ExoFlow: A Universal Workflow System for Exactly-Once DAGs

Siyuan Zhuang, *UC Berkeley;* Stephanie Wang, *UC Berkeley and Anyscale;*
Eric Liang and Yi Cheng, *Anyscale;* Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/osdi23/presentation/zhuang

This paper is included in the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-34-2

Open access to the Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

ExoFlow: A Universal Workflow System for Exactly-Once DAGs

Siyuan Zhuang
UC Berkeley

Stephanie Wang
UC Berkeley, Anyscale

Eric Liang
Anyscale

Yi Cheng
Anyscale

Ion Stoica
UC Berkeley

Abstract

Given the fundamental tradeoff between run-time and recovery performance, current distributed systems often build application-specific recovery strategies to minimize overheads. However, it is increasingly common for different applications to be composed into heterogeneous pipelines. Implementing multiple interoperable recovery techniques in the same system is rare and difficult. Thus, today's users must choose between: (1) building on a single system, and face a fixed choice of performance vs. recovery overheads, or (2) the challenging task of stitching together multiple systems that can offer application-specific tradeoffs.

We present ExoFlow, a *universal workflow* system that enables a flexible choice of recovery vs. performance tradeoffs, even within the same application. The key insight behind our solution is to *decouple execution from recovery* and provide exactly-once semantics as a separate layer from execution. For generality, workflow tasks can return *references* that capture arbitrary inter-task communication. To enable the workflow system and therefore the end user to take control of recovery, we design task annotations that specify execution semantics such as nondeterminism. ExoFlow generalizes recovery for existing workflow applications ranging from ETL pipelines to stateful serverless workflows, while enabling further optimizations in task communication and recovery.

1 Introduction

A key requirement for distributed applications is *fault tolerance*, i.e. the appearance of execution without failures even when failures occur. In general, there is a tradeoff between recovery and run-time overhead. For example, logging generally adds higher execution overhead but reduces recovery time by allowing the system to only re-execute computations that failed [23]. Meanwhile, checkpointing reduces execution overhead but can impose higher recovery overhead as the system must roll back additional computation after a failure.

Current distributed systems often choose different tradeoff points between recovery and performance based on the application. For example, Apache Spark uses lineage-based logging for batch processing [48], and Apache Flink uses checkpointing for stream processing [19].

However, it is becoming increasingly common for different applications to be composed into heterogeneous pipelines. For example, a machine learning pipeline might use batch ingest to build a training dataset, then stream the data to a batch distributed training job to reduce latency and memory overhead. If we use a single recovery strategy for the entire pipeline, performance and recovery may be suboptimal because different recovery strategies are suited to different applications. Thus, to optimize end-to-end performance and recovery, we need to *compose different recovery strategies*.

Implementing multiple, interoperable recovery techniques within the same system, let alone a single one, is challenging. For example, Spark introduced "continuous processing" to reduce performance overheads for stream processing applications, but this mode does not yet provide exactly-once semantics during failures [10]. On the other hand, Flink has added a batch processing mode, but this required building an entirely separate recovery system from the streaming path [20].

Overall, these challenges have led to patchy support for applications that have diverse requirements in the recovery-performance tradeoff space. Users must choose between: (1) building on a single system, and face a fixed choice of performance vs. recovery overheads, or (2) stitching together multiple systems that offer different application-specific tradeoffs. The latter, however, is challenging and requires coordinating the flow of data, control, and recovery across disparate systems. This is true even in a single system, if using disparate execution modes such as batch vs. streaming.

In this paper, we propose a *universal workflow* system that enables a flexible choice of recovery vs. performance tradeoffs, even within the same application. A *workflow* is a directed acyclic graph (DAG) of *tasks*, where each task encapsulates a function call and edges between tasks represent data dependencies. Workflows are used to orchestrate execution across systems and thus prioritize generality. The DAG API is popular because it allows arbitrary application code in each task, from submitting a Spark job to invoking a microservice.

In contrast to other workflow systems, however, we *decouple the unit of execution from the unit of recovery*. In particular, ExoFlow guarantees fault tolerance by durably logging the workflow DAG and coordinating task checkpoint and recovery, while execution of the DAG is handled by a generic "backend". This has three key benefits. First, it enables heterogeneous application pipelines that need multiple recovery strategies for performance. Second, it augments existing distributed execution frameworks that provide only at-most-once or at-least-once semantics with strong exactly-once semantics. Third, it disaggregates the execution backend from recovery, allowing independent deployment and scaling.

