SeBS-Flow: Benchmarking Serverless Cloud Function Workflows

Larissa Schmid
Karlsruhe Institute of Technology
Germany

Marcin Copik
ETH Zurich
Switzerland

Alexandru Calotoiu
ETH Zurich
Switzerland

Laurin Brandner
ETH Zurich
Switzerland

Anne Koziolek
Karlsruhe Institute of Technology
Germany

Torsten Hoefler
ETH Zurich
Switzerland

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

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Networks** → **Cloud computing**; • **Software and its engineering** → **Software performance**; • **General and reference** → *Performance*; *Metrics*; *Measurement*; *Evaluation*.

Keywords: benchmark, serverless, function-as-a-service, faas, workflow, orchestration, serverless DAG

1 Introduction

Serverless computing gained major adoption in the industry [29, 47], with 50-70% of cloud customers using serverless functions and containers [25]. 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 [29], 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 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, e.g., through optimized function placement, oversubscription, targeting idle resources, and co-locating functions that depend on each other [10, 22, 48].

Workflows have been adopted by the most popular commercial cloud platforms [1, 2, 4] and make up almost a third of serverless applications [27]. However, just like every FaaS platform is different [23], serverless workflows are quite distinct from each other. Not only the different APIs and

Papers	Total	Benchmarks						Platforms					Artifact?
		Micro	WebApp	Multimedia	Data Proc.	ML	Scientific	AWS	Azure	GCP	Other	Research	
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 1. Analysis of 72 research papers on serverless workflows with benchmarks.

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 2. Key features of serverless workflows platforms.

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 2.1). Even though FaaS platforms might seem like the same product, they offer drastically different performance, reliability, and cost [23, 50, 73]. 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 [74]. 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 [9], with only limited support for testing [47].

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 1). We found that publications use different applications to benchmark the performance of new ideas, do not cover the same classes of 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 [65]. Benchmarking suites and systems have been proposed for FaaS [23, 43, 50, 68], 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 [54, 65] 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 3). We design the **benchmarking suite** (Section 4) and include **six workflow benchmarks** based on solutions common in research and industry (Section 5). 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 6) and use our benchmarking suite to comprehensively evaluate the three major cloud workflow services (Section 7). We follow the FAIR principle [78] 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.

2 Background

Serverless workflows introduce multiple new challenges to the software development process due to differences in the workflows platforms (Section 2.1). To model workflows, we use the formalism and semantics of Petri Nets (Section 2.2).

2.1 Developing Serverless Workflows

While software engineers are increasingly interested in serverless applications [76], 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 [63, 76]: Workflows have been adopted by all major cloud providers, but their implementations are significantly different in capabilities (Table 2). We focus on AWS Step Functions, Google Cloud Workflows, and Azure Durable Functions, as they play a leading role.

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 [76]. As the different implementations are all provider-specific, moving workflows from one platform to another is complicated, causing vendor lock-in [63]. Azure uses the programming model of Durable Functions [20], where the workflow definition is encoded within a regular program structure of an orchestrator. The graph of functions is expressed using a mainstream programming

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

(b) Google Cloud Workflows   **(c)** AWS Step Functions

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

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 3. Pricing according to vendors' documentation [12–14, 35, 36]. Orchestration per 1000 transitions.

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 3 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.

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 [47, 75]. The example 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

2.2 Workflow Nets

We base our model on workflow nets with data (WFD-nets) [70]. 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 [55] 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 [69]. WFD-nets [69] 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 enabled if tokens are in all its input places $\bullet t = \{p | (p, t) \in F\}$. When it fires, it removes the token(s)
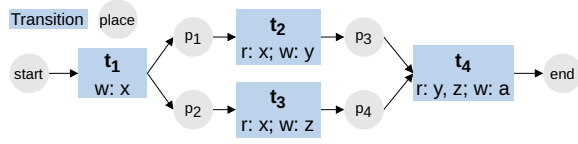
Figure 2. WFD-net with transitions $T = \{t_1, t_2, t_3, t_4\}$ and places $P = \{p_1, p_2, p_3, p_4, start, end\}$
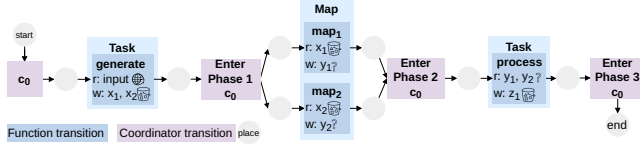

Figure 3. Workflow using our model based on WFD-nets.

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.

3 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 [69] (cf. Section 2.2) and extend them to be able to express the orchestration by the platform and how data is passed between functions.

3.1 Transitions

The set of transitions T is composed of two types, the co-ordinators 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.

3.2 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.
- 🗄 **NoSQL.** Data stored in NoSQL key-value storage provides low-latency data storage.
- 🌐 **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.

Formally, we define the set of resource annotations $A = \{o, n, p, t, r\}$ as additional element of the tuple of a WFD-net, with o representing data passing via the object storage, n via NoSQL, p via the invocation payload, t transparently, and r via reference. We define the corresponding resource annotation functions for reading and writing data as ra and rw as follows and also add them to the tuple of a WFD-net:

$$ra : \{(t, d) \in T \times D \mid d \in r(t)\} \to A$$

$$rw : \{(t, d) \in T \times D \mid d \in w(t)\} \rightarrow A$$

This means that each pair of a transition and a data element (t, d), with d being read or written by t, respectively, is assigned a resource annotation $a \in A$. By adding resource annotations, we do not change the behavior of the WFD-net. However, we enable checking the consistency of data accesses, for example, if the same data object is written and read using the same resource annotation.

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.

4 Workflows Benchmark Suite

We now present the design and implementation of SeBS-Flow[1]. 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 of their workflow services (Section 2.1). Thus, we define a platform-agnostic workflow definition (Section 4.1) based on our workflow model (Section 3). Then, we propose platform-specific generators that transcribe workflows to the respective proprietary definition of the desired platform (Section 4.2). We add the workflow representation and implementation to a serverless benchmark suite (Section 4.3).

