmarks used in three more papers can be modeled but not transcribed to platform-specific representations (Section IV). For two of them, cloud platforms are the limitations, such as ending the workflow as a result of a switch state (not possible on AWS) and using multi-stage inputs, i.e., using the output of a previously executed function as input without passing it to the functions invoked in-between. While we do not currently support transcribing the *switch* state requiring two conditions to be true, it can be easily added to the implementation. We fully support modeling and transcribing the workflows described in 53 of the 58 analyzed papers. Therefore, we conclude that our model does not have general limitations and developers can use it to model and execute their workflows.

B. Overhead of our Model

To check if our model and transcription create overhead compared to a native implementation, we evaluate available

benchmark implementations used in the analyzed papers and compare them to our transcription of their workflows. Only 10 of the 72 papers include an artifact containing workflow implementations or show their implementation as part of the paper for any of the platforms we support. None of them uses Google Cloud Workflows. In total, we find eleven AWS Step Functions state machines. One of them uses the *AND* choice type we do not transcribe, and another one adds *fail* and *success* states before ending the workflow, which only introduces overhead as compared to just ending the workflow. The other nine state machines use the same states with the same parameters in the same order as the state machines we transcribe, except for the fact that they specify each parameter explicitly as part of the state machine while we wrap them within a single *payload* entry. Four of the papers provide implementations for a total of six workflows using Azure Durable Functions. While one paper only provides an implementation using entity functions, the other five workflow implementations use activities to orchestrate tasks similar to our transcription. Since we must parse the platform-independent representation within the orchestrator, we could introduce an overhead. However, the evaluation of the 1000Genome benchmark, the benchmark with the most functions, shows that the average duration of the orchestrator function is only 13.6 milliseconds. We conclude that SeBS-Flow does not introduce noteworthy overhead in the workflows compared to their native implementation, enabling developers to obtain realistic performance results for their workflows.

C. Threats to Validity

We used only one query to find relevant works, bearing the risk of missing results. We mitigated this by evaluating different queries beforehand, evaluating the relevance of papers found and checking if relevant papers we knew were included. Regarding external validity, we found only a limited number of artifacts to evaluate the overhead, with none available that uses GC Workflows. While our transcription follows best practices and tutorials as provided by the cloud providers and matches the artifacts we found, usage in other projects could differ.

VII. EVALUATION OF CLOUD SERVICES

We use SeBS-Flow to evaluate three major cloud workflow services – AWS Step Functions, Google Cloud Workflows, and Azure – providing developers valuable insights regarding their suitability for different workloads. We investigate the following research questions:

- RQ1** What are the runtime differences between platforms?
- RQ2** What causes runtime and stability differences?
 - RQ2.1** What causes variations in the critical path?
 - RQ2.2** What causes overheads between function invocations?
- RQ3** How well can serverless workflow orchestration support scientific workflows?
- RQ4** How does the pricing compare between platforms?
- RQ5** How did the performance and stability of the platforms evolve over time?

A. Methodology

We deploy benchmarks on Azure to the *europa-west* region, on AWS to *us-east-1*, and on Google Cloud to *us-east1*. We use the lowest common memory configuration that successfully executes the workflow on AWS and Google Cloud, at least 256 MB for computational functions and 128 MB for simple web applications. We invoke the application benchmarks in *burst* mode, triggering 30 executions at once and accepting all successful workflow executions, as other work suggests that most serverless applications have potentially bursty workloads [10]. We check how often we should repeat experiments by computing non-parametric confidence intervals on the results for the MapReduce benchmark and aim at being in a 5% interval of the median with a 95% confidence interval. For the burst mode with 30 executions triggered at once, this results in 1, 1, and 6 repetitions on AWS, GCP, and Azure, respectively. We opt to execute all experiments 180 times. However, we could only obtain 30 executions of the 1000Genome benchmark on Azure due to frequent timeout issues. Benchmarks use the serverless object storage and NoSQL database on each platform.

B. RQ1: Runtime Differences among Platforms

We compare the runtime of each benchmark on selected platforms. We calculate the runtime by subtracting the first *start* timestamp from the last *end* timestamp. The results in Figure 6 do not yield a single fastest platform among all our benchmarks. AWS is the fastest platform for three out of six benchmarks while performing relatively well for the other three. While Google Cloud’s performance is comparable to AWS, it is 1.55-1.97x slower on three benchmarks. While Azure Durable functions perform very well, e.g., on MapReduce and Machine Learning, they are the slowest platform for Video Analysis, ExCamera, and the 1000Genome benchmark. For Trip Booking, Azure achieves the best median performance but suffers from large outliers. We investigate the potential causes of slowdown in the next section. All platforms demonstrate variable performance, with Azure showing the largest variance.