Previous workflow systems provide exactly-once semantics but with significant limitations. For generality, workflow systems such as Apache Airflow [3] assume that each task is nondeterministic and may have side effects on external systems that in general cannot be rolled back. Thus, each task must synchronously checkpoint its outputs *before* they can be made visible to any downstream tasks. Otherwise, the system may have to re-execute the task in case of a failure. If the re-execution produces a different result, this can cause an inconsistent view among downstream tasks and external systems.

Thus, by assuming the worst, the workflow system has only one option of ensuring fault tolerance: no task can start before its upstream tasks have finished checkpointing all of their outputs. This limits the workflow system's ability to incorporate key optimizations often employed by application-specific frameworks that exploit the application's semantics. For example, large datasets passed between tasks can often be deterministically regenerated, making checkpointing unnecessary. In addition, while some tasks may indeed have external effects, e.g., starting a transaction on an external database, some effects can also be rolled back, e.g., by aborting the transaction.

Our goal is to hand control over recovery to ExoFlow and ultimately the end user. Thus, we use two key interfaces to enable awareness of application semantics. First, we extend the typical workflow DAG API with pluggable *first-class references* to enable more flexible workflow-internal communication. A workflow task can return references to its outputs, which the workflow system then passes to downstream tasks. In contrast, current workflow systems require the application to pass data by explicitly copying and checkpointing, which can be expensive for large data, or implicitly through external storage, which makes it difficult to guarantee exactly-once semantics. By using references to capture arbitrary data movement between workflow tasks, ExoFlow leverages third-party systems' existing communication and recovery mechanisms while retaining control over workflow-level recovery.

Second, we introduce user annotations that specify relevant task semantics, i.e. whether to checkpoint a task, whether the outputs are deterministic, and whether the task has externally visible outputs. Before execution, ExoFlow checks the safety of the user's specification. During execution, ExoFlow synchronizes task execution and checkpointing. During recovery, ExoFlow coordinates *rollback*, e.g., deletion of outputs from a previous execution, and task replay. For example, before executing a task with an externally visible output, ExoFlow will first synchronize upstream checkpoints to *commit* any nondeterministic outputs, i.e. ensure they will never be rolled back. This allows the user to flexibly and safely optimize the recovery technique.

ExoFlow is built on Ray [37] and consists of a per-workflow centralized controller, a pluggable checkpoint storage, and a pluggable execution backend. Centralizing controller logic makes it simple to guarantee recovery correctness. Meanwhile, checkpointing and execution are fully disaggregated, allowing these to be scaled independently of the controller.

We demonstrate the benefits of ExoFlow with two execution backends, the Ray framework and AWS Lambdas, both distributed frameworks that provide at-most-once or at-least-once tasks. We show that references can enable $\sim 5\times$ speedup for Spark data processing workflows compared to Apache Airflow, while task annotations enable 51% lower latency for transactional serverless workflows compared to Beldi [49]. These optimizations are possible because correctness is ultimately guaranteed by ExoFlow. These results also demonstrate ExoFlow's *universality*, as the system is not specific to data processing or serverless environments. In summary, our contributions are:

1. Decoupling execution from recovery to enable a flexible tradeoff between performance and fault tolerance.
2. Designing a universal workflow system that guarantees exactly-once DAG execution.
3. Demonstrating benefits for a diverse set of applications, including an ML pipeline, serverless transactions, and graph processing that mixes stream and batch execution.

2 Motivation

2.1 Overview of recovery strategies

We use *exactly-once semantics* as our correctness condition. This condition often implies application-specific correctness properties, such as global consistency in message-passing systems [23] or linearizability in storage systems [29].

More precisely, exactly-once semantics require all outputs to appear consistent with a physical execution where all inputs were processed without failures. In a workflow setting, the inputs are the DAG and the root task arguments. Outputs are values produced by a task that are viewed by others.

Output visibility can be *internal* or *external*. For example, values passed between tasks in Figure 1a are internal because they are viewed only by other tasks. Meanwhile, `(key,val)` is external because it is sent to a key-value store. Once outputs are made external, the workflow system no longer has control over how they will be used, e.g., via reads from external key-value store clients. Outputs can also be either *deterministically* or *nondeterministically* generated.