4.1 Platform-Agnostic Workflow Definition

Our workflow model encodes the application as Petri Net (cf. Section 3). To define workflows in SeBS-Flow conforming to our model, we use a JSON syntax. Every phase has a type, relating to one of the available routing constructs (cf. Section 3.1). Coordinator transitions encode the order of phases, represented by the next field of phases that describes the consecutive step in the workflow. The next field refers to the phase name to be executed after, and the workflow terminates if this field is not set. Each phase receives the output payload of the previous function as input. This means that function implementations need to conform to the resource annotations as defined in the workflow model and download and upload data as needed accordingly. We encode the different phases as follows:

Task. A task executes a single serverless function, constituting a sequential routing. Listing 4a shows an example with the *compute_phase* executing the function *compute*.

[1]An extended definition and discussion of benchmarks can be found in the Master thesis [5].

```
"compute_phase": {
  "type": "task",
  "func_name": "compute"
}
```

(a) *Task* Statement.

```
"process_names": {
  "type": "map",
  "array": "customers",
  "root": "shorten",
  "next": "list_emails",
  "states": {
    "shorten": {
      "type": "task",
      "func_name": "short"
    }
  }
}
```

(b) *Map* Statement.

```
"root": "generate_phase",
"states": {
  "generate_phase": {
    "type": "task",
    "func_name": "generate",
    "next": "map_phase" },
  "map_phase": {
    "type": "map",
    "array": "x",
    "root": "map",
    "next": "process_phase",
    "states": {
      "map": {
        "type": "task",
        "func_name": "map" } } },
  "process_phase": {
    "type": "task",
    "func_name": "process"
  }
}
```

(c) Workflow from Figure 3.

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

Map. The map phase is a parallel routing construct and 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 4b 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 modeling a chain of tasks.

Switch. The switch phase is a conditional routing, deciding the next phase dynamically at runtime based on the given condition. The different cases are evaluated after another, with the first one fulfilling the condition being executed.

Parallel. This higher-level phase corresponds to a parallel routing and executes sub-workflows, consisting of any of the phases, concurrently.

We show an example of a complete workflow definition in Listing 4c, encoding the same workflow as shown in Figure 3. The root entry specifies the name of the phase that should be executed first, in this case the generate_phase. The states entry then contains all phases of the workflow. As mentioned above, each phase receives the output payload of the previous function as input. Therefore, the data movement is encapsulated in the functions and not controlled by the workflow orchestration. Only in the case of the map phase, we explicitly specify which array is used for distributing its elements to single functions. The level of parallelism

is then decided dynamically at runtime depending on the size of the given array.

4.2 Platform-Specific Transcription

We map the six phases building a serverless workflow to different features of the modeling language on each platform. We evaluate the overhead introduced by necessary adaptations to platforms in Section 6.

4.2.1 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 [11], 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.

4.2.2 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.

4.2.3 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.

4.3 Benchmark Suite

We follow standard design practices to build a new benchmark suite: it should be relevant, extensible, easy to use, and reproducible [17, 23, 39, 72]. Our suite is relevant as we include applications representing a variety of workloads in the industry and academia (Section 5). 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 [23], an established benchmark suite for FaaS: Benchmarks must be easy to deploy and execute to ensure their self-validation [72]. 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 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 [21, 59].

Serverless functions need cloud 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 [57] 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.

To implement a workflow using our model, a user has to provide the following (cf. Figure 5): First, the implementation of the workflow's functions in a language of their choice. Our workflow model is independent of the actual benchmark implementation. We can work with any language supported in the cloud, with currently supported Python, Node.js, C++, and Java through SeBS. Second, any data used as input to the workflow. Third, the specification how the functions should be orchestrated using our platform-agnostic workflow definition in JSON (cf. subsection 4.1). SeBS-Flow takes both as input and deploys the workflow with the functions to the respective cloud the user chooses, transcribing the workflow to their platform-specific representations by traversing the JSON file. This is transparent to the user and fully automated.

5 Benchmark Applications

In SeBS-Flow, we implement six benchmarks covering real-life workloads. Also, we implement four microbenchmarks
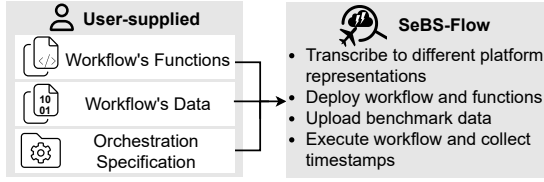
Figure 5. Process of executing a workflow using SeBS-Flow.

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 4. Key features of different benchmarks.

used in the evaluation: function chain, object storage performance, parallel invocations (Section 7.3.1), and selfish detour (Section 7.3.2). The selected benchmarks cover various domains that use workflows (Table 4), and correspond to previous findings on the characterization of workflow use cases [27, 28] regarding control-flow, number of functions, parallel invocations of the same functions, longer runtimes, and workload sizes: We include the sequential TripBooking benchmark (50% of workflows), four benchmarks using less than ten different functions (72% of workflows), five benchmarks involving parallel invocations of the same function (52% of workflows), and the 1000Genome workflow contains functions with a runtime of over a minute (25% of workflows). Moreover, the analysis by Eismann et al. [27] classifies 72% of serverless workflows as "small," consisting of 2 to 10 functions, 23% as "medium" with 11 to 1000 functions, and only 4% as "large" with more than 1000 functions. With our workloads, four of six applications can be classified as "small" and two more as "medium." Furthermore, the analysis of traces from production workloads on Azure Durable Functions [49] also showed that 40% of workflows are sequential, most workflows do not use more than ten functions, the median number of different functions in a workflow is three, and the median workflow execution time is 5.6 seconds, confirming the representativeness of our applications. We visualize only one of the benchmarks here, but provide figures for the other benchmarks in the supplementary material.

Video Analysis. The benchmark detects objects in a video, and parallelizes the sequential benchmark in vSwarm [8] (Figure 6). Functions decode video frames and apply the Faster R-CNN model [58]. 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 compute Y_i, all detections with confidence $p > 0.5$. Finally, detections are accumulated in acc, returning the final payload Y. We used
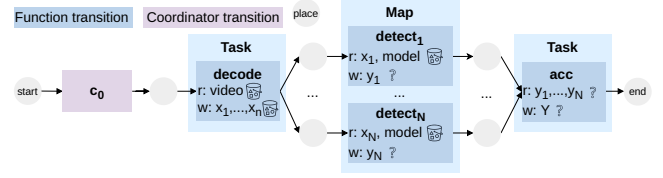

Figure 6. The Video Analysis benchmark.

