sAirflow: Adopting Serverless in a Legacy Workflow Scheduler[⋆]

Filip Mikina[1], Pawel Zuk[2][0000−0002−4904−7171], and Krzysztof Rzadca[3][0000−0002−4176−853X]

[1] Institute of Informatics, University of Warsaw fmikina@gmail.com
[2] University of Southern California pawelzuk@isi.edu
[3] Institute of Informatics, University of Warsaw krzadca@mimuw.edu.pl

Abstract. Serverless clouds promise efficient scaling, reduced toil and monetary costs. Yet, serverless-ing a complex, legacy application might require major refactoring and thus is risky. As a case study, we use Airflow, an industry-standard workflow system. To reduce migration risk, we propose to limit code modifications by relying on change data capture (CDC) and message queues for internal communication. To achieve serverless efficiency, we rely on Function-as-a-Service (FaaS). Our system, sAirflow, is the first adaptation of the control plane and workers to the serverless cloud — and it maintains the same interface and most of the code. Experimentally, we show that sAirflow delivers the key serverless benefits: scaling and cost reduction. We compare sAirflow to MWAA, a managed (SaaS) Airflow. On Alibaba benchmarks on warm systems, sAirflow performs similarly while halving the monetary cost. On highly parallel workflows on cold systems, sAirflow scales out in seconds to 125 workers, reducing makespan by 2x-7x.

Keywords: Function-as-a-Service, FaaS, cloud applications, software migration

1 Introduction

Serverless cloud [8] products (AWS Lambda, GCP Cloud Run, Azure Functions, OpenWhisk) propose a new kind of contract between the provider and the customer. The customer supplies just the code of the function to execute, while the provider manages resources at the granularity of individual invocations. For customers, this reduces the toil of maintaining infrastructure, and often reduces monetary costs, as typically, the customer pays only for the consumed resources. The providers can optimize hardware while providing highly dynamic horizontal scaling: from zero when a function is not invoked to hundreds of concurrent invocations. This invocation-by-invocation management of resources *by providers* has received considerable attention [8]: e.g., avoiding cold-starts by optimizing the environment pre-warming and evictions [20]; reducing response latency in

[⋆] Accepted to Euro-Par 2024. Compared with the camera-ready version, this manuscript contains additional experimental results.

a warm system by changing how invocations are allocated to workers [5], or by more efficient scheduling at a worker node [36]. Comparatively, *applications* received less attention; and most of the serverless applications described in the literature [24] seem to be built from scratch. Yet, research and industry alike operate on proven, tested and well-understood legacy systems. Throwing away old code and starting from scratch is risky [10]. Yet, there are few blueprints for refactoring towards serverless [37].

The research question we address in this paper is: How can we effectively refactor a complex, legacy application to reap the benefits of serverless computing, including seamless scalability and cost-effectiveness, without introducing significant disruptions to its existing code structure?

As an example of a legacy application we take Airflow, an industry standard for authoring, scheduling, running and monitoring workflows, in particular, data processing pipelines [26]. Airflow is widely used in the industry, both on-premises and as a SaaS offering (AWS' MWAA, Google's Cloud Composer, and Azure's Data Factory Managed Airflow).

The primary challenge in refactoring Airflow lies in its reliance on a metadata database updated with SQL queries from many code locations. To overcome this obstacle, we utilize database-level change data capture (CDC) to transform metadata updates into events then transmitted to the control plane. This pattern allows us to transform Airflow's architecture into event-drive one. Functions from the original Airflow control plane execute as serverless functions (lambdas) that are triggered by events delivered through message queues. For example, when a user submits a new DAG, a CDC-triggered event invokes the scheduler. Similarly, we launch user-defined work on serverless offerings: a FaaS executor for shorter tasks (up to 15 min.); and a universal Container-as-a-Service (CaaS) executor. When a task ends, an event triggers a metadata update, that, in turn, triggers the scheduler. No sAirflow code continuously pulls or runs in the background.

The contribution of this paper is as follows: (1) We propose a new software design pattern for adapting to serverless legacy, database-driven applications: to minimize changes in the code, keep the database interactions; and rely on change data capture (CDC) to produce events driving the control plane. (2) sAirflow is the first serverless adaptation of Airflow's control plane and executors. This enables sAirflow to scale horizontally in seconds to 125 executors and to halve monetary costs. sAirflow thus efficiently surfaces through a legacy system the key advantages of FaaS: elasticity and usage-based pricing.

This paper is organized as follows. We start by reviewing related work in Section 2. We then describe Function as a Service (FaaS) and Airflow in Section 3. Section 4 describes the design and the key implementation details of sAirflow. Section 5 describes how sAirflow is deployed to the cloud; it also describes the evaluation method. Section 6 describes results of experiments comparing sAirflow with MWAA.

The source code is available at `https://github.com/fiffeek/beeflow` .

2 Related work

Workflow management systems (WMSs): [13] is a recent survey. Popular WMS include Airflow, Pegasus [9], FuncX [33, 15] (both mostly used in scientific computing), Pachyderm [14] (bioinformatics), Argo Workflows (big data) and Kubeflow (ML). We evaluate sAirflow on workflows derived from Alibaba Cloud [38], following [21], advocating real-world-based traces.

Running workflows in the cloud: A survey [22] classifies the available approaches, challenges, and evaluation techniques for scientific workflows in the cloud. In particular, [28, 7, 18] focus on systems approaches to achieve a reliable and scalable scientific WMS. Improvements often concentrate on the scheduling algorithm [32, 3] with, e.g., reinforcement learning [17], or prediction of task execution times[19]. sAirflow uses a complementary approach: by switching to a different execution model (serverless), we reduce the platform and worker costs.

Serverless computing: In contrast to many papers optimizing serverless backends[20, 5, 36], we take the perspective of a developer of serverless applications. Our contribution is analogous to adaptations of existing systems to FaaS: Unix shell [27], MapReduce [11], or parallelizing Python [2]. A survey [24] of 89 serverless applications does not analyze whether an application was migrated to serverless.

A serverless blueprint [31] does not address the challenges of starting from a legacy system. CDC was proposed as one of possible methods to *interoperate* with legacy systems in the software architecture context [25], but they do not specifically target *migrating* large, legacy code, nor they quantify performance.

Serverless Workflow Management Frameworks (WMF): Some serverless WMFs are built from scratch [1, 16], thus introducing migration risks and incompatibilities. [16] introduces a container-based WMF; similarly to sAirflow, they rely on messages for internal communication. analyzes serverless-based WMF. [30] extends FaaS with stateful, addressable instances to run workflows (compatible with Durable Functions, not Airflow). [23] uses serverless containers in HyperFlow by extending the executors to run on AWS Lambda and AWS Fargate; HyperFlow orchestrates a one-off workflow, in contrast to Airflow's continously-running control plane coordinating recurring runs of multiple workflows. [6] states that FaaS-based workflow orchestrators (AWS Step Functions or Apache OpenWhisk Composer) are more cost-efficient and easier to scale than Airflow. Our sAirflow addresses this exact shortcoming.

Apache Airflow extensions and serverless: Airflow scheduler improvements include [34, 12]; they propose adding components to predict resource requirements. [46] provides an Airflow executor (a plugin) allowing job scheduling on AWS Batch and managed Kubernetes. While this plug-in partly addresses the scaling of executors, the control plane is always running — in contrast to sAirflow that additionally uses serverless architecture for the control plane.

3 System context: FaaS and Airflow

FaaS and related serverless offerings: A *function* is the core building block in a FaaS platform. A programmer defines a function by writing code, packaging it, specifying its memory limits, and defining the invoking triggers. As serverless applications are event-driven, the programmer must bind the invocation of a function to an event: e.g., a direct HTTP call; or a periodic, cron-like schedule. For resiliency, event producers should be decoupled from consumers. A queuing broker (e.g. Kafka) temporary queues events, thus allowing multiple producers and consumers, or consumers to go briefly offline.

FaaS has, however, certain limitations. First, the maximum time of a single function run is limited (e.g., 15 min in AWS). Second, a *cold start* occurs when new resources are assigned for an invocation; a cold start increases the response time by between 300 ms and 24 s [4] — a significant delay for short functions. Third, the functions are expected to be stateless, as the environment is ephemeral. Therefore, state must be stored externally (e.g., in blob storage) which may significantly slow down some applications.

Apache Airflow: a workflow scheduler: Airflow processes *DAG*s (workflows) of *tasks*. Tasks are the smallest units of work, with user-defined processing, e.g., copying files or creating a database table from a query. Dependencies between tasks are defined through an API or special operators in the task code. A workflow can be launched manually or scheduled to run, e.g., every day at 4 am. A single execution of a workflow is internally represented by a *DAG run*. Similarly, a single execution of a task is a *task instance*.

The scheduler monitors and orchestrates all tasks and DAGs. By default, once per minute, the scheduler collects DAGs, parses tasks' statuses, and adjusts the metadata on the DAGs and the tasks.

Tasks are launched through local or remote *executors*. A *local executor* launches a task in a child OS process. A *remote executor* sends a task to an external service responsible for queueing and launching. For example, the Kubernetes executor contacts a pre-configured Kubernetes cluster and uses this cluster's API to request a pod that will execute the task.