Output visibility and determinism are important because together they determine the recovery procedures that will guarantee exactly-once semantics (Figure 1b). For example, consider the cases if `A` is nondeterministic and we do not checkpoint `a_out` in Figure 1a. Suppose `C` views an initial value $\texttt{a\_out}_1$ and produces $\texttt{c\_out}_1$, but we lose $\texttt{a\_out}_1$ due to a failure. If we re-execute `A` to produce $\texttt{a\_out}_2$ and pass this to `B`, the outputs of `B` and `C` will not be consistent with a failure-free execution. To handle this case, we also need to "rollback" $\texttt{c\_out}_1$ and re-execute `C` on $\texttt{a\_out}_2$.

We encounter additional problems in the opposite case where `B` finishes and we then lose $\texttt{a\_out}_1$. `B` has already made `(key,val)` external and these values may depend on $\texttt{a\_out}_1$.

If we execute C on a_out_2, c_out will be inconsistent with (key,val). Thus, the only way to guarantee correctness in this case is to either: (1) "commit" a_out_1 before executing B, e.g., by checkpointing it, or (2) gain application semantics about how to roll back visibility of (key,val).

Meanwhile, deterministic outputs are safe to view as long as the task can be replayed on its original inputs and recomputed outputs can be deduplicated. The external output in Figure 1a can for example be deduplicated by attaching a deterministic req_id.

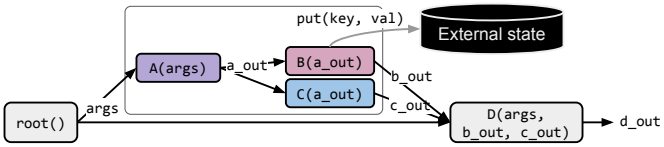
Solution space. Handling nondeterministic outputs is generally done in two ways: (1) global checkpointing and rollback on failure, or (2) logging and deterministic replay on failure [23]. Both "commit" a prefix of a failure-free execution by saving the outputs of a task frontier, allowing recovery to resume execution from a consistent set of intermediate outputs. Global checkpointing advances this frontier several tasks at a time and upon failure, rolls back to the last frontier to undo partially visible nondeterministic outputs. For outputs that cannot be rolled back, however, upstream nondeterministic outputs must first be committed by taking a global checkpoint. Logging-based methods advance the frontier one task at a time by committing each nondeterministic output before making it visible, thus avoiding additional rollback on failure.

Note that rollback and durability options vary based on output visibility. External outputs may be impossible to roll back, e.g., a transaction commit cannot be undone, or make durable, as third-party system context is not always serializable.

Current workflow systems guarantee exactly-once semantics by: (1) durably checkpointing each internal output before making it visible, and (2) requiring the developer to make external outputs idempotent and durable. This one-size-fits-all approach does not leverage application-specific recovery methods (Figure 1b). Furthermore, existing workflow systems have fundamental limits on internal outputs, usually because they must be sent between tasks through the workflow controller. Apache Airflow uses a database to coordinate tasks, which imposes a maximum output size on the order of MBs [3], and direct task communication in FaaS is limited [24]. Together, these force developers to use external outputs for much of their task communication [24, 42].

Our goal is to support different recovery methods in a single workflow system and even within a single application. The key insight behind ExoFlow is that knowing the DAG structure makes it simple to identify a consistent execution frontier, allowing the recovery methods before and after the frontier to be decoupled. For example, a_out is internal to the outlined sub-DAG in Figure 1a and thus its recovery method can be chosen flexibly as long as the inputs (args) and outputs (b_out,c_out,key,val) are consistent.

Thus, our solution consists of two parts. First, *references* enable ExoFlow to capture a broader range of inter-task communication as internal outputs, without being involved in the physical communication. This encourages recovery flexibility within a sub-DAG and recovery independence across sub-DAGs. References enable efficient passing of task outputs of any size and location as well as outputs that may not be serializable.



(a) Workflow DAG

	Internal	External
Nondeterministic	Commit output OR on failure, rollback visibility	Commit output *before* visibility OR if possible, rollback visibility on failure
Deterministic	Replay failed task(s) on previous inputs, dedupe outputs	Also dedupe external outputs

(b) Recovery strategies for workflow DAGs

Figure 1: **(a)** An example workflow with internal outputs (e.g., a_out) and external outputs (e.g., put(key,val)). **(b)** The most efficient recovery strategy depends on output visibility and nondeterminism.