$F = 10$ frames and batch size $B = 5$, yielding two parallel functions in the map phase.

Trip Booking. The benchmark represents web applications, and it mocks a common example of reserving a hotel, car rental, and flight [6, 53]. 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 [32] 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.

MapReduce. We base our example on prior implementations [8, 52] 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.

ExCamera. ExCamera [31] 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 [31] and the available implementation [30]. We use $M = 30$ total frames and a chunk size of $N = 6$, resulting in five parallel functions.

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 [56], and a Random Forest [18], creating two concurrent functions.

1000Genomes. This is a scientific workflow that identifies mutational overlaps using data from the 1000 Genomes project [7]. 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 Sorting Intolerant from Tolerant (SIFT) scores. In the last phase, `mutation_overlap` measures the overlap in Single Nucleotide Polymorphisms (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.

6 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 1, p. 2 for their categorization). We provide the complete list of papers and analysis results in the supplementary material.

6.1 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 4). 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. The third one uses a *switch* state requiring two conditions to be true. While we do not support transcribing this currently, transcription

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 within the scope of programming models not allowing for communication between functions and using orchestration based on dynamic characteristics of the system, and developers can use it to model and execute their workflows.

6.2 Overhead of our Model

To check if our model and transcription (cf. Sec. 4.2) 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 currently do not transcribe this choice type and are therefore not able to generate the same state machine. However, if we would add the transcription, the resulting state machine would look similar. 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, which does not affect the overhead. 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, with the workflow's median runtime being 3757.55 seconds. 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.

6.3 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 the results included relevant papers we knew as a gold standard [26]. 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.

7 Evaluation of Cloud Services

We use SeBS-Flow to evaluate three major cloud workflow services – AWS Step Functions, Google Cloud Workflows, and Azure Durable Functions – 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 overheads in the orchestration?
RQ2.2 What causes variations in the critical path?
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?

7.1 Methodology

We deploy benchmarks and resources used by them on Azure to the *europe-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 [27]. We check how often we should repeat experiments by computing nonparametric confidence intervals on the measurements of the MapReduce benchmark and aim at being in a 5% interval of the median using 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.

7.2 RQ1: Runtime Differences among Platforms

We compare the runtime of each benchmark on the selected platforms. We calculate the runtime by subtracting the first *start* timestamp from the last *end* timestamp. The results in Figure 7 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

(a) Video Analysis, 2048MB.

(b) ExCamera, 256MB.

(c) MapReduce, 256MB.

(d) Trip Booking, 128MB.

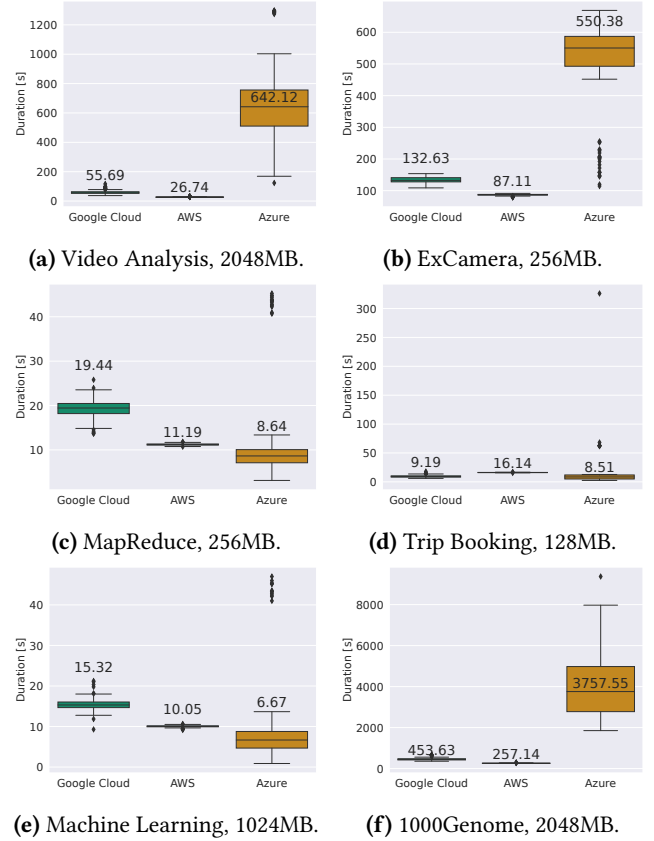**(e)** Machine Learning, 1024MB.

(f) 1000Genome, 2048MB.

Figure 7. Runtime of benchmark applications on AWS Step Functions, GC Workflows, and Azure Durable, *burst* invocations.

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.

7.3 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 to investigate the reasons behind this: 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 T_O by subtracting the critical path T_C from the total runtime. Figure 8 shows the critical path and overhead for all

(a) Video Analysis, 2048MB.

(b) ExCamera, 256MB.

(c) MapReduce, 256MB.

(d) Trip Booking, 128MB.

(e) Machine Learning, 1024MB.
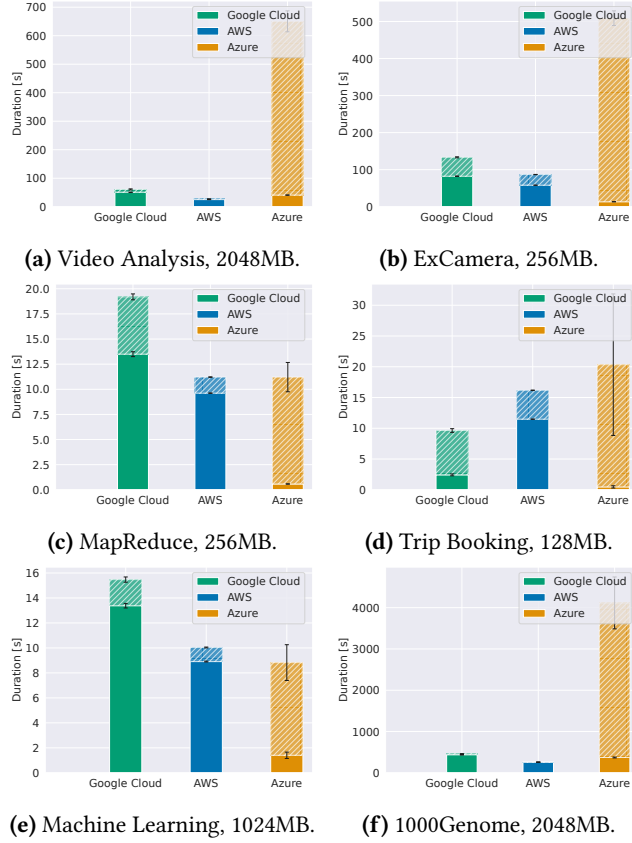

(f) 1000Genome, 2048MB.