C. RQ2: Causes for Runtime and Stability Differences.

According to our results, AWS and Google Cloud provide a performance-reliable workflow service, whereas the variability is considerably higher on Azure. Thus, we split the runtime into two components: the critical path T_C computed as the sum of all states’ maximum runtime within one phase, and the overhead T_O caused by the scheduling and data movement conducted by the cloud workflow service. We calculate the overhead by subtracting the critical path from the total runtime. Figure 7 presents the critical path and overhead for all benchmarks. Azure’s runtime is dominated by scheduling overhead: For example, the overhead of the ExCamera benchmark is, on average, 495.5s, more than $36\times$ as long as its critical path of 13.5s. The ML benchmark incurs the least overhead of $5\times$ the length of its critical path. Also, Azure’s critical path is very fast across all benchmarks, demonstrating the fastest critical path for ExCamera, MapReduce, and Machine Learning. Google Cloud, however, has the slowest critical path throughout the

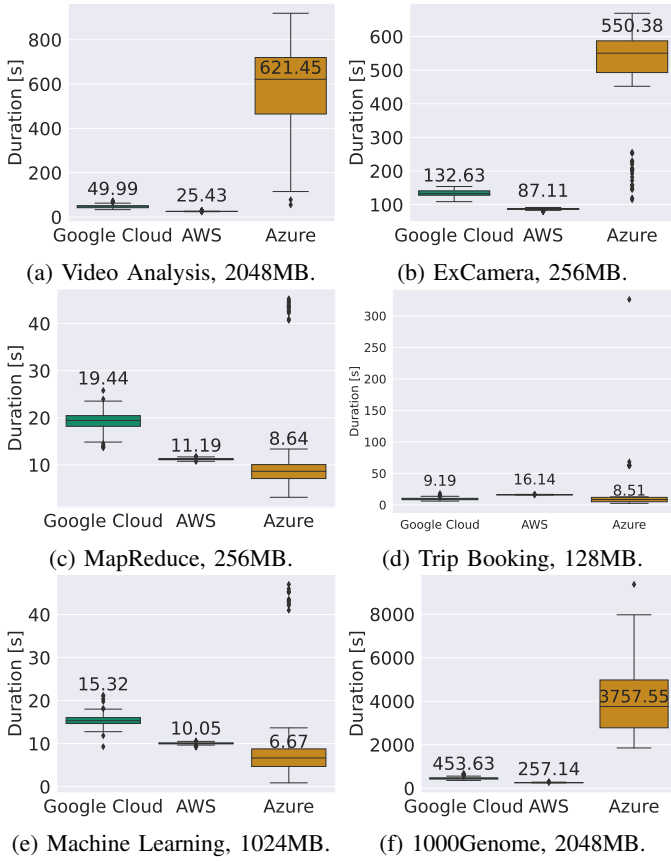


Fig. 6: Runtime of benchmark applications on AWS Step Functions, GC Workflows, and Azure Durable, *burst* invocations.

entire benchmark suite. In summary, orchestration overhead causes long runtimes and performance variances on Azure. For AWS and Google Cloud, however, the critical path varies.

1) *RQ2.1 Sources of Overhead*: We analyze three common sources of overhead: object storage I/O, parallel schedule, and function return payload.

a) *Cloud Storage I/O*: The data downloaded from the object storage differs between benchmarks (Table IV, p. 5), with hundreds of megabytes in ExCamera, 1000Genomes, and Video Analysis. These benchmarks experience the highest relative overhead of $36.7\times$, $10\times$, and $14.95\times$ their critical paths on Azure. To verify that this correlation is indeed causation, we execute a microbenchmark evaluating the cloud storage I/O performance. We invoke 20 functions in parallel where each attempts to download a file of size D from the storage. Figure 8a shows that the overhead remains stagnant for AWS at around one second and nearly stagnant on Google Cloud, increasing a bit for downloads larger than 1MB. On Azure, we observe an overhead of almost 149 and 4.9 seconds for 128 and 1 MB files, respectively. This can account for a significant part of the large overhead measured on Azure Durable.

b) *Parallel Scheduling*: Another potential source of overhead are parallel invocations within a benchmark. Benchmarks with the highest degree of parallelism – ExCamera and 1000Genomes – show the largest overheads of Azure. We test

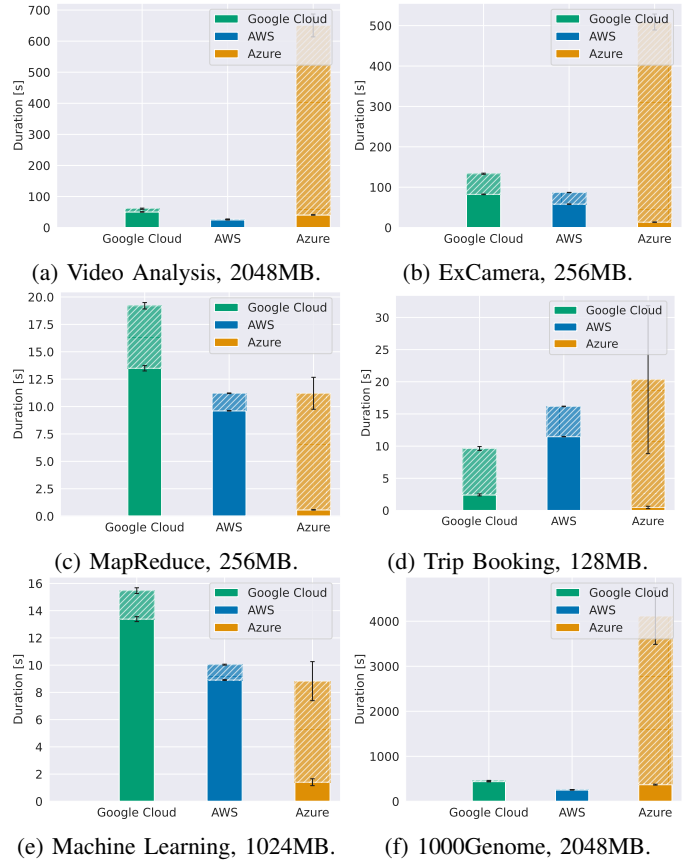
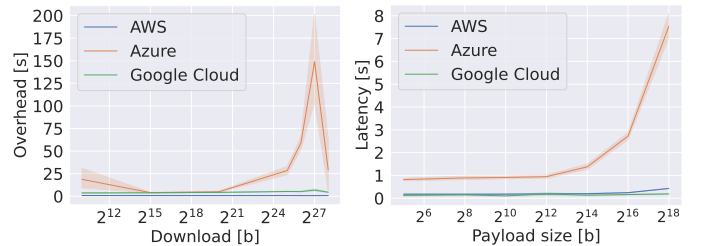


Fig. 7: Critical path (opaque) and overhead (hatched) of different benchmarks on considered platforms, *burst* invocations.