Second, we support *annotations* to specify task semantics (checkpointing, nondeterminism, output visibility). These allow the system to determine recovery correctness before execution. The system "commits" the application to this specification by durably logging the DAG before execution, then coordinates and synchronizes task checkpoints during execution.

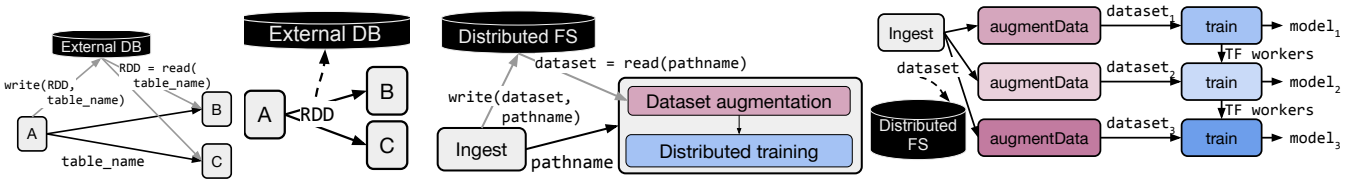
2.2 Applications

We use three representative applications to show the value of: (1) making workflow-internal outputs more flexible, and (2) exposing application semantics to the workflow controller:

1. Extract-transform-load (ETL) pipelines: Using references to pass large data as internal outputs.
2. Machine learning (ML) pipelines: Using references to pass large data and leveraging application semantics.
3. Serverless workflows: Leveraging application semantics to reduce recovery overheads, in a way that is agnostic to external systems.

ETL pipelines. Workflow systems such as Apache Airflow are commonly used to orchestrate extract-transform-load (ETL) pipelines composed of data processing jobs. Figure 2a shows an example in which a Spark job A performs batch data cleaning and writes the data to an external database, e.g., Delta Lake [11]. Jobs B and C then load the data for querying.

Current practice for exactly-once workflow execution requires all of A's outputs to be made durable *before* executing B and C. Synchronous checkpointing adds high overhead for large and distributed data. In addition, B and C must each reload the data, imposing an unnecessary memory copy. This is of course unnecessary if A is deterministic. Execution systems such as Spark leverage this property to natively support distributed in-memory caching. Ideally, A should pass its output as a cached RDD [48] to B and C (Figure 2b), avoiding the round trip to external storage, allowing B and C to share



(a) (b) (c) (d)

Figure 2: **(a)** ETL workflow today, using external outputs for communication. **(b)** The same ETL workflow with internal outputs only. **(c)** ML training workflow today, with external outputs and manual orchestration within a task. **(d)** The same ML workflow with internal outputs only, and orchestration is handled by the workflow system. Third-party framework state (`TF workers`) can be passed between workflow tasks.

physical memory, and enabling asynchronous checkpointing.

Building such optimizations into a workflow system would enable orchestration of arbitrary DAGs and third-party frameworks. However, even with awareness of task determinism, current workflow systems cannot execute Figure 2b due to limitations in workflow-internal data passing.

ML pipelines. Machine learning (ML) pipelines are similar to ETL pipelines, but with an ML application as the end consumer. This requires composition of traditional ETL systems with distributed ML frameworks for training and inference. Figure 2c shows a typical ML training workflow, in which training data is extracted and transformed in the `Ingest` task, then consumed by a distributed training job. Loading data into the training job may itself require complex and possibly distributed data processing, with computations such as random transforms to augment datasets [40]. Furthermore, datasets are often large enough that preprocessing must be pipelined with training to maximize GPU utilization.

Current workflow systems cannot effectively orchestrate within the training task, as training data and worker state must be passed through distributed memory. Expanding workflow-internal outputs would enable workflows such as Figure 2d. To reduce the overhead of recovery, however, the workflow system also requires application semantics, such as whether dataset augmentation is deterministic. Also, the model output can be consumed in a variety of ways, from local one-off testing during development to deployment on an ML serving system during production. All of these factors affect the optimal correct recovery strategy.

Serverless workflows. In the functions-as-a-service (FaaS) model, the user breaks their application into small functions that can be transparently executed and scaled without explicit resource provisioning. Serverless functions have a limited lifetime, all local state is transient, and failure handling is usually limited to function retries. This makes it challenging to build fault-tolerant nontrivial applications directly on FaaS [28].