4 sAirflow: Design and Implementation

We had the following design requirements for sAirflow. (1) *Compatibility* with all Airflow interfaces. (2) *Scalability, Performance and Availability* at least on par with the managed Airflow. (3) *Reproducibility* by persisting infrastructure in code. (4) *Precise intervention* by limiting the modifications of the Airflow source code. (5) *Pay-as-you-use:* Minimize fixed costs. sAirflow achieves all but the last goal, as the database (with the accompanying CDC mechanism), an external dependency, are not pay-as-you-use. Currently, even the AWS serverless database, Aurora, does not scale down to 0; and additionally, it does not support CDC, thus requiring extensive changes in Airflow code (Section 4.2), contradicting our penultimate goal. However, we emphasize these are just external dependencies

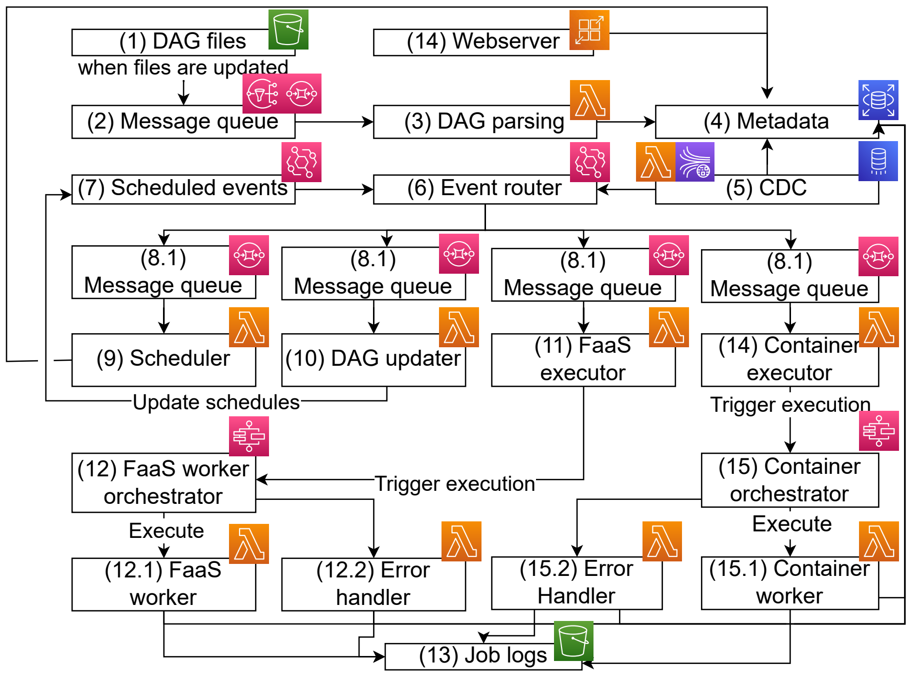


Fig. 1: sAirflow on AWS (icons' source: AWS)

that, in the future, could be replaced with pay-as-you-use products, if the cloud provider introduces them (as, e.g., Google's recent Datastream for CDC).

Fig. 1 presents the high-level design of sAirflow. The figure maps sAirflow architecture to the standard serverless idioms (e.g., a message queue); and then to concrete AWS products (e.g., SQS). As other providers are comparable, porting sAirflow to, e.g., GCP, would require minor code changes.

4.1 Control flow

We introduce the event-based architecture of sAirflow by describing the flow of control in the system, with component numbers referring to Figure 1. An Airflow user submits a new (or updated) workflow which is persisted as a DAG file in the blob storage (1). The storage sends notifications via a message queue (2). The notification triggers a function (3) that parses the DAG files and updates the Metadata DB (4). To reduce the load when multiple DAGs are uploaded, we batch these invocations. A similar flow is triggered from the web UI (14).

Changes in the metadata DB are captured by the Change Data Capture (CDC) (5) that produces an event routed to the Event Router (6). Applying CDC to Airflow was the key architectural difficulty we had to solve (Section 4.2)

The Event Router (6) routes the following events:

A *change of a parsed DAG* is routed to a function (10) that parses the schedule and updates the scheduled events in EventBridge, a cron-like module (7).
A *periodic event* represents a single launch of a workflow that, e.g., is scheduled daily at 4am. This event is routed to the Airflow scheduler (9). The scheduler creates a new *DAG run* in the metadata DB (4).
A *DAG run* event is routed to the scheduler (9), Section 4.3. The scheduler determines if new tasks should be created and which tasks should be executed. Upon deciding to run a task, the scheduler changes the task's status in the metadata DB (4) to queued, triggering a *task queued* event.
A *task queued* event is routed to one of two executors (workers): the Function Executor (11) for invocations of up to 15 min.; or the Container Executor (14). Executors start the user-defined task (12.1) and handle failures (12.2), Section 4.4. We stress that executors do not actively wait for the completion of the user work. Once the task is completed (or fails), the executor saves logs (13) and updates the metadata DB (4), triggering a *task finished/failed* event.
A *task finished/failed* event is routed to the scheduler (9), that reruns a task, queues its successors, or marks the DAG run as complete.

4.2 Change Data Capture (CDC)

In serverless architecture, propagating database changes in an event-driven manner requires additional effort. A Change Data Capture (CDC) pattern allows us to reuse most Airflow code (rather than reimplement all database interactions). In AWS cloud, CDC is provided through Database Migration Service (DMS), an external dependency. DMS creates an event on a change in the SQL database and forwards it to a pre-configured destination. We use Amazon Kinesis Data Streams, coupled with a short function (executed as AWS Lambda) to pre-parse the event (e.g., remove redundancies).

DMS introduces a significant delay to the control loop. Typically, it takes 1-1.5 s between the change in the database and the event being delivered to the event router. Our experiments will later show this delay considerably affects sAirflow performance.

The alternative to CDC is to manually inject code that generates events near each DB modification. Apart from the volume of the code modifications needed, the problem with that approach is the joint, transactional nature of the event and the database change. A naive implementation has a *dual write problem*: if the process fails after the database change but before the event is sent, the event might be lost; and if we reverse the order, the event might be consumed by a reader before the change in the database is committed (and visible to the reader).

4.3 Scheduler

Airflow scheduler determines which tasks to launch. Airflow runs the scheduler as a separate, always-running thread, even if workflows are executed only sporadically. Moreover, all previous attempts to serverless Airflow kept this always-on scheduler (Section 2). The change of the Airflow scheduler's architecture to event-based — without major refactoring, and retaining Airflow scheduling semantics — was one of the key difficulties while working on sAirflow.

In sAirflow, an event triggers the execution of a scheduling algorithm — for example, an event produced when a task has just been completed. A single pass of the scheduler is executed in a single FaaS invocation. To increase reliability, Airflow can be configured to run multiple, redundant schedulers. In contrast, sAirflow's reliability directly relies on the guarantees provided by FaaS (e.g., multiple availability zones for AWS Lambda). In Airflow, most of the scheduler code executes in a critical section. For consistency, sAirflow feeds the scheduler from a single-shard message queue. The algorithm, however, is largely not modified:

1. For each DAG ready to execute: create a DAG run.
2. For each task in each DAG run with all predecessors completed: create a scheduled task instance.
3. For each scheduled task instance, label it queued.

In contrast to Airflow which might launch some (short) tasks and then actively poll their state running the scheduling loop, sAirflow consistently relies on changes in the metadata database, delivered to external executors.

4.4 Executors and workers

An executor starts a task instance and then monitors its execution. sAirflow moves the task handling logic to AWS Step Functions; this enables sAirflow to avoid always-on workers polling the state of the user task. AWS Step Functions executes the user code (as a lambda or in a container, details follow). If the user code fails, AWS Step Functions calls a short sAirflow lambda that handles this failure.

sAirflow provides two executors (Function and Container), but the framework algorithm is common:

1. **Invoke execution**: AWS services invoke sAirflow code in an isolated environment. The environment contains a handler that intakes the metadata about the task to execute. The metadata is then passed to the worker.
2. **Pull configuration**: The worker downloads the deployment configuration from the blob storage.
3. **Pull DAG files**: The worker downloads the DAG files defining the workload.
4. **Start task**: The worker starts the task using LocalTaskJob, a standard Airflow component that executes the task in the process of the worker. When a task completes (or fails), this component modifies the metadata DB.
5. **Push logs**: When a task completes, logs are collected throughout the runtime and sent to the blob storage. sAirflow needs minor modifications to push the logs and not close the logging sinks (thus, a single Lambda instance can serve for multiple invocations).

The function executor and the container executor differ by what service they use to run the worker. The *Function Executor* uses FaaS, AWS Lambda. While FaaS scales extremely well, the execution duration is limited (in AWS, to 15 min).

This executor forwards task instances from an AWS SQS to a serverless orchestrator, AWS Step Functions.

In the *Container Executor*, the worker launches code in a container through an external service, AWS Batch with AWS Fargate. Containers typically have unbounded execution duration. As in a standard managed container service, a container must specify the limits on the memory, CPU and number of copies. AWS Batch on AWS Fargate supports horizontal scaling (including to 0), thus is consistent with our pay-per-use requirement. AWS Fargate does not limit the duration of execution but typically scales out slower than AWS Lambda [23] and with a significant start-up overhead. On each invocation, the start-up might involve downloading an image and initializing a container, which results in a minutes-long delay. While [23] reports a 1-minute start time, in our experiments, we additionally observed significant variance.

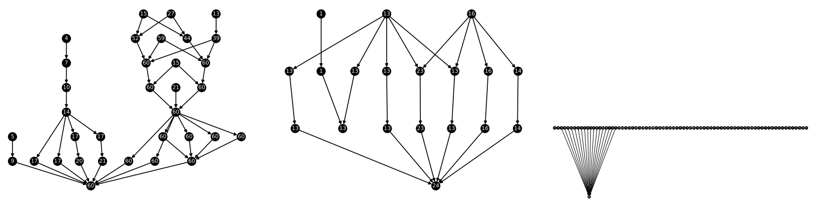
5 Deployment and Evaluation Method

To benchmark the performance, we deploy sAirflow in the cloud and compare it to a SaaS solution, Amazon Managed Workflows for Apache Airflow (MWAA). Cloud has myriad configuration and deployment options; our goal is to create environments that are as similar to each other as possible to achieve fair comparison at a reasonable cost. In this section, we describe how we deploy and configure both systems and then how we generate the workflows.