(a) Overhead of storage I/O, 20 functions, $2^{10} \leq D \leq 2^{28}$, 256MB , *warm* invocations. (b) Invocation latency, 512MB , *burst* invocations.

Fig. 8: Analysis of different sources of overhead.

this by executing a microbenchmark that spawns N functions in parallel, each one sleeping for T seconds, and start 30 such invocations concurrently. Figure 9 shows the relative overhead of the actual runtime compared to the function execution time. AWS functions demonstrate modest overhead, with largest values for the shortest duration. GC functions present a larger slowdown that increases with the number of parallel tasks. There, the system puts a cap on scaling up and reuses containers, as 30 invocations with $N = 2$, $T = 1$ start 60 different function containers on AWS, but only 30 on Google Cloud. On the other hand, Azure experiences an order of magnitude larger

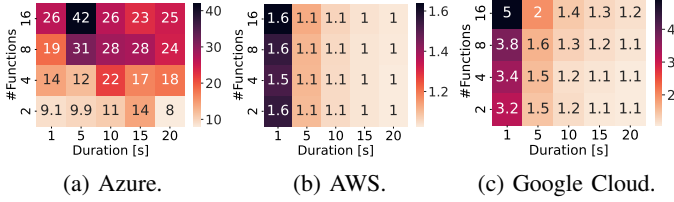


Fig. 9: The overhead of parallel sleep microbenchmark, $2 \leq N \leq 16$, $1 \leq T \leq 20$, 256MB, *burst* invocations.

overhead that increases with the parallelism factor but does not seem to be correlated to the function runtime.

To better understand the impact of limited parallel scalability on our benchmarks, we measure the number of distinct sandboxes allocated at any given time until the last function execution has terminated. We invoke 30 concurrent executions of workflow benchmarks and display the scaling behavior in Figure 10. Throughout the benchmarks, AWS and Google Cloud exhibit similar scaling behaviors, and their scale-up curves reveal the same local maxima, with phase transitions visible. However, we can also see that AWS spins up new containers more quickly. Azure produces a much more constant curve that remains similar throughout the benchmarks, never allocating more than 10 containers simultaneously.

c) Return Payload: We evaluate the overhead resulting from the function return payload size. We deploy a microbenchmark consisting of a function chain, where functions return M bytes of result sent to the consecutive function, with ten functions and test varying input sizes until Google Cloud’s limit. We invoke the chain 30 times simultaneously and use results from warm invocations only. Figure 8b shows that the latency remains constant for AWS and Google Cloud, while it increases dramatically for Azure from 16 kB, suggesting an influence of remote storage or queue. While this may present a significant source of overhead in applications, our benchmarks do not return payloads larger than 1MB, and this overhead can only account for a part of the slowdown.

d) Conclusions: The microbenchmarks demonstrate that a significant part of the overhead observed on Azure originates from the parallel schedules and storage IO. Another potential source is dynamic orchestration. A statically scheduled system could optimize function placement, data prefetching, and scalability by using a priori knowledge.

2) *RQ2.2 Critical Path Discrepancy:* The runtime of benchmarks across platforms shows that additionally to varying overhead, the critical path of computation can be significantly different. To understand the reasons behind this difference, we analyze how the critical path is impacted by two factors: the varying CPU allocation and frequency of cold starts.

a) OS Noise: The cloud provider controls the CPU allocation to a serverless function, either in relation to the memory configuration on AWS and GCP [28], [52], or in an undisclosed fashion on Azure. We use the selfish detour benchmark to quantify OS noise [53], which allows us to estimate how long the function is suspended by the OS, which in turn approximates

Benchmark	Cold starts			State transitions	
	AWS	GCP	Azure	AWS	GCP
Video	86.94%	68.61%	3.89%	7	20
MapReduce	100%	68.17%	1.0%	14	54
Trip Booking	100%	38.24%	0.6%	9	16
ExCamera	73.58%	69.34%	0.94%	21	73
ML	100%	99.26%	2.60%	6	18
1000Genome	98.16%	72.40%	7.72%	26	96

TABLE V: Relative #cold starts and #state transitions.

the vCPU timeshare. The benchmark runs a tight loop and records the event that one iteration took significantly more cycles than expected N times. The magnitude and frequency of these events characterize the suspension and noise. We deploy a workflow with a single function executing the benchmark, invoke it 30 times concurrently, collect $N = 5000$ events, and sample warm invocations to obtain consistent results. Figure 12a compares the relative to the expected suspension time according to the cloud documentation. We observe less noise on Google Cloud when compared to AWS, with more than 20% difference on 1024MB memory. We normalize the critical path per platform using the following approximation: given a function with memory configuration M , we represent the relative duration of function suspension as S_M and compute the normalized critical path $T'_C = T_C * (1 - S_M)$. We observe the largest relative discrepancy on two benchmarks, MapReduce (Figure 12b) and Machine Learning (Figure 12c). The overall trend observed in Section VII-B remains unchanged: Google Cloud demonstrates the longest critical path duration. The suspension time explains the shorter critical path on Azure when testing low-memory function configurations on AWS and GCP: Azure functions receive larger CPU allocations.