Figure 8. Critical path (opaque) and overhead (hatched) of different benchmarks on considered platforms, *burst* invocations.

benchmarks. Azure's runtime is dominated by highly variable scheduling overhead: For example, the overhead of the ExCamera benchmark is, on average, 495.5s, more than 36× as long as its critical path of 13.5s. The ML benchmark incurs the least overhead of 5× 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 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. We therefore explore different causes for the differing behavior of the cloud platforms in the following Sections 7.3.1 and 7.3.2.

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

Cloud Storage I/O. The data downloaded from the object storage differs between benchmarks (Table 4, p. 7), with hundreds of megabytes in ExCamera, 1000Genomes, and Video Analysis. These benchmarks experience the highest relative

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

(b) Invocation latency, chain of 10 functions, $2^5 \leq M < 2^{18}$, 256MB, *warm* invocations.

Figure 9. Analysis of different sources of overhead.

overhead of 36.7×, 10×, and 14.95× 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 9a 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, however, we observe an overhead of almost 149 and 4.9 seconds for 128 and 1 MB files, respectively. Therefore, data downloads can account for a significant part of the large overhead measured on Azure Durable.

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 this by executing a microbenchmark that spawns N functions in parallel, each one sleeping for T seconds, and start 30 such invocations concurrently. Figure 10 shows the relative overhead of the actual runtime of the workflow compared to the function execution time. AWS functions demonstrate modest overhead, with largest values for the shortest duration. The absolute overhead incurred for a certain number of parallel functions remains relatively constant, causing the relative overhead to decrease with higher function executin times. GC functions present a larger relative 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 relative 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
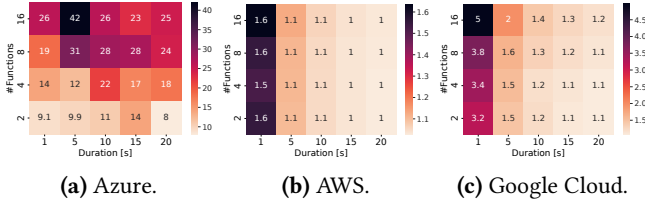
(a) Azure. **(b)** AWS. **(c)** Google Cloud.

Figure 10. The overhead of parallel sleep microbenchmark, $2 \leq N \leq 16$, $1 \leq T \leq 20$, 256MB, *burst* invocations.

behavior in Figure 11. 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.

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 9b 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.

Conclusions. The microbenchmarks demonstrate that a significant part of the overhead observed on Azure originates from the parallel schedules and storage I/O. Moreover, the return payload can be a source of overhead on Azure for larger payloads. To minimize overheads, workflows downloading large amounts of data, using high levels of parallelism, and high return payloads may therefore better be deployed to AWS or Google Cloud, with AWS demonstrating less overheads and better scalability across our benchmarks.

7.3.2 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.

OS Noise. The cloud provider controls the CPU allocation to a serverless function, either in relation to the memory configuration on AWS and GCP [35, 46], or in an undisclosed fashion on Azure. We use the selfish detour benchmark to

quantify OS noise [40], which allows us to estimate how long the function is suspended by the OS, which in turn approximates 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 13a 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 13b) and Machine Learning (Figure 13c). The overall trend observed in Section 7.2 remains unchanged: Google Cloud demonstrates the longest critical path duration. The suspension time explains the shorter critical path on Azure as compared to AWS and GCP for benchmarks with low-memory configurations: Azure functions receive larger CPU allocations.

Cold Starts. Cold invocations add significant overhead to the function execution [23]. Table 5 shows the frequency of cold starts in our measurements, with cold starts identified using the containerID (see Section 4.3). Azure Durable performs significantly better, experiencing almost no cold starts, likely because function apps on Azure can hold many invocations concurrently [23]. While the low scalability causes high orchestration overheads, it benefits the computations by putting them in warm containers. Figure 12 shows the impact of cold starts on the critical path and overhead. We show only the duration of warm starts on Azure Durable, as all benchmarks show a very low amount of cold starts on Azure. On AWS and GCP, however, there is a high percentage of cold starts in our measurement data. We therefore collected another 60 workflow invocations 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× and 4.5× 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.

Conclusions. Our experiments show that Azure achieves short critical paths due to larger CPU allocations and less cold starts. To minimize the critical path, benchmarks using a low-memory configuration and only being executed occasionally can therefore be deployed to Azure. High-memory configurations, however, get higher CPU shares on AWS and GCP. GCP shows the slowest critical path even for warm
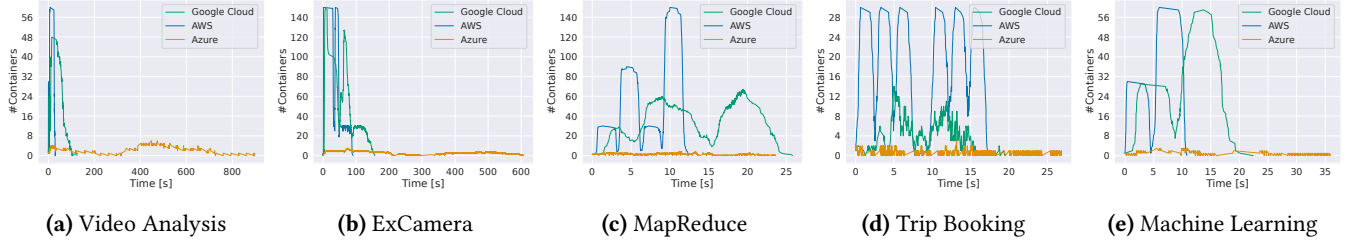
(a) Video Analysis   **(b)** ExCamera   **(c)** MapReduce   **(d)** Trip Booking   **(e)** Machine Learning

Figure 11. Scaling profiles: the number of distinct containers used for 30 consecutive workflow invocations.