Managed Workflows for Apache Airflow: We run MWAA with a *small* environment (as the large environment is four times more expensive). By default, MWAA uses the Celery executor, with 5 tasks on each worker node. Unless explicitly specified, this parameter was not changed. The environment starts with one worker and might be scaled to 25 workers. Thus, MWAA can run up to 125 tasks concurrently (a *large* environment might support more tasks, but the scalability issues would simply be deferred). Each worker has 1vCPU and 2GB of RAM, so each task gets roughly 0.2vCPU and 400MB of RAM. MWAA runs two schedulers in parallel (high availability setting), which might be an advantage compared with sAirflow's single scheduler: both schedulers run the scheduling loop processing the workload, which could improve the task throughput.

sAirflow: For a fair comparison, we match sAirflow's configuration to MWAA. Both systems use Airflow 2.4.3. The resources used by sAirflow's services are scaled to ones used by MWAA. The worker functions have a memory limit of 340MB (which corresponds to vCPU of around 0.2 as AWS allocates 1vCPU per 1769MB of memory. The scheduler uses 512MB of RAM (around 0.35vCPU). We use *db.t3.small* (2vCPU, 2GB memory) instance with PostgreSQL (without high availability and *SQL proxy*).

Workloads: We experiment both on synthetic and on realistic workflows, characterized by: n, the number of tasks in the DAG; p_i, the duration of execution of a task in seconds; and T, the period: the DAG executes every T minutes. When measuring *warm* starts, we use $T = 5$ which allows AWS Lambda to reuse previously allocated resources. In contrast, when measuring *cold* starts,



(a) job 3441830 (b) job 581851 (c) job 3302772

Fig. 2: Sample DAGs obtained from jobs in the Alibaba trace. Note that the last DAG is highly parallel (77 tasks in total, 76 of which run in parallel on start-up). Some of the tasks do not have a downstream dependency.

we use $T = 30$: AWS Lambda should always spawn new resources for each invocation. When $T = 5$, we run the given DAG for an hour (12 invocations); when $T = 10$, we also run the DAG for an hour (but with 6 invocations); finally, when $T = 30$, we run the DAG for 1.5 hours (and get 3 invocations). We do not increase the runtime of experiments (to get more invocations), as MWAA would get too expensive.

As common in evaluating schedulers [35], tasks in both realistic and synthetic DAGs `sleep()` for time p: this does not influence the results, as both MWAA and sAirflow execute tasks in isolated environments with static CPU and memory limits. Additionally, workload traces do not contain all information needed to run the tasks (e.g., binaries, environments or parameters).

We use synthetic *chain* and parallel DAGs; and realistic DAGs generated from Alibaba cloud traces [38]. A *chain DAG* has tasks sequentially executing one after another (no parallelism). The optimal execution time of a chain DAG is $n * p$. A *parallel DAG* models highly-parallelizable workloads: after a short startup task, n tasks can be executed in parallel. The optimal execution time is p. Finally, for *Alibaba* DAGs, we extract the DAG shapes and task durations from the batch jobs of Alibaba cloud traces [38]. After filtering out chain and parallel DAGs, we select 30 different DAGs at random. To reduce the costs of experiments, we limit the runtime of any task to at most 60 s.

Fig. 2 shows three examples of derived DAGs. For example, the DAG in (2a) has $n = 34$ tasks: 13 tasks were shortened to 60 s. The *critical path*, i.e., the path with the longest duration, is 439 s, while the *longest path*, i.e., the path with the maximum number of nodes, is 8 nodes.

Metrics: The key metric, measuring the overall efficiency of the system, is the *DAG makespan*, the difference between DAG's start and end times reported by Airflow. More formally, denoting by v_i the task's ready time, s_i the start time, and by c_i the completion time, the makespan is $C_{\max}(D) = \max_{i \in D} c_i - \min_{i \in D} v_i$. Additionally, to better understand performance, we also report the *task duration*, $(c_i - s_i)$: the difference between the duration and the workload p_i shows the per-task overhead of the system; and the *task wait time*, $(s_i - v_i)$, showing the start-up overhead.

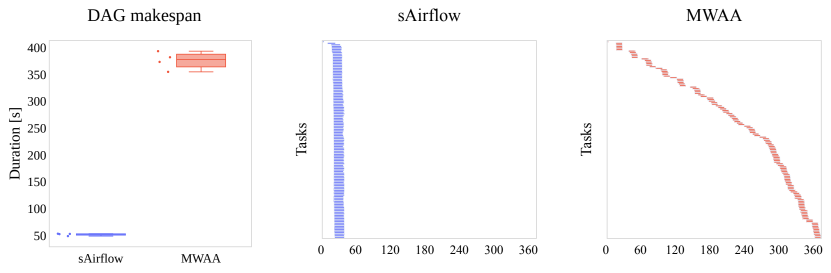


Fig. 3: Parallel DAGs, $p = 10$, $T = 30$, $n = 125$ (cold starts). sAirflow shortens the makespan by 7.2x (left). Gantt charts (middle, right) show a single run.

6 Experimental Evaluation Results

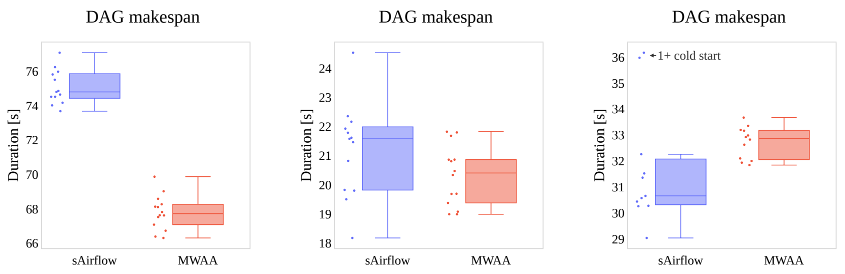
For a comprehensive performance comparison, we consider three setups. We start with the *function (FaaS) executor* in two variants, cold (Section 6.1) and warm (Section 6.2). Then, Section 6.3, we run sAirflow with the *container executor*: as it cannot reuse environments, all the executions are cold. We conclude with cost estimation (Section 6.4). Due to limited space, we show only the key trends and results; for transparency, all metrics and all results are in the Appendix.

6.1 Function executor and cold starts

To measure how the systems handle sporadic load, we run Parallel DAGs with $p = 10$ and $n \in \{16, 32, 64, 125\}$. Due to space constraints, Fig. 3 shows the largest $n = 125$ (full results in the Appendix). We run parallel DAGs to focus on workload with enough work. Thus, any inefficiencies will be directly caused by scaling problems. MWAA is configured to start with one worker and horizontally scale out to up to 25 (supporting 125 concurrent invocations). sAirflow is similarly limited to 125 concurrent FaaS invocations. The 30 minute interval between runs ($T = 30$) ensures that both systems de-provision resources between consecutive runs.

sAirflow is much faster in scaling out to match the demand, exposing Lambda's horizontal scaling with minimal overheads. The managed version of Airflow needs up to *5 minutes* to add a new worker node (Fig. 3, right), whereas sAirflow starts all workers almost immediately, thus completing the whole workload in less than a minute (Fig. 3, first column). Due to MWAA's long horizontal scaling time, sAirflow reduces the makespan by, on average: 1.9 times ($n = 16$), 3.7 times ($n = 32$), 6.13 times ($n = 64$) and 7.2 times ($n = 125$).

In sAirflow, the recorded task durations ($c_i - s_i$) increase when more tasks try to start at the same time: a 10-second-long task takes 12 s to complete when $n = 64$ and 17s when $n = 128$. In these settings, the transactional nature of the internal Airflow's code becomes a bottleneck.



(a) chain, $n = 5$ (b) parallel, $n = 16$ (c) parallel, $n = 125$

Fig. 4: Warm system, $p = 10$, $T = 5$. The first DAG run is not reported.

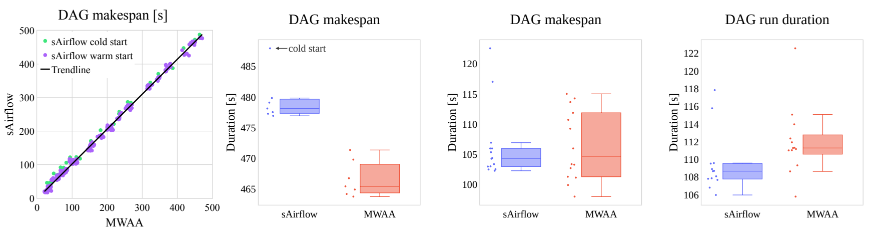
6.2 Function executor and warm starts

To measure the performance under constant load, we now focus on warm invocations. To ensure MWAA is run warm, we disable horizontal scaling by equating the minimum and the maximum number of workers (equal to 25, thus 125 parallel tasks). In sAirflow, we pre-warm the system with a single invocation of the DAG: this invocation warms the lambdas executing the control plane code, as well as the workers. To isolate this effect, we used a one-task DAG and we measured the cold start increasing the waiting time to almost 12 s (vs a median of 2.5 s on a warm system). To focus on warm performance, we now exclude the first DAG invocation from the results unless explicitly stated. However, there is no guarantee that the remaining invocations are all warm — occasional cold starts do happen, and we will take them into account when describing results.