b) Cold Starts.: Cold invocations add significant overhead to the function execution [11]. Table V shows the frequency of cold starts in our measurements, with cold starts identified using the containerID (see Section IV-C). Azure Durable performs significantly better, likely because function apps on Azure can hold many invocations concurrently [11]. While the low scalability causes high orchestration overheads, it benefits the computations by putting them in warm containers. Figure 11 shows the impact of cold starts on the critical path and overhead. Due to the high percentage of cold starts in our measurement data, we collected another 60 workflow invocations for AWS and GCP with at least one warm function and show the critical path for the resulting completely warm invocations. Google Cloud and AWS functions perform up to $2.0\times$ and $4.5\times$ better, respectively, achieving almost the same performance as Azure. Thus, cold starts are a major factor influencing the slowdown and performance instability observed in many benchmarks.

D. RQ3: Usability for Scientific Workflows

There is rising interest in the scientific community to use serverless solutions [41], accompanied by experimentation with serverless offerings of the platforms [54] and management systems for serverless execution of workflows [55]–[58]. However, they do not consider the workflow orchestration systems

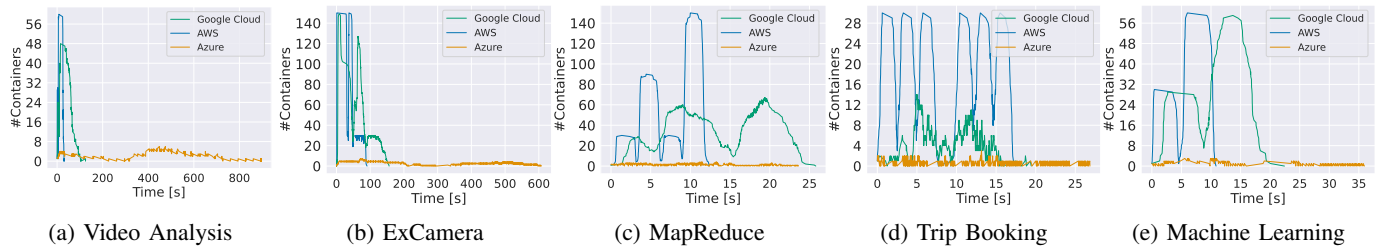


Fig. 10: Scaling profiles: the number of distinct containers used for 30 consecutive workflow invocations.

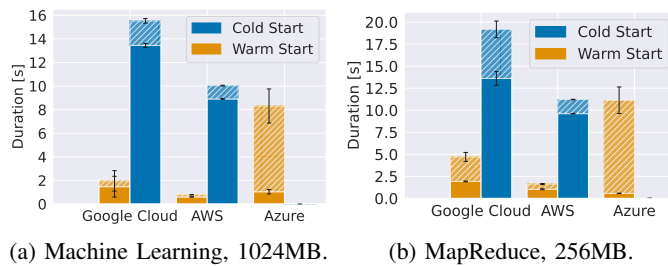


Fig. 11: Critical path (opaque) and overhead (hatched) of warm and cold invocations.

the cloud platforms offer. We use the scientific benchmark *1000Genome* to compare cloud services and the HPC system Ault using nodes equipped with Intel(R) 6154@3.00GHz CPU, repeating measurements five times.

First, we compare the runtime of the total workflow, as shown in Figure 13a. While the workflow execution time is, on average, 457.7s and 259.8s on GCP and AWS, respectively, the execution takes only 7.7s on Ault. GCP exhibits a coefficient of variation of 12.2%, while AWS has a coefficient of variation of only 3.3% - even lower than 4.1% on Ault. Interestingly, I/O takes less than one second on AWS, meaning that the computation is slower in the cloud. Then, we compare the scaling behavior of the different platforms for the `individuals` task of the workflow. We employ strong scaling, i.e., adding more jobs while keeping the size of the input file the same, resulting in smaller chunks per job. Figure 13b shows the speedup of 1.96 and 1.95 on AWS, 1.91 and 1.95 on GCP, and 1.51 and 1.24 on Ault for 10 and 20 jobs, respectively. The cloud platforms achieve a nearly-optimal speedup, which is not surprising given the high overhead for the baseline execution.

E. RQ4: Pricing

We compare the average cost of executing a workflow and estimate the prices, as shown in Table III, p. 3. Functions invoked during the execution of a workflow are billed based on the integral of memory and duration. Figure 14 visualizes the cost of workflow execution split into two groups: function execution (opaque) and the cost of orchestrating the state machine (hatched). Note that, due to Azure’s billing and measurement system, we could only retrieve an average cost value over all workflow invocations. Even though the Trip Booking benchmark is a simple pipeline with error catching,

running it with workflow orchestration still adds significant state transition costs. Azure is the most expensive service for the *1000Genome* benchmark. Google Cloud is the most expensive for MapReduce due to the high number of state transitions. AWS Step Functions are the most expensive solution for the other four benchmarks because functions cost $6.7\times$ more for computation than Google Cloud Functions. The price charged for state transitions is nearly identical between AWS and Google Cloud, even though AWS charges $2.5\times$ more: the AWS state language requires fewer states to implement the benchmarks (Table V).