	Cold starts			State transitions	
Benchmark	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 5. Relative #cold starts and #state transitions.

(a) Machine Learning, 1024MB.   **(b)** MapReduce, 256MB.
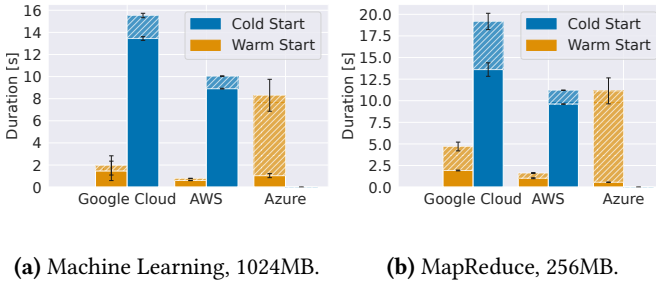
Figure 12. Critical path (opaque) and overhead (hatched) of warm and cold invocations.

invocations, while AWS can be competitive to Azure, making it a good choice for frequently executed workflows.

7.4 RQ3: Usability for Scientific Workflows

There is rising interest in the scientific community to use serverless solutions [28], accompanied by experimentation with serverless offerings of the platforms [51] and management systems for serverless execution of workflows [41, 42, 61, 62]. However, they do not consider the workflow orchestration systems 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 14a. 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 14b 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 w.r.t. 5 and 10 jobs, respectively. The cloud platforms achieve a nearly-optimal speedup, which is not surprising given the high overhead for the baseline execution.

7.5 RQ4: Pricing

We compare the average cost of executing a workflow and estimate the prices, as shown in Table 3, p. 3. Functions invoked during the execution of a workflow are billed based on the integral of memory and duration. Figure 15 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× 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× more: the AWS state language requires fewer states to implement the benchmarks (Table 5). Overall, we observe that Azure is expensive for workflows where it is also the slowest platform, such as for 1000Genome, but offers the cheapest pricing for benchmarks it also executes fastest, such as ML and MapReduce. Contrary to that, AWS shows high pricing for benchmarks it is fastest, such as Video Analysis and ExCamera

In addition to execution and orchestration costs, workflows generate charges when accessing the object and NoSQL storage. While the prices of read and write operations on the
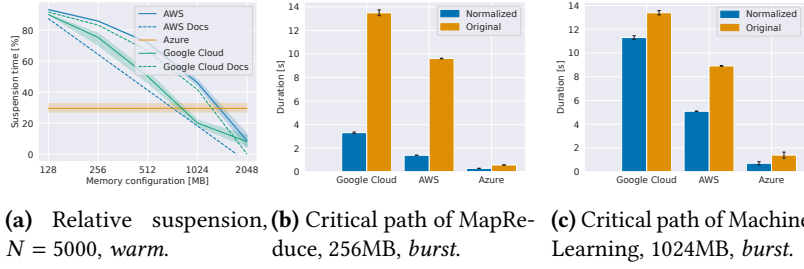
(a) Relative suspension, $N = 5000$, *warm.*

(b) Critical path of MapReduce, 256MB, *burst.*

(c) Critical path of Machine Learning, 1024MB, *burst.*

Figure 13. Analysis of OS noise.

(a) Complete workflow.

(b) `individuals`.

Figure 14. Scalability of 1000Genome workflow.

object storage are the same across clouds, the billing models for key-value storage differ: DynamoDB charges 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 this, 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 ¢0.68 and ¢1.08 per 1000 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 ¢2.4. There, the estimated cost of CosmosDB request units is equal to ¢0.68 and adds 28.5% of workflow price.

7.6 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 *europe-west* for Azure, *europe-west-1* for GCP, and *us-east-1* for AWS. The 2024 invocations are run in the same regions, except for GCP in *us-east1*, and use Python 3.8. Figure 16 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 for MapReduce is the same in 2024 as in 2022, the overhead of ML has been approximately halved from 2022 to 2024.

7.7 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 [27, 28]. 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
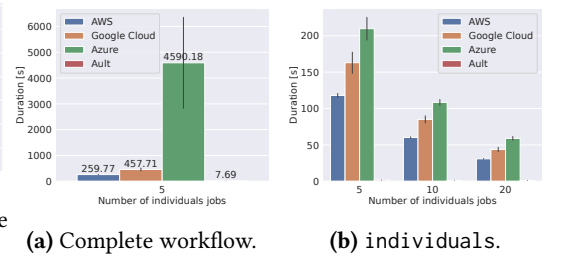
times to obtain stable results, there could be performance variability based on the time of day. However, systematically investigating this is beyond the scope of our work.

8 Related Work

Multiple benchmark suites have been proposed to cover different aspects of serverless computing, from microarchitecture to the application level [3, 15, 23, 43, 50, 67, 71]. However, all of them consider only the execution of single functions. Das et al. [24] benchmark serverless edge computing platforms. Other performance studies of serverless applications focus on non-workflow orchestration systems, e.g., using cloud storage and queue triggers [37, 38, 64, 68]. Grambow et al. [37] propose BeFaaS, providing an application benchmark modeling an online shop where the functions communicate using synchronous and asynchronous calls. In contrast, SeBS-Flow targets serverless workflow orchestrations.

ServerlessBench [79] considers a function chain microbenchmark orchestrated by AWS Step Functions, but only measures runtime of the workflow and time in between function invocations for varying payload sizes. Kousiouris et al. [44] use microbenchmarks to estimate the overhead of orchestration in OpenWhisk. López et al. [33] investigate the orchestration overhead with microbenchmarks of function chains and parallel functions. Shahidi et al. [66] evaluate the performance and cost of two stateful workflows on AWS and Azure. Barcelona-Pons et al. [16] use a microbenchmark to test the performance of fork-join parallelism in workflow orchestrators. With SeBS-Flow, we provide not only microbenchmarks, but also six applications from different domains that can automatically be deployed to different cloud platforms. Based on these benchmarks, we present a broader evaluation of the performance of cloud platforms.