Chain DAGs emphasize the per-task overheads (Fig. 4a): sAirflow is on the average 0.8s slower than MWAA when launching a task, a consequence of the lack of real-time CDC data streaming on AWS. As multiple events have to be sent in sequence to execute the next task: the previous task's completion triggers the scheduler, which marks the next task as queued, which in turn triggers the push of another event. This sequence results in a higher latency due AWS DMS overheads: it might take up to 1 second to push the event out of the database through Kinesis to EventBridge.

Parallel DAGs: When fewer tasks run in parallel ($n = 16$, Fig. 4b, and $n = 32$), MWAA's and sAirflow's DAG execution times are comparable. MWAA is marginally faster for $n = 16$ (by 1.2 s); and similar to sAirflow for $n = 32$. The task wait time for sAirflow is shorter and less variable due to sAirflow's event-driven architecture, in contrast with MWAA's polling executor. sAirflow is faster than MWAA on larger DAGs with more parallelism, $n = 64$ and $n = 125$, Fig. 4c. Each of the two outliers in Fig. 4c can be traced to a cold start of a FaaS worker.

Alibaba traces: comparable performance: DAGs derived from the Alibaba traces show that performance on realistic, industrial workloads confirms the observed trends, with sAirflow outperforming MWAA when parallelism is sufficient. In this analysis, we include the first cold-start execution for sAirflow. Overall, makespans are similar (as emphasized by the trend line in the scatterplot, Fig. 5a). A detailed analysis of three example DAGs confirms the earlier



(a) 30 DAGs (b) job 3441830 (c) job 581851 (d) job 3302772

Fig. 5: Alibaba DAGs: makespans on all DAGs (left) and a detailed analysis on the three DAGs from Fig. 2

trends. On chain-like DAGs, such as the one in Fig. 2a, sAirflow's makespan is minimally worse than MWAA (Fig. 5b): 478 s for sAirflow vs. 465 s for MWAA. The task wait time is the same in both systems, while the per-task overheads are higher in sAirflow; thus, the overall makespan is longer (478 s for sAirflow vs. 465 s for MWAA). There are 8 nodes on the critical path for the DAG; thus, the 13 s increase can be attributed to the per-task overheads. The DAG from Fig. 2b shows a workload where both systems perform similarly (Fig. 5c). Where sAirflow loses in terms of the task duration overhead, it gains on better performance concerning the task wait time. Finally, the DAG in Fig. 2c is similar to our parallel DAG, with over 70 tasks that can be executed in parallel; as expected, sAirflow completes the DAG slightly faster than MWAA.

6.3 Container executor

We measure sAirflow using container workers (AWS Batch with AWS Fargate). Due to space constraints, we only report the key metrics; full results are available in the Appendix. Launching even a single task DAG on a container worker increases the waiting time to 100.5s (from 2.5s with FaaS worker). Yet, beyond that delay, sAirflow with container workers still efficiently scales horizontally: a parallel DAG with $n = 32$ tasks of $p = 10$ seconds completes in approx. 140s (compared with approx. 160s needed by cold-starting MWAA).

6.4 Monetary Cost Estimates

While the cloud pricing changes, we believe that the qualitative price difference between on-demand VMs and transient serverless should remain relatively stable, rendering the cost comparison based on current prices at least qualitatively correct in the longer term. We assume systems run continuously over 24 hours; we analyze costs with four types of workload (heavy, sporadic parallelizable, sporadic light, and constant light, see Appendix for the exact definitions and results). We lower-bound MWAA costs by assuming that MWAA'a autoscaling bugs are resolved. We upper-bound sAirflow costs by excluding the free tier, and assuming the database and the CDC are always available (while with a sporadic load, CDC might be switched off).

Overall, sAirflow halves the fixed cost. The total cost of sAirflow is lower by 17–48%. As serverless products eliminate paying for idle resources, sAirflow is cheaper on sporadic and unpredictable workloads. sAirflow, due to the platform's elasticity, also eliminates the need to account for the worst-case load when deploying the service (in contrast to MWAA, that cannot reliably downscale). In both systems, the costs are driven principally by the size of the database.

7 Conclusions

We show how to adapt an existing, complex application, Apache Airflow, to the serverless cloud. Our prototype uses FaaS for Airflow's control plane, message queues populated by change data capture (CDC) for internal messaging, and FaaS and CaaS for workers. Through micro-benchmarks and 30 real-world DAGs derived from the Alibaba Cloud traces, we compare the performance of our system with a commercially-maintained version on AWS (MWAA). Our results show that sAirflow with FaaS workers scales notably better than MWAA. When a workflow has enough parallelism, a cold system scales in seconds to 125 workers, reducing completion times by 2x-7x. Conversely, sequential workflows, particularly with long tasks requiring containers, highlight increased latencies stemming from propagating CDC events (approx. 2 s); and launching containers through a queuing system (approx. 90 s).

Adopting a comprehensive, legacy system like Apache Airflow to serverless illustrates the difficulties and effort involved. Our extensive experiments show the performance penalty directly resulting from gaps in the current serverless offerings: the SQL database and the CDC process. Ideally, these two capabilities should be integrated into a single cloud-native serverless service.

Acknowledgements

This research was supported by Polish National Science Center grant Opus (UMO-2017/25/B/ST6/00116), and by the Department of Energy under grant DE-SC0024387.

References

1. Jiang, Q., Lee, Y.C., Zomaya, A.Y.: Serverless execution of scientific workflows. In: International Conference on Service-Oriented Computing, pp. 706–721 (2017)
2. Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., Recht, B.: Occupy the cloud: Distributed computing for the 99%. In: SoCC, Proc. Pp. 445–451 (2017)
3. Kijak, J., Martyna, P., Pawlik, M., Balis, B., Malawski, M.: Challenges for scheduling scientific workflows on cloud functions. In: CLOUD, pp. 460–467 (2018)
4. Manner, J., Endress, M., Heckel, T., Wirtz, G.: Cold start influencing factors in function as a service. In: UCC Proc. Pp. 181–188. IEEE (2018)
5. Aumala, G., Boza, E., Ortiz-Aviles, L., Totoy, G., Abad, C.: Beyond load balancing: Package-aware scheduling for serverless platforms. In: CCGRID, Proc. IEEE (2019)

6. Barcelona-Pons, D., Garcia-Lopez, P., Alvaro, R., Gomez-Gomez, A., Paris, G., Sanchez-Artigas, M.: FaaS Orchestration of Parallel Workloads. In: WOSC (2019)
7. Cai, Z., Li, X., Ruiz, R.: Resource Provisioning for Task-Batch Based Workflows with Deadlines in Public Clouds. IEEE TCC **7**(3), 814–826 (2019)
8. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The Rise of Serverless Computing. Commun. ACM **62**(12), 44–54 (2019)
9. Deelman, E., Vahi, K., Rynge, M., Mayani, R., da Silva, R.F., Papadimitriou, G., Livny, M.: The evolution of the pegasus workflow management software. Computing in Science & Engineering **21**(4), 22–36 (2019)
10. Fairbanks, G.: Ignore, Refactor, or Rewrite. IEEE Software **36**(2), 133–136 (2019)
11. Giménez-Alventosa, V., Moltó, G., Caballer, M.: A framework and a performance assessment for serverless MapReduce on AWS Lambda. FGCS **97**, 259–274 (2019)
12. Ilyushkin, A., Bauer, A., Papadopoulos, A.V., Deelman, E., Iosup, A.: Performance-Feedback Autoscaling with Budget Constraints for Cloud-based Workloads of Workflows, (2019). arXiv: `1905.10270 [cs.DC]`.
13. Mitchell, R., Pottier, L., Jacobs, S., Silva, R.F.d., Rynge, M., Vahi, K., Deelman, E.: Exploration of Workflow Management Systems Emerging Features from Users Perspectives. In: Big Data, pp. 4537–4544 (2019)
14. Novella, J.A., Emami Khoonsari, P., Herman, S., Whitenack, D., Capuccini, M., Burman, J., Kultima, K., Spjuth, O.: Container-based bioinformatics with Pachyderm. Bioinformatics **35**(5), 839–846 (2019)
15. Chard, R., Babuji, Y., Li, Z., Skluzacek, T., Woodard, A., Blaiszik, B., Foster, I., Chard, K.: Funcx: A federated function serving fabric for science. In: Proceedings of the 29th International symposium on high-performance parallel and distributed computing, pp. 65–76 (2020)
16. Dessalk, Y.D., Nikolov, N., Matskin, M., Soylu, A., Roman, D.: Scalable Execution of Big Data Workflows Using Software Containers. In: MEDES. ACM (2020)
17. Farid, M., Latip, R., Hussin, M., Abdul Hamid, N.A.W.: Scheduling Scientific Workflow Using Multi-Objective Algorithm With Fuzzy Resource Utilization in Multi-Cloud Environment. IEEE Access **8**, 24309–24322 (2020). `https://doi.org/10.1109/ACCESS.2020.2970475`
18. Lopez, P.G., Arjona, A., Sampe, J., Slominski, A., Villard, L.: Triggerflow: Trigger-Based Orchestration of Serverless Workflows. In: DEBS. ACM (2020)
19. Pham, T.-P., Durillo, J.J., Fahringer, T.: Predicting Workflow Task Execution Time in the Cloud Using A Two-Stage Machine Learning Approach. IEEE Transactions on Cloud Computing **8**(1), 256–268 (2020). `https://doi.org/10.1109/TCC.2017.2732344`
20. Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Bianchini, R.: Characterization and Optimization of the Serverless Workload at a Large Cloud Provider. USENIX (2020)
21. Versluis, L., Mathá, R., Talluri, S., Hegeman, T., Prodan, R., Deelman, E., Iosup, A.: The Workflow Trace Archive: Open-Access Data From Public and Private Computing Infrastructures. TPDS **31**(9) (2020)
22. Ahmad, Z., Jehangiri, A.I., Ala'anzy, M.A., Othman, M., Latip, R., Zaman, S.K.U., Umar, A.I.: Scientific Workflows Management and Scheduling in Cloud Computing: Taxonomy, Prospects, and Challenges. IEEE Access **9** (2021)
23. Burkat, K., Pawlik, M., Balis, B., Malawski, M., Vahi, K., Rynge, M., da Silva, R.F., Deelman, E.: Serverless Containers – Rising Viable Approach to Scientific Workflows. In: eScience (2021)

24. Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C.L., Iosup, A.: The state of serverless applications: Collection, characterization, and community consensus. IEEE TSE **48**(10), 4152–4166 (2021)
25. Gilbert, J., Price, E.: Software Architecture Patterns for Serverless Systems: Architecting for innovation with events, autonomous services, and micro frontends. Packt Publishing (2021)
26. Harenslak, B.P., de Ruiter, J.: Data pipelines with apache airflow. Simon and Schuster (2021)
27. Mahéo, A., Sutra, P., Tarrant, T.: The serverless shell. In: Middleware: Industrial Track, pp. 9–15 (2021)
28. Ahmad, Z., Jehangiri, A.I., Mohamed, N., Othman, M., Umar, A.I.: Fault Tolerant and Data Oriented Scientific Workflows Management and Scheduling System in Cloud Computing. IEEE Access **10** (2022)
29. Bedford, T.: Diagnosing Airflow's Auto-Scaling Flaw in AWS MWAA, (2022). `https://technical.thombedford.com/267` (visited on 01/22/2023)
30. Burckhardt, S., Chandramouli, B., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C.S., Zhu, X.: Netherite: Efficient execution of serverless workflows. VLDB **15**(8), 1591–1604 (2022)
31. Copik, M., Calotoiu, A., Taranov, K., Hoefler, T.: FaasKeeper: a Blueprint for Serverless Services, (2022). arXiv: `2202.05711 [cs.DC]`.
32. Kamran, A., Farooq, U., Rabbi, I., Zia, K., Assam, M., Alsolai, H., Al-Wesabi, F.N.: A Unified Mechanism for Cloud Scheduling of Scientific Workflows. IEEE Access **10** (2022)
33. Li, Z., Chard, R., Babuji, Y., Galewsky, B., Skluzacek, T.J., Nagaitsev, K., Woodard, A., Blaiszik, B., Bryan, J., Katz, D.S., *et al.*: FuncX: Federated function as a service for science. IEEE Transactions on Parallel and Distributed Systems **33**(12), 4948–4963 (2022)
34. Lin, E., Xu, L., Bramhavar, S., de Oca, M.M., Gorsky, S., Yi, L., Groetsema, A., Chou, J.: Global Optimization of Data Pipelines in Heterogeneous Cloud Environments, (2022). arXiv: `2202.05711 [cs.DC]`.
35. Przybylski, B., Pawlik, M., Zuk, P., Lagosz, B., Malawski, M., Rzadca, K.: Using Unused: Non-Invasive Dynamic FaaS Infrastructure with HPC-Whisk. In: Supercomputing. IEEE (2022)
36. Zuk, P., Przybylski, B., Rzadca, K.: Call Scheduling to Reduce Response Time of a FaaS System. In: CLUSTER, pp. 172–182. IEEE (2022)
37. Hamza, M., Akbar, M.A., Smolander, K.: The Journey to Serverless Migration: An Empirical Analysis of Intentions, Strategies, and Challenges. In: Product-Focused Software Process Improvement. Springer (2024)
38. Alibaba, Clusterdata: Public trace data sets of production clusters, `https://github.com/alibaba/clusterdata/tree/master/cluster-trace-v2018`
39. Amazon Web Services, Amazon EventBridge Pricing, `https://aws.amazon.com/eventbridge/pricing/` (visited on 04/13/2023)
40. Amazon Web Services, Amazon Managed Workflows for Apache Airflow Pricing, `https://aws.amazon.com/managed-workflows-for-apache-airflow/pricing/` (visited on 01/22/2023)
41. Amazon Web Services, Amazon S3 Pricing, `https://aws.amazon.com/s3/pricing/?p=pm&c=s3&z=4` (visited on 04/13/2023)
42. Amazon Web Services, Amazon SQS Pricing, `https://aws.amazon.com/sqs/pricing/` (visited on 04/23/2023)

43. Amazon Web Services, Amazon SQS Short and Long Polling, `https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-short-and-long-polling.html#sqs-short-long-polling-differences` (visited on 04/23/2023)
44. Amazon Web Services, AWS Batch on AWS Fargate, `https://docs.aws.amazon.com/batch/latest/userguide/fargate.html` (visited on 01/22/2023)
45. Amazon Web Services, AWS Step Functions Pricing, `https://aws.amazon.com/step-functions/pricing/` (visited on 04/13/2023)
46. Elzeiny, A.: Apache Airflow: Native AWS Executors, `https://github.com/aelzeiny/airflow-aws-executors`

A Cold starts and Function Executor

For transparency, we report full results from the following experiments:

– A detailed analysis of the overheads by analyzing a single-task DAG, Fig. 6.
– Parallel DAGs with $n = 16$, $n = 32$, $n = 64$ and $n = 125$, Fig 7.

We refer to the main text for the analysis of these results.

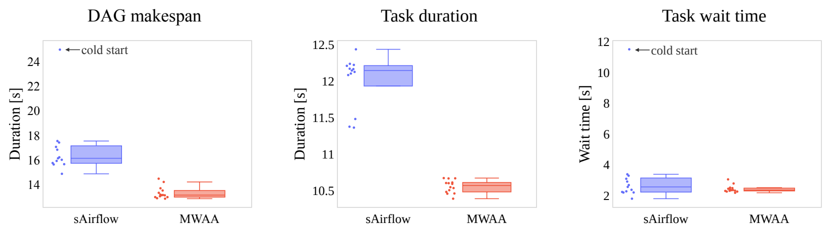
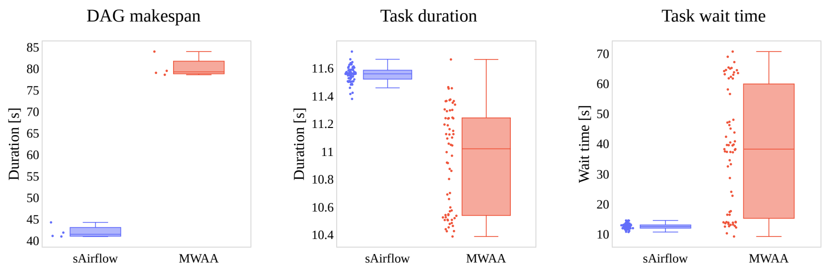
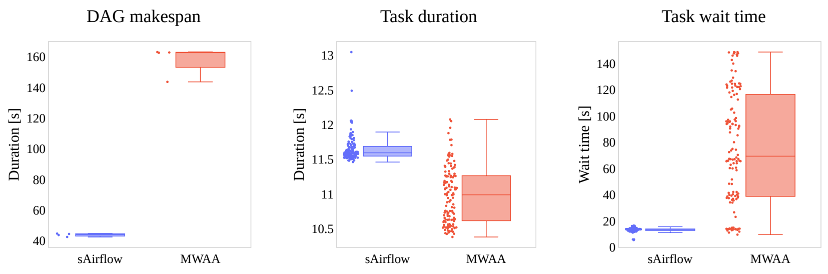


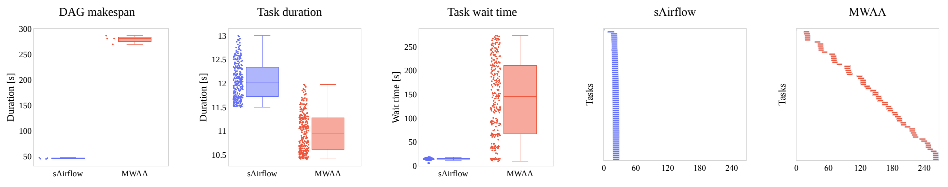
Fig. 6: A single-task DAG (a chain with $n = 1$), $p = 10$, $T = 5$. The first DAG run, resulting in a cold start, corresponds to the outlier in the left and right figures.



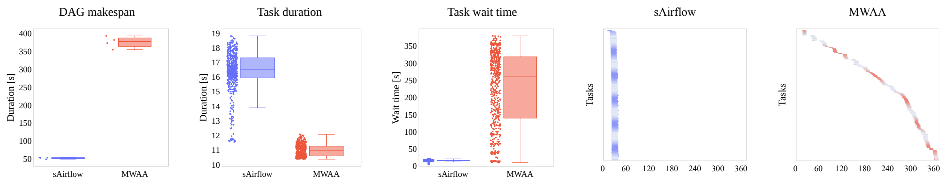
(a) $n = 16$



(b) $n = 32$



(c) $n = 64$



(d) $n = 125$

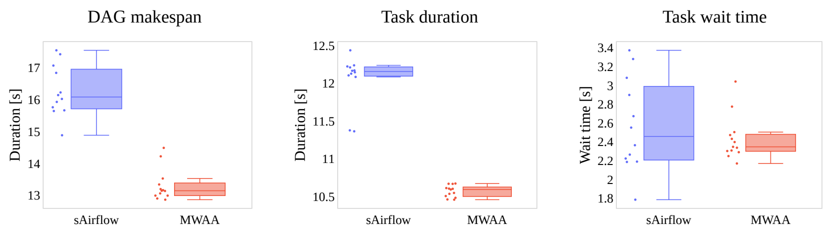
Fig. 7: Parallel DAGs, function executor, cold starts, $p = 10$, $T = 30$. Gantt charts on the right side correspond to one of the DAG runs.