In addition to execution and orchestration costs, workflows can generate charges when accessing the object and NoSQL storage. In all three clouds, the prices of read and write operations on the object storage are exactly the same. However, the billing models for key-value storage differ: DynamoDB charges for operations according to the amount of data read and written in strictly defined size increments; CosmosDB applies the same pricing to request units but does not explicitly define expected consumption; and Datastore has higher costs per operation but makes the cost independent of the item size. To understand the impact of price differences, we analyze the full execution of the Trip Booking benchmark. One workflow invocation requires three insertions and three deletions, with all items taking at most a few hundred bytes. While the estimated storage costs are similar on each platform, between $\text{€}0.68$ and $\text{€}1.08$ for one thousand executions, they impact the final cost differently. NoSQL operations add only 2.74% and 6.72% of the total price on AWS and GCP, respectively. The total execution cost on Azure is just $\text{€}2.4$. There, the estimated cost of CosmosDB request units is equal to $\text{€}0.68$ and adds 28.5% of workflow price.

F. RQ5: Evolution of Performance

Finally, we assess the performance stability over time by comparing July 2022 and January 2024 results. The executions from 2022 contain 30 invocations per workflow using Python 3.7, in cloud regions *europa-west* for Azure, *europa-west-1* for GCP, and *us-east-1* for AWS. The 2024 invocations are run in the same regions, except for GCP in *us-east-1*, and use Python 3.8. Figure 15 shows the results. The critical path and overhead of the MapReduce and ML benchmark are approximately the same on Google Cloud. The runtime on AWS is quite stable without any notable differences between 2022 and 2024. Azure has a stable duration of the critical path. While the overhead

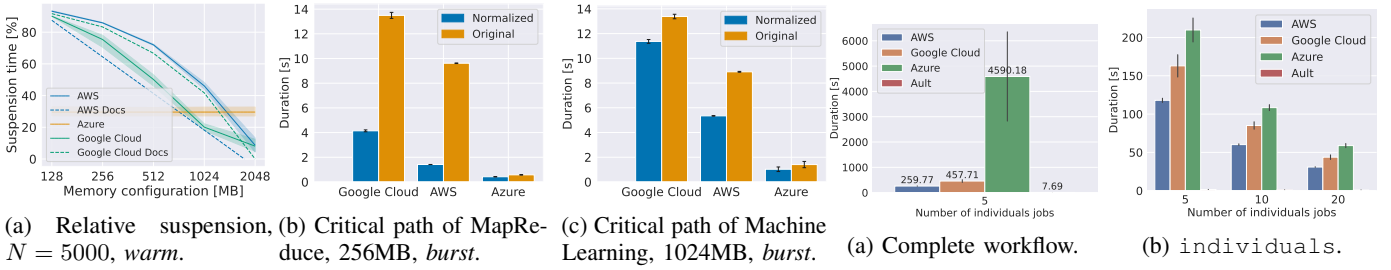


Fig. 12: Analysis of OS noise.

Fig. 13: Scalability of the 1000Genome workflow.

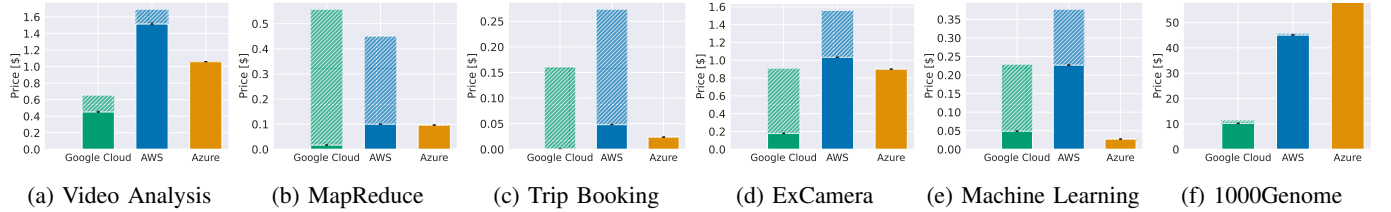


Fig. 14: Price per 1000 workflow executions: function costs are opaque and state transition costs are translucent.

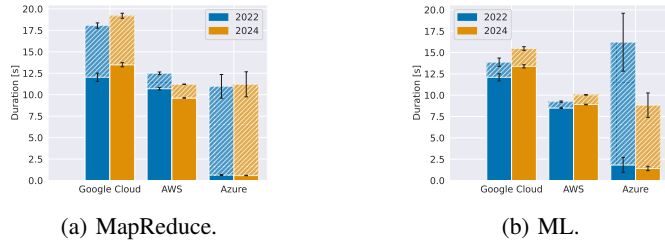


Fig. 15: Comparison of critical path (opaque) and overhead (hatched) between 2022 and 2024, *burst* invocations.

for MapReduce is the same in 2024 as in 2022, the overhead of ML has been approximately halved from 2022 to 2024.

G. Threats to Validity

A threat to the external validity is our choice of benchmark applications. We mitigate this by using applications from different domains that correspond to previous findings on the characterization of workflow use cases [10], [41]. Regarding internal validity, the different geographical regions and different week days we conducted our measurements on could have an impact. While we repeat each experiment six times to obtain stable results, there could be performance variability based on the time of day. However, systematically investigating this is beyond the scope of our work.