Wen et al. [77] 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

(a) Video Analysis **(b)** MapReduce **(c)** Trip Booking **(d)** ExCamera **(e)** Machine Learning **(f)** 1000Genome

Figure 15. Price per 1000 workflow executions: function costs are opaque and state transition costs are translucent.
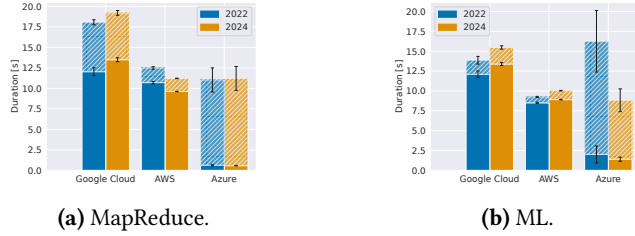

(a) MapReduce.   **(b)** ML.

Figure 16. Comparison of critical path (opaque) and over-head (hatched) between 2022 and 2024, *burst* invocations.

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. XFBench [45] provides chaining of different functions and deploying them to AWS Step Functions and Azure Durable Functions, while we focus on realistic and complete applications. Moreover, they do not consider cloud-native data movement between functions via cloud storage, do not evaluate the overhead of their platform transcription, and can not compare pricing between platforms.

Other authors analyzed the productivity of workflow languages and proposed alternative models. AFCL [60] 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 [20] and propose Netherite [19], a new engine to replace Azure Durable Functions.

9 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 researches with valuable insights regarding their different behaviors and properties.

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. Koziolek, and T. Hoefler reviewed and revised the manuscript.

References

[1] 2016. AWS Step Functions. https://aws.amazon.com/step-functions/. Accessed 25-01-2024.

[2] 2019. Azure Durable Functions. https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview. Accessed 25-01-2024.

[3] 2020. FaaSTest. https://github.com/nuweba/faasbenchmark. Accessed: 2020-08-01.

[4] 2020. Google Cloud Workflows. https://cloud.google.com/workflows. Accessed 25-01-2024.

[5] 2022. A Platform-Agnostic Model and Benchmark Suite for Serverless Workflows. https://doi.org/10.3929/ethz-b-000574821. Master's Thesis.

[6] 2023. Step Functions Workflow Collection: Saga Pattern. https://github.com/aws-samples/step-functions-workflows-collection/tree/main/saga-pattern-tf. Accessed: 2024-08-02.

[7] Accessed 04-02-2025. 1000 Genomes Project. https://www.internationalgenome.org/.

[8] [Online; accessed 27. July 2024]. vSwarm - Serverless Benchmarking Suite. https://github.com/ease-lab/vSwarm.

[9] Gojko Adzic and Robert Chatley. 2017. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 884–889. https://doi.org/10.1145/3106237.3117767

[10] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.

[11] AWS [Online; accessed 1 August 2024]. Iterating a Loop Using Lambda. https://docs.aws.amazon.com/step-functions/latest/dg/tutorial-create-iterate-pattern-section.html.

[12] AWS Lambda Pricing [Online; accessed 1 August 2024]. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[13] AWS Step Functions Pricing [Online; accessed 1 August 2024]. AWS Step Functions Pricing. https://aws.amazon.com/step-functions/pricing/.

[14] Azure Functions Pricing [Online; accessed 1 August 2024]. Azure Functions Pricing. https://azure.microsoft.com/en-us/pricing/details/functions/.

[15] Timon Back and Vasilios Andrikopoulos. 2018. Using a Microbenchmark to Compare Function as a Service Solutions. In *Service-Oriented and Cloud Computing*. Springer International Publishing, 146–160. https://doi.org/10.1007/978-3-319-99819-0_11

[16] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. 2019. FaaS Orchestration of Parallel Workloads. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 25–30. https://doi.org/10.1145/3366623.3368137

[17] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. 2009. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. In *Proceedings of the Second International Workshop on Testing Database Systems* (Providence, Rhode Island) *(DBTest '09)*. ACM, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/1594156.1594168

[18] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324

[19] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. 2022. Netherite: Efficient Execution of Serverless Workflows. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1591–1604. https://doi.org/10.14778/3529337.3529344

[20] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable Functions: Semantics for Stateful Serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (oct 2021), 27 pages. https://doi.org/10.1145/3485510

[21] Robert Chatley and Thomas Allerton. 2020. Nimbus: improving the developer experience for serverless applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 85–88. https://doi.org/10.1145/3377812.3382135

[22] M. Copik, M. Chrapek, L. Schmid, A. Calotoiu, and T. Hoefler. 2024. Software Resource Disaggregation for HPC with Serverless Computing. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 139–156. https://doi.org/10.1109/IPDPS57955.2024.00021

[23] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[24] Anirban Das, Stacy Patterson, and Mike Wittie. 2018. EdgeBench: Benchmarking Edge Computing Platforms. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. https://doi.org/10.1109/ucc-companion.2018.00053

[25] Datadog. 2024. The State of Serverless. https://www.datadoghq.com/state-of-serverless/. Accessed: 2024-01-28.

[26] Oscar Dieste, Anna Grimán, and Natalia Juristo. 2009. Developing search strategies for detecting relevant experiments. *Empirical Software Engineering* 14 (2009), 513–539.

[27] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2022. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Transactions on Software Engineering* 48, 10 (2022), 4152–4166. https://doi.org/10.1109/TSE.2021.3113940

[28] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2020. A review of serverless use cases and their characteristics. *arXiv preprint arXiv:2008.11110* (2020).

[29] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2021. Serverless Applications: Why, When, and How? *IEEE Software* 38, 1 (2021), 32–39. https://doi.org/10.1109/MS.2020.3023302

[30] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[31] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[32] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *SIGMOD Rec.* 16, 3 (dec 1987), 249–259. https://doi.org/10.1145/38714.38742

[33] Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Alvaro 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)*. 148–153. https://doi.org/10.1109/UCC-Companion.2018.00049

[34] Google Cloud [Online; accessed 1 August 2024]. Invoke Cloud Functions or Cloud Run. https://cloud.google.com/workflows/docs/calling-run-functions.

[35] Google Cloud [Online; accessed 14 July 2024]. Cloud Functions Pricing. https://cloud.google.com/functions/pricing.