B Warm starts and Function Executor

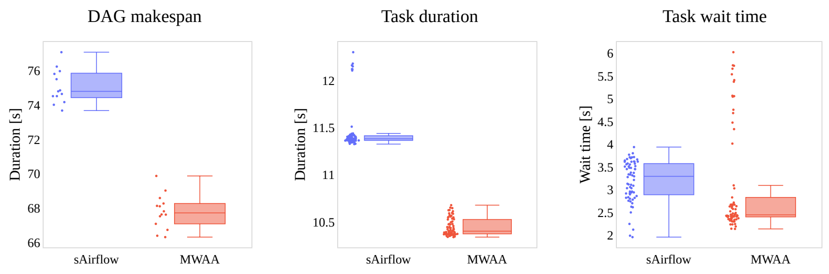
For transparency, we report the full results from the following experiments:

- Chain DAGs with $n = 1$, $n = 5$ and $n = 10$, Fig 8.
- Parallel DAGs with $n = 16$, $n = 32$, $n = 64$ and $n = 125$, Fig 9.

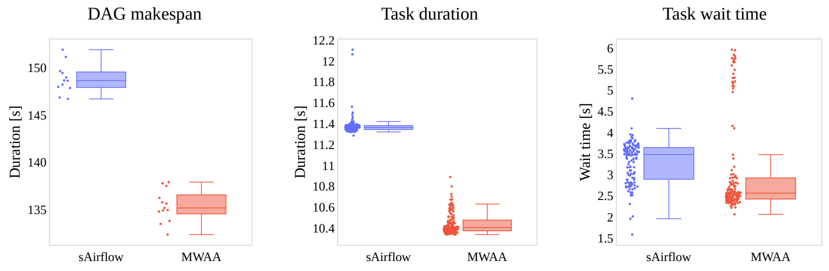
We refer to the main text for the analysis of these results.



(a) $n = 1$

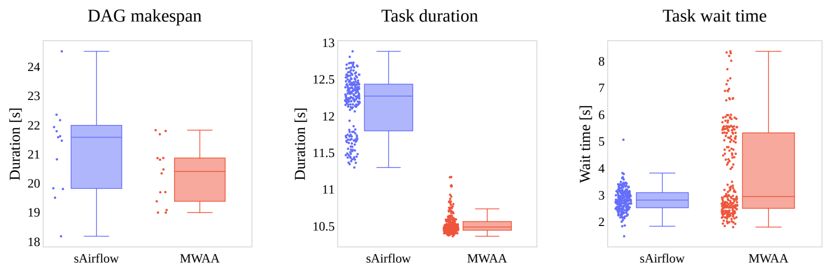


(b) $n = 5$

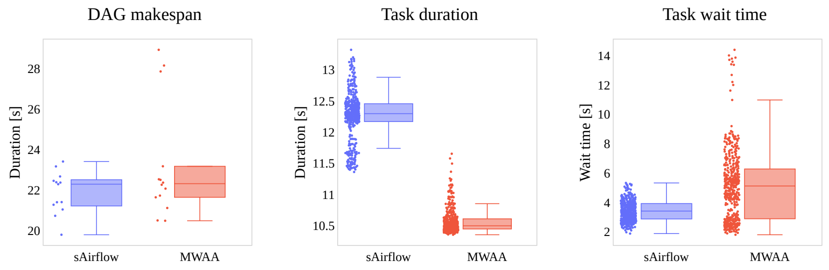


(c) $n = 10$

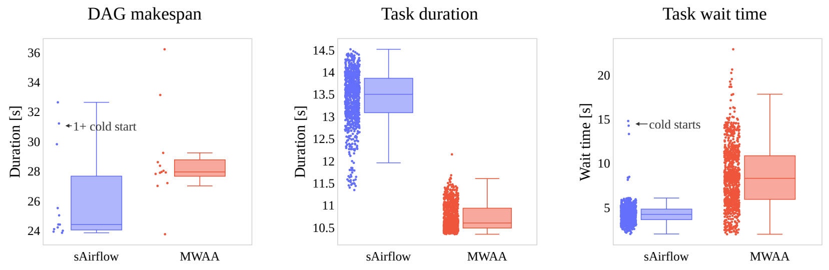
Fig. 8: Chain DAG, function executor, warm starts, $p = 10$, $T = 5$. The first DAG run is not reported.



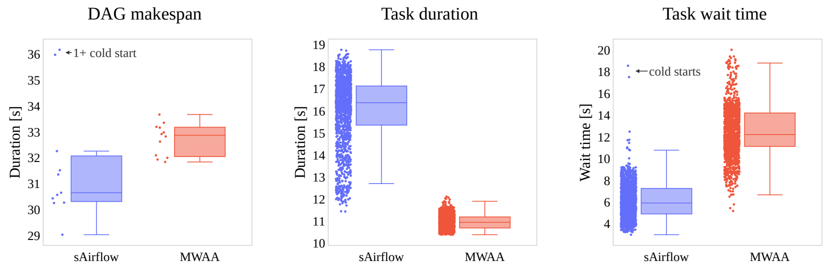
(a) $n = 16$



(b) $n = 32$



(c) $n = 64$



(d) $n = 125$

Fig. 9: Parallel DAGs, function executor, warm starts, $p = 10$, $T = 5$. Gantt charts on the right side correspond to one of the DAG runs.

C Parallel Forest: Both systems perform similarly on multiple DAGs

To show that sAirflow works equally well for multiple DAG, we employ an experiment where multiple copies of the same DAG run in parallel. The DAG has the same parameters no matter how many copies of it are run. We use Parallel Forest DAGs with $n = 8$, $p = 10$, and $k \in \{1, 2, 4, 8\}$. We compare sAirflows overall performance (the first execution of each of the DAGs will warm the system) to MWAA's warm executions.

The resulting metrics (Figure 10) show that the trend across systems is similar. Both systems are almost equally affected by running more copies of the same DAG. As sAirflow is notably better at parallelizing and minimizing the task wait time, the median DAG makespan is not affected as much throughout the experiment. For $k = 1$ sAirflow and MWAA yield 20.90 seconds and 19.60 seconds respectively, for $k = 8$ it is 28.16 seconds and 23.87 seconds respectively.

As the DAG is highly parallel sAirflow faces the same challenges and wins as in the Parallel DAGs experiment. For instance, with $k = 8$ there were in total $8 * 8$ tasks being run across the DAGs. The correlated metrics are close (Figure 11). Thus, sAirflow works equally well when the workload is split into multiple DAGs.

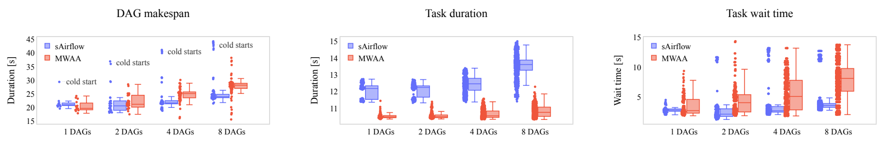


Fig. 10: Parallel forest DAGs, $n = 8$, $p = 10$, $T = 5$, comparison of the system where the same copies of the DAGs are run in parallel (for $k \in \{1, 2, 4, 8\}$).

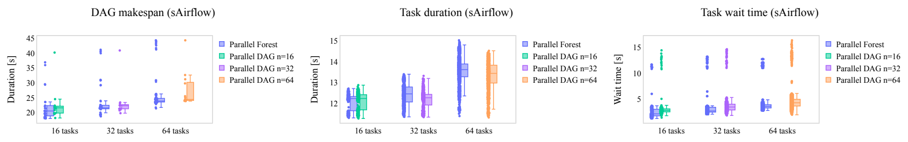


Fig. 11: Comparison between experiments on sAirflow for Parallel DAGs and Parallel Forest grouped by the total number of tasks in the DAGs. For instance, for forest with 8 DAGs there were $8 * 8$ tasks in total, which corresponds to one DAG with $n = 64$ tasks.

D Function Executor, Alibaba DAGs

We use the DAGs generated from the Alibaba Cloud traces to show that performance on realistic, industrial workloads confirms the trends observed in earlier experiments — with sAirflow performing better on more parallel DAGs, while MWAA on more sequential DAGs. In this analysis, we include the first cold-start execution for sAirflow.

In these instances, task durations in a DAG vary. The critical path might also be longer than 5 min., the interval we used earlier. Thus, we set $T = 5$ for DAGs with a critical path less or equal to 200 s; and $T = 10$ for the remaining DAGs. This prevents DAG runs from overlapping, at the risk of getting more cold starts in sAirflow (but not MWAA).

Both systems yield similar performance (Fig. 12) concerning the DAG makespan. The difference between the DAG's critical path and the makespan (the overhead) is also similar (Fig. 13a). sAirflow's overhead is roughly 10% higher (Fig. 13a) in comparison with MWAA, which we attribute to the longer task durations (Fig. 15). The DAG overhead metric averaged over all tasks in a DAG confirms this (Fig. 13b).

To better describe the performance, we normalize the DAG's overhead by a ratio between the DAG maximum parallelism and the longest path (number of nodes) in the DAG. Assume the following notation, where D is the DAG run of graph d:

- $C_{\max}(D)$: the DAG makespan:
- p_d: the critical path duration, equal to $\sum_{\text{i: task i} \in \text{critical path of } d} p_i$
- n_W: the maximum parallelism - the maximum number of tasks that would run in parallel if the DAG is run on a system without any overhead and unlimited resources;
- n_L: the number of nodes on the longest path;

We normalize the performance by:

$$(C_{\max} - p_d) * \left(\frac{n_L}{n_W}\right) \qquad (1)$$

The first component, $(C_{\max} - p_d)$ captures the system overhead on a DAG, whereas the second, $\left(\frac{n_L}{n_W}\right)$ represents the parallelizability of a DAG. MWAA has less overhead on linear DAGs, and sAirflow performs better on more parallelizable DAGs. The metric aims to describe this correlation backed by the distribution of the values (Fig. 14).