VIII. RELATED WORK

Multiple benchmark suites have been proposed to cover different aspects of serverless computing, from microarchitecture to the application level [11], [12], [17], [18], [59]–[67]. Kousiouris et al. [68] use microbenchmarks to estimate the overhead of orchestration in OpenWhisk. López et al. [69] investigate the orchestration overhead with microbenchmarks of

function chains and parallel functions. Shahidi et al. [70] evaluate the performance and cost of two stateful workflows on AWS and Azure. Barcelona-Pons et al. [71] use a microbenchmark to test the performance of fork-join parallelism in workflow orchestrators. Other performance studies of serverless focus on non-workflow orchestration [65], [72], [73].

Wen et al. [74] conducts a performance investigation of serverless workflows using two applications and microbenchmarks with varying numbers of functions, payload size, and parallelism. While they measure the execution time and estimate overhead, they do not evaluate scalability, billing, or investigate overhead sources. Instead, we focus on a wider collection of applications and propose a unifying model that allows developers to deploy and evaluate a single implementation across many cloud platforms. Moreover, we make all benchmark codes available and provide a ready-to-use benchmarking platform. Finally, we evaluated serverless Google Cloud Workflows instead of the non-serverless Google Cloud Composer.


Other authors analyzed the productivity of workflow languages and proposed alternative models. AFCL [75] is a custom and provider-independent orchestration language for serverless workflows, implemented on top of AWS Step Functions and IBM Composer. Burckhardt et al. explore the semantics of Durable Functions [23] and propose Netherite [76], a new engine to replace Azure Durable Functions.

IX. CONCLUSIONS

We propose SeBS-Flow, the first benchmark suite for serverless workflows. We follow the established benchmark design principles: introduce a platform-agnostic workflow model, propose a collection of six representative applications, and integrate them into an existing benchmark suite to ensure reproducibility and ease of use. We support the three major cloud providers, and benchmarks can be ported to other services by implementing a single interface transcribing our model to

the cloud-specific interface. We conduct a comprehensive and long-term evaluation of the performance and cost of proposed benchmark applications, investigating factors influencing the runtime and variance: cold startups, noise, scheduling, and the storage I/O. With the new benchmark suite, we enable benchmarking of the same workflow on different platforms, providing software developers and researchers with valuable insights regarding their different behaviors and properties.

ACKNOWLEDGMENTS

Larissa Schmid is supported by the pilot program Core Informatics of the Helmholtz Association (HGF).  This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreement PSAP, No. 101002047). We would also like to thank the Swiss National Supercomputing Centre (CSCS) for providing us with access to their HPC machine Ault. We thank Amazon Web Services for supporting this research with credits through the AWS Cloud Credit for Research, and Google Cloud Platform through the Google Cloud Research Credits program with the award GCP19980904.

AUTHORSHIP STATEMENT

The authors contributed to the paper as follows: M. Copik, A. Calotoiu, and T. Hoefler conceived the initial idea and L. Schmid, M. Copik, A. Calotoiu, and T. Hoefler designed the study; L. Brandner implemented the initial model, and L. Schmid extended and formalized it; L. Brandner implemented the benchmarks and L. Schmid and M. Copik extended and improved the implementation; L. Schmid and M. Copik collected data; L. Schmid, M. Copik, and L. Brandner analyzed and interpreted the results; L. Schmid and M. Copik conducted the literature study; L. Schmid and M. Copik wrote the draft manuscript; and L. Schmid, M. Copik, A. Calotoiu, A. Koziolok, and T. Hoefler reviewed and revised the manuscript.

REFERENCES

- [1] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021.
- [2] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of function-as-a-service software development in industrial practice," *Journal of Systems and Software*, vol. 149, pp. 340–359, 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121218302735>
- [3] Datadog, "The state of serverless," <https://www.datadoghq.com/state-of-serverless/>, 2024, accessed: 2024-01-28.
- [4] A. Mahgoub, L. Wang, K. Shankar, Y. Zhang, H. Tian, S. Mitra, Y. Peng, H. Wang, A. Klimovic, H. Yang *et al.*, "{SONIC}: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 285–301.
- [5] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, pp. 923–935.
- [6] M. Copik, M. Chrapek, L. Schmid, A. Calotoiu, and T. Hoefler, "Software resource disaggregation for hpc with serverless computing," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 139–156. [Online]. Available: <https://doi.ieeeecomputersociety.org/10.1109/IPDPS57955.2024.00021>