[36] Google Cloud Workflows Pricing [Online; accessed 1 August 2024]. Google Cloud Workflows Pricing. https://cloud.google.com/

workflows/pricing.

[37] Martin Grambow, Tobias Pfandzelter, Luk Burchard, Carsten Schubert, Max Zhao, and David Bermbach. 2021. BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 1–8. https://doi.org/10.1109/IC2E52221.2021.00014

[38] Ryan Hancock, Sreeharsha Udayashankar, Ali José Mashtizadeh, and Samer Al-Kiswany. 2022. OrcBench: A Representative Serverless Benchmark. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 103–108. https://doi.org/10.1109/CLOUD55607.2022.00028

[39] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems. ACM, 73:1–73:12. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15).

[40] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Netgauge: A Network Performance Measurement Framework, In Proceedings of High Performance Computing and Communications, HPCC'07 (Houston, USA). 4782, 659–671.

[41] Qingye Jiang, Young Choon Lee, and Albert Y. Zomaya. 2017. Serverless Execution of Scientific Workflows. In *Service-Oriented Computing*, Michael Maximilien, Antonio Vallecillo, Jianmin Wang, and Marc Oriol (Eds.). Springer International Publishing, Cham, 706–721.

[42] Aji John, Kristiina Ausmees, Kathleen Muenzen, Catherine Kuhn, and Amanda Tan. 2019. SWEEP: Accelerating Scientific Research Through Scalable Serverless Workflows. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion* (Auckland, New Zealand) *(UCC '19 Companion)*. Association for Computing Machinery, New York, NY, USA, 43–50. https://doi.org/10.1145/3368235.3368839

[43] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. https://doi.org/10.1109/cloud.2019.00091

[44] George Kousiouris, Chris Giannakos, Konstantinos Tserpes, and Teta Stamati. 2022. Measuring Baseline Overheads in Different Orchestration Mechanisms for Large FaaS Workflows. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (Beijing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 61–68. https://doi.org/10.1145/3491204.3527467

[45] Varad Kulkarni, Nikhil Reddy, Tuhin Khare, Harini Mohan, Jahnavi Murali, Mohith A, Ragul B, Sanjai Balajee, Sanjjit S, Swathika D, Vaishnavi S, Yashasvee V, Chitra Babu, Abhinandan S. Prasad, and Yogesh Simmhan. 2024. XFBench: A Cross-Cloud Benchmark Suite for Evaluating FaaS Workflow Platforms. In *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 543–556. https://doi.org/10.1109/CCGrid59990.2024.00067

[46] lambda-vcpu Online; accessed 17 July 2024. Memory and Computing Power. https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html.

[47] 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. https://doi.org/10.1016/j.jss.2018.12.013

[48] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. 2021. {SONIC}: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 285–301.

[49] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 26 (June 2022), 28 pages. https://doi.org/10.1145/3530892

[50] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/3401025.3401738

[51] 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. https://doi.org/10.1016/j.future.2017.10.029

[52] MapReduce [Online; accessed 27. July 2024]. A MapReduce Overview. https://towardsdatascience.com/a-mapreduce-overview-6f2d64d8d0e6.

[53] Caitie McCaffrey. 2015. Applying the Saga Pattern. https://www.youtube.com/watch?v=xDuwrtwYHu8. Accessed: 2024-08-02.

[54] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, José Nelson Amaral, Petr Tůma, and Alexandru Iosup. 2021. Methodological Principles for Reproducible Performance Evaluation in Cloud Computing. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1528–1543. https://doi.org/10.1109/TSE.2019.2927908

[55] James L. Peterson. 1977. Petri Nets. *ACM Comput. Surv.* 9, 3 (sep 1977), 223–252. https://doi.org/10.1145/356698.356702

[56] John C. Platt. 1999. Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods. In *ADVANCES IN LARGE MARGIN CLASSIFIERS*. MIT Press, 61–74.

[57] Redis Online; accessed 3 June 2024. Redis. https://redis.io/.

[58] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR* abs/1506.01497 (2015). arXiv:1506.01497 http://arxiv.org/abs/1506.01497

[59] Sashko Ristov, Philipp Gritsch, David Meyer, and Michael Felderer. 2024. GoSpeechLess: Interoperable Serverless ML-based Cloud Services. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 394–395. https://doi.org/10.1145/3639478.3643123

[60] Sasko Ristov, Stefan Pedratscher, and Thomas Fahringer. 2021. AFCL: An Abstract Function Choreography Language for serverless workflow specification. *Future Generation Computer Systems* 114 (2021), 368–382. https://doi.org/10.1016/j.future.2020.08.012

[61] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2022. 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* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 46–60. https://doi.org/10.1145/3503221.3508407

[62] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. DayDream: Executing Dynamic Scientific Workflows on Serverless Platforms with Hot Starts. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–18. https://doi.org/10.1109/SC41404.2022.00027

[63] Josep Sampe, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. 2021. Toward Multicloud Access Transparency in Serverless Computing. *IEEE Software* 38, 1 (2021), 68–74. https://doi.org/10.1109/MS.2020.3029994

[64] Joel Scheuner, Simon Eismann, Sacheendra Talluri, Erwin van Eyk, Cristina Abad, Philipp Leitner, and Alexandru Iosup. 2022. Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications. arXiv:2205.07696 [cs.DC]

[65] Joel Scheuner and Philipp Leitner. 2020. Function-as-a-Service performance evaluation: A multivocal literature review. *Journal of Systems and Software* 170 (2020), 110708. https://doi.org/10.1016/j.jss.2020.

110708

[66] Narges Shahidi, Jashwant Raj Gunasekaran, and Mahmut Taylan Kandemir. 2021. Cross-Platform Performance Evaluation of Stateful Serverless Workflows. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 63–73. https://doi.org/10.1109/IISWC53511.2021.00017

[67] 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* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296

[68] N. Somu, N. Daw, U. Bellur, and P. Kulkarni. 2020. PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications. In *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*. 144–151.

[69] N. Trcka, W.M.P. Aalst, van der, and N. Sidorova. 2008. *Analyzing control-flow and data-flow in workflow processes in a unified way*. Technische Universiteit Eindhoven.

[70] Nikola Trčka, Wil M. P. van der Aalst, and Natalia Sidorova. 2009. Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. In *Advanced Information Systems Engineering*, Pascal van Eck, Jaap Gordijn, and Roel Wieringa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 425–439.