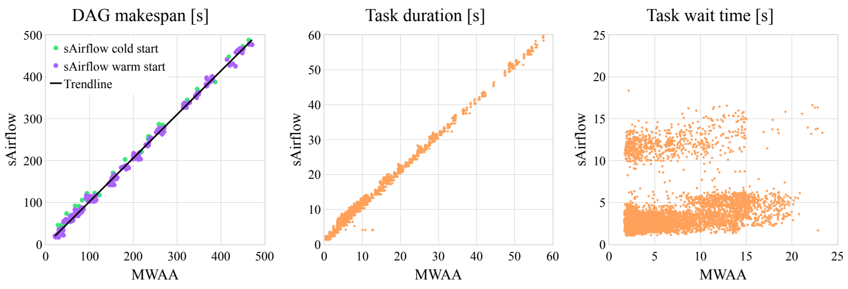
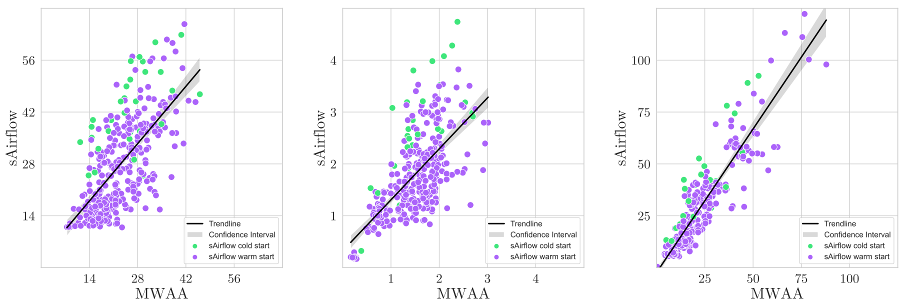
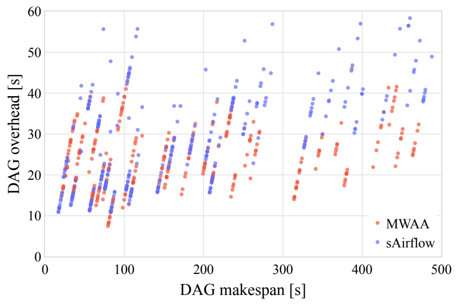


Fig. 12: Metrics comparison for MWAA and sAirflow for DAGs generated from the Alibaba Cloud traces.

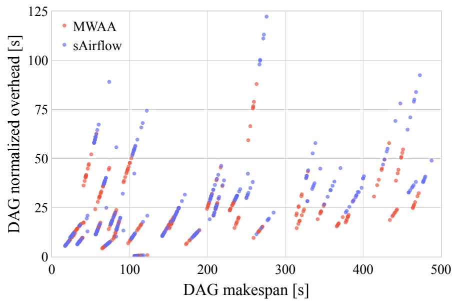


(a) Absolute (b) Per-task (c) Normalized

Fig. 13: Comparison of overhead for DAGs generated from Alibaba Cloud traces



(a) DAG overhead distribution.



(b) y-axis represents Equation 1.

Fig. 14: DAG overhead metric on DAGs generated from Alibaba Cloud traces for each system, with different normalization methods applied.

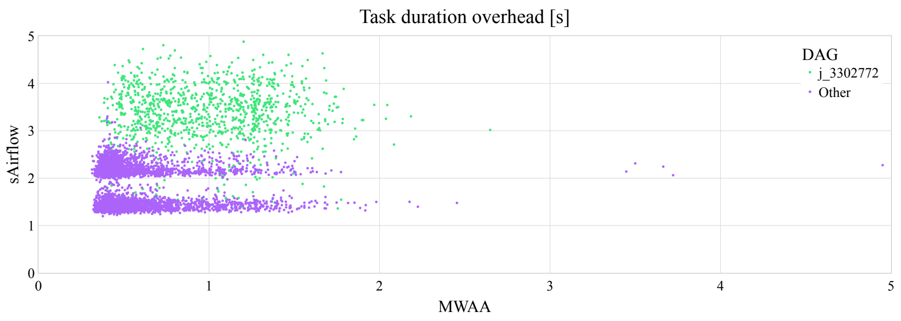


Fig. 15: Comparison of task duration overheads for both systems. The overhead is calculated as the collected task execution time minus the task duration in the DAG. In an ideal system, the overhead for this metric is zero.

E Container executor

We now measure sAirflow using container workers (using AWS Batch with AWS Fargate). The task overhead depends on the number of tasks submitted in parallel and the size of the underlying Docker container image for the worker. [23] comparises costs and performance of FaaS and CaaS on AWS; here, we confirm that similar patterns exist when these executors are exposed through sAirflow.

For all experiments, sAirflow requests 0.5 vCPU and 512MB of memory from AWS Fargate, the lowest configuration available. The typical latency for Batch with Fargate is 60–90 s of provisioning time and then 30 s of the start-up time (these numbers confirm measurements reported in [46]). With the container executor, containers are not reused (no warm starts, in sharp contrast to the FaaS executor). Each time a container starts, its content is pulled from the registry (AWS ECR).

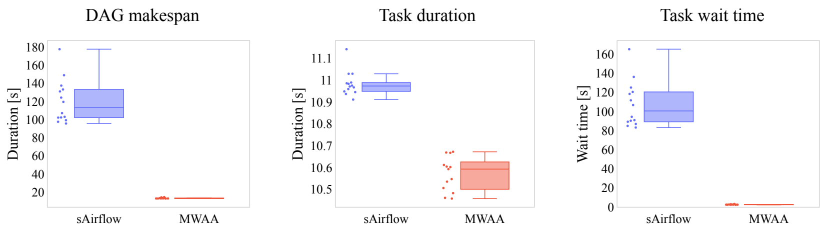


Fig. 16: Chain DAG, $n = 1$, $p = 10$, $T = 5$, sAirflow on CaaS. MWAA keeps at least one worker running, while sAirflow needs to create a new cold container for each task.

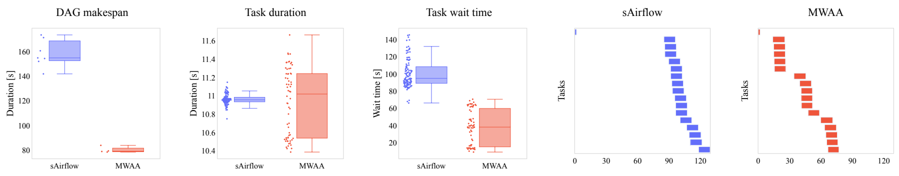
E.1 Chain DAGs: sAirflow has more overhead on a single task DAG for CaaS executor

This experiment shows the overhead that AWS Batch adds over a single task DAG (a chain DAG with $n = 1$, $p = 10$, $T = 5$) on sAirflow with CaaS executor and compare the results with the previous setup, comparing sAirflow with FaaS executor to MWAA (Fig. 8a).

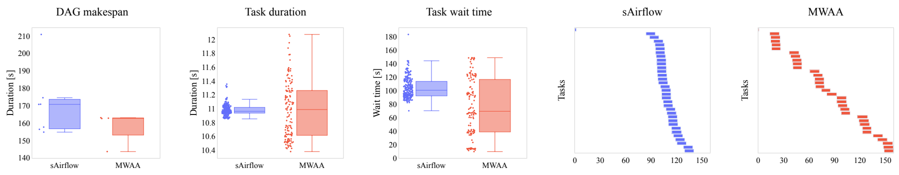
We start with an experiment on a single-task DAG (a chain DAG with $n = 1$, $p = 10$, $T = 5$), Fig. 16). As expected, the task wait time is higher in sAirflow, as MWAA has one active worker by default (which cannot be removed). In sAirflow, replacing AWS Lambda (the function executor) with AWS Batch (the container executor) results in the median wait time increasing from 2.5 s (Fig. 6) to 100.5 s. This follows from the AWS Fargate provisioning time and the container start-up time (e.g., loading the dependencies). As we rely on Apache Airflow, all its dependencies, and other libraries required by sAirflow. This image needs to be propagated on each start up by AWS Fargate. The expected latency on a single task is up to 2.5 minutes but in the end, this number might vary depending on the queuing in AWS Batch [44]. Yet, the task duration time is almost 1 s shorter (Fig. 16, 8a), which follows from the fact that the minimum configuration for AWS Fargate allocated more resources than using the AWS Lambda (the defaults for sAirflow specify 0.2vCPU and 400MB of memory for the function).

E.2 Parallel DAGs: sAirflow with the container executor can match MWAA scaling

To measure how sAirflow with container workers scales horizontally, we run two experiments with the Parallel DAG with $p = 10$, $T = 10$, and $n \in \{16, 32\}$. The first, immediately-completing task of the workload is executed using the function executor, while the rest is executed on CaaS using the container executor. This configuration models a workload with a short coordinating task (running on FaaS), followed by long-running processing (while the experiment uses relatively short $p = 10$ to reduce MWAA costs, the results would naturally extend for



(a) $n = 16$



(b) $n = 32$

Fig. 17: Parallel DAG, $p = 10$; sAirflow on CaaS (except for the immediately-completing DAG root executed on FaaS). MWAA results are from the cold start experiments. The Gantt charts on the right side shows a single DAG runs. MWAA does not manage to scale the cluster in time, so each job executes on the same worker node. sAirflow executes each task on a new container spawned by AWS Batch. sAirflow's start-up overhead (caused by AWS Batch queueing and loading the container image) heavily varies.

long-running tasks). Consequently, only two tasks on the critical path, including one AWS Batch, mean the AWS Batch overhead is included only once.