- [7] "AWS Step Functions," <https://aws.amazon.com/step-functions/>, 2016, accessed 25-01-2024.
- [8] "Azure Durable Functions," <https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview>, 2019, accessed 25-01-2024.
- [9] "Google Cloud Workflows," <https://cloud.google.com/workflows>, 2020, accessed 25-01-2024.
- [10] S. Eismann, J. Scheuner, E. v. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 4152–4166, 2022.
- [11] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 64–78. [Online]. Available: <https://doi.org/10.1145/3464298.3476133>
- [12] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "Faasdom: a benchmark suite for serverless computing," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 73–84. [Online]. Available: <https://doi.org/10.1145/3401025.3401738>
- [13] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 133–145.
- [14] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the planet of serverless computing: A systematic review," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, jul 2023. [Online]. Available: <https://doi.org/10.1145/3579643>
- [15] G. Adzic and R. Chatley, "Serverless computing: economic and architectural impact," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 884–889. [Online]. Available: <https://doi.org/10.1145/3106237.3117767>
- [16] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0164121220301527>
- [17] N. Somu, N. Daw, U. Bellur, and P. Kulkarni, "Panopticon: A comprehensive benchmarking tool for serverless applications," in *2020 International Conference on COMMunication Systems NETWORKS (COMSNETS)*, 2020, pp. 144–151.
- [18] J. Kim and K. Lee, "FunctionBench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, Jul. 2019. [Online]. Available: <https://doi.org/10.1109/cloud.2019.00091>
- [19] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. v. Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tüma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1528–1543, 2021.
- [20] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, "The fair guiding principles for scientific data management and stewardship," *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.
- [21] J. Wen, Z. Chen, Y. Liu, Y. Lou, Y. Ma, G. Huang, X. Jin, and X. Liu, "An empirical study on challenges of application development in serverless computing," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 416–428. [Online]. Available: <https://doi.org/10.1145/3468264.3468558>
- [22] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Rocalaberia, and A. Arjona, "Toward multicloud access transparency in serverless computing," *IEEE Software*, vol. 38, no. 1, pp. 68–74, 2021.
- [23] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: Semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: <https://doi.org/10.1145/3485510>
- [24] J. Wen, Z. Chen, and X. Liu, "Software engineering for serverless computing," *arXiv preprint arXiv:2207.13263*, 2022.

- [25] “AWS Step Functions Pricing,” <https://aws.amazon.com/step-functions/pricing/>, [Online; accessed 1 August 2024].
- [26] “Azure Functions Pricing,” <https://azure.microsoft.com/en-us/pricing/details/functions/>, [Online; accessed 1 August 2024].
- [27] “Google Cloud Workflows Pricing,” <https://cloud.google.com/workflows/pricing>, [Online; accessed 1 August 2024].
- [28] “Cloud Functions Pricing,” <https://cloud.google.com/functions/pricing>, [Online; accessed 14 July 2024].
- [29] “AWS Lambda Pricing,” <https://aws.amazon.com/lambda/pricing/>, [Online; accessed 1 August 2024].
- [30] N. Trcka, W. Aalst, van der, and N. Sidorova, *Analyzing control-flow and data-flow in workflow processes in a unified way*, ser. Computer science reports. Technische Universiteit Eindhoven, 2008.
- [31] J. L. Peterson, “Petri nets,” *ACM Comput. Surv.*, vol. 9, no. 3, p. 223–252, sep 1977. [Online]. Available: <https://doi.org/10.1145/356698.356702>
- [32] L. Brandner, “A platform-agnostic model and benchmark suite for serverless workflows,” Master Thesis, ETH Zurich, Zurich, 2022.
- [33] “Iterating a Loop Using Lambda,” <https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-create-iterate-pattern-section.html>, [Online; accessed 1 August 2024].
- [34] “Invoke Cloud Functions or Cloud Run,” <https://cloud.google.com/workflows/docs/calling-run-functions>, [Online; accessed 1 August 2024].
- [35] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao, “How to build a benchmark,” in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’15. New York, NY, USA: ACM, 2015, pp. 333–336. [Online]. Available: <http://doi.acm.org/10.1145/2668930.2688819>
- [36] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, “How is the weather tomorrow?: Towards a benchmark for the cloud,” in *Proceedings of the Second International Workshop on Testing Database Systems*, ser. DBTest ’09. New York, NY, USA: ACM, 2009, pp. 9:1–9:6. [Online]. Available: <http://doi.acm.org/10.1145/1594156.1594168>
- [37] T. Hoefler and R. Belli, “Scientific Benchmarking of Parallel Computing Systems.” ACM, Nov. 2015, pp. 73:1–73:12, proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).
- [38] R. Chatley and T. Allerton, “Nimbus: improving the developer experience for serverless applications,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 85–88. [Online]. Available: <https://doi.org/10.1145/3377812.3382135>
- [39] S. Ristov, P. Gritsch, D. Meyer, and M. Felderer, “Gospeechless: Interoperable serverless ml-based cloud services,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 394–395. [Online]. Available: <https://doi.org/10.1145/3639478.3643123>
- [40] “Redis,” <https://redis.io/>, Online; accessed 3 June 2024.
- [41] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, “A review of serverless use cases and their characteristics,” *arXiv preprint arXiv:2008.11110*, 2020.
- [42] “vSwarm - Serverless Benchmarking Suite,” <https://github.com/ease-lab/vSwarm>, [Online; accessed 27. July 2024].
- [43] S. Ren, K. He, R. B. Girshick, and J. Sun, “Faster R-CNN: towards real-time object detection with region proposal networks,” *CoRR*, vol. abs/1506.01497, 2015. [Online]. Available: <http://arxiv.org/abs/1506.01497>
- [44] C. McCaffrey, “Applying the saga pattern,” <https://www.youtube.com/watch?v=xDuwrwYHu8>, 2015, accessed: 2024-08-02.
- [45] “Step functions workflow collection: Saga pattern,” <https://github.com/aws-samples/step-functions-workflows-collection/tree/main/saga-pattern-tf>, 2023, accessed: 2024-08-02.
- [46] H. Garcia-Molina and K. Salem, “Sagas,” *SIGMOD Rec.*, vol. 16, no. 3, p. 249–259, dec 1987. [Online]. Available: <https://doi.org/10.1145/38714.38742>
- [47] “A MapReduce Overview,” <https://towardsdatascience.com/a-mapreduce-overview-6f2d64d8d0e6>, [Online; accessed 27. July 2024].
- [48] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, “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)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376. [Online]. Available: <https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi>
- [49] S. Fouladi, F. Romero, D. Iyer, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, “From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 475–488. [Online]. Available: <http://www.usenix.org/conference/atc19/presentation/fouladi>
- [50] J. C. Platt, “Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods,” in *ADVANCES IN LARGE MARGIN CLASSIFIERS*. MIT Press, 1999, pp. 61–74.
- [51] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: <http://dx.doi.org/10.1023/A%3A1010933404324>
- [52] “Memory and computing power,” <https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html>, Online; accessed 17 July 2024.
- [53] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, “Netgauge: A Network Performance Measurement Framework,” in *Proceedings of High Performance Computing and Communications, HPCC’07*, vol. 4782. Springer, 9 2007, pp. 659–671.
- [54] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, “Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions,” *Future Generation Computer Systems*, vol. 110, pp. 502–514, 2020. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X1730047X>
- [55] A. John, K. Ausmees, K. Muenzen, C. Kuhn, and A. Tan, “Sweep: Accelerating scientific research through scalable serverless workflows,” in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC ’19 Companion. New York, NY, USA: Association for Computing Machinery, 2019, p. 43–50. [Online]. Available: <https://doi.org/10.1145/3368235.3368839>
- [56] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, “Serverless execution of scientific workflows,” in *Service-Oriented Computing*, M. Maximilien, A. Vallecillo, J. Wang, and M. Oriol, Eds. Cham: Springer International Publishing, 2017, pp. 706–721.
- [57] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, “Mashup: Making serverless computing useful for hpc workflows via hybrid execution,” in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 46–60. [Online]. Available: <https://doi.org/10.1145/3503221.3508407>
- [58] R. B. Roy, T. Patel, and D. Tiwari, “Daydream: Executing dynamic scientific workflows on serverless platforms with hot starts,” in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–18.
- [59] D. Ustiugov, T. Amariuca, and B. Grot, “Analyzing tail latency in serverless clouds with stellar,” in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 51–62.
- [60] M. Shahradd, J. Balkind, and D. Wentzlaff, “Architectural implications of function-as-a-service computing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1063–1075. [Online]. Available: <https://doi.org/10.1145/3352460.3358296>
- [61] “Faastest,” <https://github.com/nuweba/faasbenchmark>, 2020, accessed: 2020-08-01.
- [62] “Faasdom,” <https://github.com/faas-benchmarking/faasdom>, 2020, accessed: 2020-08-01.
- [63] T. Back and V. Andrikopoulos, “Using a microbenchmark to compare function as a service solutions,” in *Service-Oriented and Cloud Computing*. Springer International Publishing, 2018, pp. 146–160. [Online]. Available: https://doi.org/10.1007/978-3-319-99819-0_11
- [64] A. Das, S. Patterson, and M. Wittie, “EdgeBench: Benchmarking edge computing platforms,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, Dec. 2018. [Online]. Available: <https://doi.org/10.1109/ucc-companion.2018.00053>
- [65] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bernbach, “Befaas: An application-centric benchmarking framework for faas platforms,” in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, 2021, pp. 1–8.
- [66] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, “Faasdom,” *Proceedings of the 14th ACM International Conference on Distributed*