[71] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. 2021. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 51–62. https://doi.org/10.1109/IISWC53511.2021.00016

[72] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. 2015. How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (Austin, Texas, USA) *(ICPE '15)*. ACM, New York, NY, USA, 333–336. https://doi.org/10.1145/2668930.2688819

[73] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 133–145.

[74] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the Planet of Serverless Computing: A Systematic Review. *ACM Trans. Softw. Eng. Methodol.* 32, 5, Article 131 (jul 2023), 61 pages. https://doi.org/10.1145/3579643

[75] Jinfeng Wen, Zhenpeng Chen, and Xuanzhe Liu. 2022. Software engineering for serverless computing. *arXiv preprint arXiv:2207.13263* (2022).

[76] Jinfeng Wen, Zhenpeng Chen, Yi Liu, Yiling Lou, Yun Ma, Gang Huang, Xin Jin, and Xuanzhe Liu. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 416–428. https://doi.org/10.1145/3468264.3468558

[77] Jinfeng Wen and Yi Liu. 2021. A Measurement Study on Serverless Workflow Services. In *2021 IEEE International Conference on Web Services (ICWS)*. IEEE, Los Alamitos, CA, USA, 741–750. https://doi.org/10.1109/ICWS53863.2021.00102

[78] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific data* 3, 1 (2016), 1–9.

[79] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/3419111.3421280

A Artifact Appendix

A.1 Abstract

Our artifact contains the implementation of SeBS-Flow, data, and analysis scripts. We provide the following components:

- *sebs-flow-implementation* - Source code of the benchmark suite.
- *sebs-flow-artifact* - Benchmarking results obtained for the paper together with Python plotting and analysis scripts used for data analysis.

A.2 Description & Requirements

A.2.1 How to access.

https://doi.org/10.5281/zenodo.14809924

A.2.2 Hardware dependencies.
Unix system capable of sending HTTP requests to the cloud.

A.2.3 Software dependencies.

- Docker (at least 19)
- Python 3.7+ with `pip` and `venv`
- `libcurl` and its headers must be available to install `pycurl`
- Standard Linux tools and `zip` installed

A.2.4 Benchmarks.
We provide benchmarks and data used as part of the implementation in `serverless-benchmarks/benchmarks/600.workflows` and `serverless-benchmarks/benchmarks-data/600.workflows`.

A.3 Set-up

To install the benchmark suite with support for all platforms used for our evaluation, use `./install.py -aws -azure -gcp`. This will create a virtual environment in `python-venv`, and install necessary Python dependencies and third-party dependencies. To use SeBS-Flow, the new Python virtual environment has to be activated: `. python-venv/bin/activate`. To deploy benchmarks to a platform, account credentials musst be supplied. See `serverless-benchmarks/docs/platforms.md` for details.

For measuring the execution of serverless workflows, we use a *Redis* instance deployed on a VM in the same cloud region as the workflow and its resources. See `serverless-benchmarks/docs/workflows.md` for details.

A.4 Evaluation workflow

A.4.1 Major Claims.
RQ1: Runtime. We evaluate the runtime differences between platforms by executing Experiment E1. Results are shown in Figure 7 and 8 and discussed in Section 7.2.

RQ2.1: Orchestration Overhead. We evaluate the overheads caused by cloud storage I/O via executing Experiment E3 (Figure 8), by parallel scheduling via executing Experiment E4 (Figure 10), and by the return payload via Experiment E5 (Figure 9b).

RQ2.2: Critical Path Discrepancy. We evaluate how the critical path is impacted by OS noise via executing Experiment E6 (Figure 13) and using data from E1 (Figure 13b, 13c) and by cold starts by analyzing the results from E1.

RQ3: Usability for Scientific Workflows. We evaluate how well serverless workflow orchestrations are suited for execution of scientific benchmarks by comparing execution times and scaling of the workflow on cloud platforms and an HPC system using data from E1 for 1000Genomes and with Experiments E7 and E8 (Figure 14).

RQ4: Pricing. We evaluate the differences in pricing between the cloud platforms by comparing execution cost of our application benchmarks using data from E1 (Figure 15).

A.4.2 Experiments.
All configuration files used are provided per platform and benchmark executed as part of the artifact.

E1: Burst execution of application benchmarks. Execution of all application benchmarks. The paper uses 180 *burst* workflow executions per benchmark with 30 executions triggered at once.

E2: Warm execution of application benchmarks. Execution of Machine Learning and MapReduce benchmarks in *warm* mode. The paper collects 60 workflow executions with at least one *warm* function invocation per benchmark with 30 executions triggered at once.

E3: Parallel Download. Execution of the parallel download microbenchmark with 20 functions downloading a file in parallel, with filesizes from $2^{10}b$ to $2^{28}b$. The paper uses 30 *burst* executions triggered at once.

E4: Parallel Sleep. Execution of the parallel sleep microbenchmark with $2 \leq N \leq 16$ functions and sleep durations of $1 \leq T \leq 20s$. The paper uses 30 *burst* executions triggered at once.

E5: Function Chain. Execution of the function chain microbenchmark with 10 functions, with functions returning $2^5 \leq M2^{18}b$ sent to the next function. The paper uses 30 *warm* executions.

E6: OS Noise. Execution of the selfish detour microbenchmark collecting $N = 5000$ events, executed with memory configurations from 128MB to 2048MB. The paper uses 30 *warm* executions.

E7: Execution of 1000Genomes on HPC system. Execution of the 1000Genomes workflow on an HPC system. The paper uses five repetitions per configuration and a Intel(R) 6154@3.00GHz CPU. To reproduce the result that the execution on an HPC system is much faster, however, the workflow can be run in virtually every HPC environment.

E8: Scaling of `individuals` *task of 1000Genomes workflow.* Execution of the workflow `6101.1000-genome-individuals` with inputs *small-10* and *small-20*. The paper uses 180 *burst* workflow executions per input with 30 executions triggered at once.

A.5 Notes on Reusability

SeBS-Flow can be extended to evaluate workflow orchestrations on new serverless platforms. Moreover, new workflow benchmarks can be added to evaluate their performance on different platforms. Repetition of benchmarks over time can give insights into the evolution of performance.