We compare sAirflow with cold MWAA to measure sporadic, rather than continuous workloads. MWAA takes around 240–270 s to provision an additional worker. The difference between MWAA and sAirflow is that MWAA keeps the worker running in case there is more load (and there are notable issues in this approach due to the poor support for marking the removal of a worker [29]).

Despite being slower for $n = 16$ tasks (Fig. 17a), sAirflow scales similarly when the number of tasks grows and outperforms the MWAA's slow autoscaling (Fig. 17b). AWS Batch (Fig. 17a and 17b) has worse scaling than AWS Lambda (Fig. 7c and 7d). The parallel tasks do not start in the same burstable manner. As a consequence, there is no spike in the task duration metric (Fig. 17b) as the load on the database during the tasks' startup is more evenly distributed in comparison with the previous experiments (Fig. 3).

F Monetary Cost Estimates

In this section, we estimate the monetary gains from serverless-ing Airflow. When designing sAirflow, one of our goals was to reduce fixed costs. Yet, as Airflow relies on an always-available transactional database, sAirflow's cost must take into account that service — and an associated CDC mechanism. However, the resulting fixed cost is significantly lower. The daily cost for running a small environment class for MWAA without any additional workers is $11.76 [40], sAirflow amounts to $6.03 in an equivalent configuration (Table 6). Most of the sAirflow's cost is attributed to running the database and the CDC subsystem. We compare the systems using four workloads with different characteristics:

1. **Heavy Load**: A single DAG with 50 tasks in parallel, scheduled every 3 minutes, runs 20 times; each task takes 3 minutes. Thus, each DAG run takes 3 minutes, and the execution is finished in an hour.
2. **Distributed Load**: A single DAG with 400 tasks, scheduled every 4 hours, runs 6 times. DAG execution time is always ¡ 1 hour. Requires to scale to 35 tasks in parallel. Each task takes 1 minute to finish. In total, 6*400 tasks will run.
3. **Sporadic Light Load**: A chain DAG with 20 tasks, scheduled every 24 hours, runs 1 time. Each task takes 30 seconds to finish.
4. **Constant Load**: A single DAG with 100 tasks in parallel, scheduled every 24 hours, runs only once. Each task takes 24 hours to finish.

In every workload, we run both systems continuously for a 24-hour period. We make the assumption that MWAA's autoscaling operates without any downtime(although this assumption is not accurate [29]). We assume that MWAA'a autoscaling works with zero downtime (which is not the case). In all workloads, we run both systems for 24 hours. We exclude Free Tier from calculations for sAirflow.

In general, sAirflow reduces the fixed cost by half. The final cost of sAirflow is lower by 17-48% (Table 1). It's important to note that we do not include the Free Tier in our calculations for sAirflow.

Table 1: Summary of the monetary cost comparison between MWAA and sAirflow in different scenarios. All cost is given in [$], rounded up to two decimal places. Breakdown for sAirflow is in the following tables.

Scenario	MWAA			sAirflow			
	Fixed	Workers	Total	Fixed	Variable	Executor	Total
(1) Heavy	11.76	0.50	12.26	6.03	1.27	FaaS	7.30
					0.89	CaaS	6.92
(2) Distributed	11.76	1.98	13.74	6.03	1.44	FaaS	7.47
(3) Sporadic	11.76	0	11.76	6.03	0.02	FaaS	6.05
(4) Constant	11.76	31.68	43.44	6.03	29.66	CaaS	35.69

Table 2: Cost breakdown of the major serverless components for running sAirflow in Scenario (1) with FaaS executor.

Component	Notes	Cost [$]
Function Worker (Lambda)	1000 invocations (one per task), 340MB memory, 3min each	0.9963
Function Executor (Lambda)	1000 invocations (one per task in the scheduled state), 256MB memory, 1s each	0.0044
Scheduler (Lambda)	1530 invocations, 512MB memory, 10s each; (input batch size is 10 events, there are $15 * 1000$ events for the tasks data and $20 * 15$ for the DAG schedules)	0.1278
CDC event forwarded (Lambda)	1530 invocations, 512MB memory, 1s each	0.0131
Step functions	1000 invocations, 4 state transitions each; [45]	0.1000
Dag files pull (S3)	1000 GET requests in each task for the DAG file [41]	0.0004
Push task logs (S3)	1000 PUT requests [41]	0.0050
Eventbridge	1000 * 15 events ingested [39]	0.0150
SQS FIFO	4320 calls (86400/20, seconds in the entire day per 20 seconds poll interval [43]); Scheduler queue; [42]	0.0022
SQS	8640 calls (86400/10, seconds in the entire day per 10 seconds poll interval [43]); Scheduler queue; [42]	0.0035
Total		1.2677

Serverless offerings eliminate the necessity of optimizing deployment towards the worst-case scenario. By design, sAirflow is a more cost-effective option for sporadic and unpredictable workloads. The extent of cost savings varies depending on the specific nature of the workloads.

Table 3: Cost breakdown of the major serverless components for running sAirflow in Scenario (2) with FaaS executor.

Component	Notes	Cost [$]
Function Worker (Lambda)	2400 invocations (one per task), 340MB memory, 1min each	0.7974
Function Executor (Lambda)	2400 invocations (one per task in the scheduled state), 256MB memory, 1s each	0.0105
Scheduler (Lambda)	3609 invocations, 512MB memory, 10s each; (input batch size is 10 events, there are 15 * 2400 events for the tasks data and 6 * 15 for the DAG schedules)	0.3015
CDC event forwarded (Lambda)	3609 invocations, 512MB memory, 1s each	0.0308
Step functions	2400 invocations, 4 state transitions each; [45]	0.24
Dag files pull (S3)	2400 GET requests in each task for the DAG file [41]	0.001
Push task logs (S3)	2400 PUT requests [41]	0.012
Eventbridge	2400 * 15 events ingested [39]	0.036
SQS FIFO	4320 calls (86400/20, seconds in the entire day per 20 seconds poll interval [43]); Scheduler queue; [42]	0.0022
SQS	8640 calls (86400/10, seconds in the entire day per 10 seconds poll interval [43]); Scheduler queue; [42]	0.0035
Total		1.4349

Table 4: Cost breakdown of the major serverless components for running sAirflow in Scenario (3) with FaaS executor.

Component	Notes	Cost [$]
Function Worker (Lambda)	20 invocations (one per task), 340MB memory, 1min each	0.0033
Function Executor (Lambda)	20 invocations (one per task in the scheduled state), 256MB memory, 1s each	0.0001
Scheduler (Lambda)	32 invocations, 512MB memory, 10s each; (input batch size is 10 events, there are 1 * 20 events for the tasks data and 20 * 15 for the DAG schedules)	0.0027
CDC event forwarded (Lambda)	32 invocations, 512MB memory, 1s each	0.0003
Step functions	20 invocations, 4 state transitions each; [45]	0.002
Dag files pull (S3)	20 GET requests in each task for the DAG file [41]	0
Push task logs (S3)	20 PUT requests [41]	0.0001
Eventbridge	20 * 15 events ingested [39]	0.0003
SQS FIFO	4320 calls (86400/20, seconds in the entire day per 20 seconds poll interval [43]); Scheduler queue; [42]	0.0022
SQS	8640 calls (86400/10, seconds in the entire day per 10 seconds poll interval [43]); Scheduler queue; [42]	0.0035
Total		0.0145

Table 5: Cost breakdown of the major serverless components for running sAirflow in Scenario (4) with CaaS executor.

Component	Notes	Cost [$]
Container Worker (Batch)	100 invocations (one per task), 0.25vCPU, 500MB memory, 24hours each	29.62
Container Executor (Lambda)	100 invocations (one per task in the scheduled state), 256MB memory, 1s each	0.0004
Scheduler (Lambda)	152 invocations, 512MB memory, 10s each; (input batch size is 10 events, there are 15 * 100 events for the tasks data and 1 * 15 for the DAG schedules)	0.0127
CDC event forwarded (Lambda)	152 invocations, 512MB memory, 1s each	0.0013
Step functions	100 invocations, 4 state transitions each; [45]	0.01
Dag files pull (S3)	100 GET requests in each task for the DAG file [41]	0
Push task logs (S3)	100 PUT requests [41]	0.0005
Eventbridge	100 * 15 events ingested [39]	0.0015
SQS FIFO	4320 calls (86400/20, seconds in the entire day per 20 seconds poll interval [43]); Scheduler queue; [42]	0.0022
SQS	8640 calls (86400/10, seconds in the entire day per 10 seconds poll interval [43]); Scheduler queue; [42]	0.0035
Total		29.6521

Table 6: sAirflow's fixed price components breakdown. All cost is given in [$], rounded up to two decimal places.

Component	Specification	Daily	Daily HA	Monthly	Monthly HA
RDS	instance: db.t3.small, SSD: 20GB	0.94	1.88	28.58	57.16
DMS	instance: t3.small, SSD: 10GB	0.90	1.80	27.43	54.86
Kinesis	data streams	0.72	0.72	21.90	21.90
NAT	instance: t2.micro, on-demand	0.28	0.55	8.36	16.71
ECR	container images, 11*400MB	0.02	0.02	0.50	0.50
SQL proxy		0.72	0.72	21.90	21.90
AppRunner	2GB of memory in a stopped state	0.34	0.34	10.92	10.92
Total		3.92	6.03	119.59	183.95