and *Event-based Systems*, Jul 2020. [Online]. Available: <http://dx.doi.org/10.1145/3401025.3401738>

- [67] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 30–44. [Online]. Available: <https://doi.org/10.1145/3419111.3421280>
- [68] G. Kousiouris, C. Giannakos, K. Tserpes, and T. Stamati, "Measuring baseline overheads in different orchestration mechanisms for large faas workflows," in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 61–68. [Online]. Available: <https://doi.org/10.1145/3491204.3527467>
- [69] P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, A. Ruiz Ollobarren, and D. Arroyo Pinto, "Comparison of faas orchestration systems," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 148–153.
- [70] N. Shahidi, J. R. Gunasekaran, and M. T. Kandemir, "Cross-platform performance evaluation of stateful serverless workflows," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 63–73.
- [71] D. Barcelona-Pons, P. García-López, A. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas, "Faas orchestration of parallel workloads," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–30. [Online]. Available: <https://doi.org/10.1145/3366623.3368137>
- [72] J. Scheuner, S. Eismann, S. Talluri, E. van Eyk, C. Abad, P. Leitner, and A. Iosup, "Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," 2022.
- [73] R. Hancock, S. Udayashankar, A. J. Mashtizadeh, and S. Al-Kiswany, "Orcbench: A representative serverless benchmark," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 103–108.
- [74] J. Wen and Y. Liu, "A measurement study on serverless workflow services," in *2021 IEEE International Conference on Web Services (ICWS)*. Los Alamitos, CA, USA: IEEE, sep 2021, pp. 741–750.
- [75] S. Ristov, S. Pedratscher, and T. Fahringer, "Afcl: An abstract function choreography language for serverless workflow specification," *Future Generation Computer Systems*, vol. 114, pp. 368–382, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167739X20302648>
- [76] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proc. VLDB Endow.*, vol. 15, no. 8, p. 1591–1604, apr 2022. [Online]. Available: <https://doi.org/10.14778/3529337.3529344>