Recently, *serverless workflow* systems [16, 46, 49] have gained popularity as a solution, especially for stateful applications. A common strategy for guaranteeing exactly-once execution is to provide fault-tolerant APIs to capture external outputs. For example, Figure 3 shows an example of a trip reservation workflow [25] that places the order if and only if both the hotel and flight were successfully reserved. Systems such as Aft [46], Beldi [49], and Boki [32] guarantee exactly-once semantics by providing a transactional key-value store to manage external output visibility.

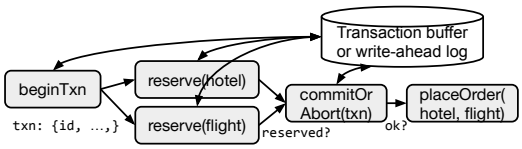


Figure 3: Serverless workflow systems [32, 46, 49] guarantee exactly-once semantics by interposing on all communication to external storage, e.g., through a transaction buffer, and explicitly managing visibility of these external effects.

However, each system offers different isolation levels that require different recovery strategies. Aft buffers uncommitted writes, which are safe to rollback, while Beldi and Boki use write-ahead logging. Thus, each system implements their own recovery procedures, e.g., durability and task re-execution.

ExoFlow factors out workflow recovery to enable flexibility and optimizations. Instead of providing opinionated APIs for external outputs, we treat external systems such as the transaction buffer in Figure 3 as a black box. ExoFlow does not interpose on the communication to this external system and instead requires that the application can specify task semantics such as whether the external effect can be rolled back. These semantics can be specified by a particular transaction system, i.e. Aft or Beldi.

3 API

3.1 Overview and requirements

ExoFlow is a general workflow layer that abstracts a workflow *backend*, i.e. a distributed framework providing at-least-once and/or at-most-once remote function invocation. We overview the application-facing API (Table 1) and requirements. The application must be able to: (1) differentiate deterministic tasks, and (2) for tasks with external outputs, ensure that the task is idempotent or specify an idempotent rollback function.

DAG interface. The application invokes workflow tasks and specifies arguments using `f.bind` (Table 1). The caller receives a `WorkflowDAG` that represents the task's output and that can be passed to other tasks as dependencies. Workflow execution is *lazy*: to evaluate a `WorkflowDAG`, the developer must `run` it. This is to simplify recovery, as the workflow system can check DAG-level properties before executing it.

The workflow backend should implement an RPC-like inter-

Workflow API	Semantics
`f.options(Opts).bind(Value | WorkflowDAG) → WorkflowDAG`	Create a workflow task `f`. Creates and returns a `WorkflowDAG`, whose value is lazily evaluated. The caller may pass the `WorkflowDAG` to another task. The return value of `f` can be a `WorkflowDAG`, i.e. a nested workflow.
`run(WorkflowDAG w, str name) → Value`	Run the workflow `w` and return the result. Optionally take a string identifier for this workflow.
`run_async(WorkflowDAG w, str name) → Fut`	Run the workflow `w` asynchronously and return a future that can be used to retrieve the result.
`Ref.get() → Value`	Used by the application to dereference to a value. `Ref` construction is backend-specific.
`bool Opts.checkpoint=True`	True if the task's output should be saved.
`bool Opts.deterministic=False`	True if outputs are deterministically generated.
`bool Opts.can_rollback=False`	True if task has no external outputs, or if they can be rolled back. If False, the task must be idempotent.
`Fn Opts.rollback=null`	If external outputs can be rolled back, a function to do so. The function must be idempotent, and any `WorkflowDAG` arguments must be a subset of the original workflow task `f`'s arguments.
`Ref._id() → ID`	Used by the workflow system to compare equality.
`Ref._checkpoint() → Fut[Value]`	Used by the workflow system to coordinate checkpointing. The `Value` is the checkpoint data or metadata.
`Ref._restore(Value)`	Used by the workflow system to reload from a saved checkpoint.

Table 1: Workflow API. Top: API calls exposed to the application. Middle: Task annotations specified by application or third-party library. Bottom: ExoFlow-internal `Ref` API, pluggable by execution backend.

face. Within a task, the application can invoke arbitrary local or distributed execution. For greater generality, we also adopt the *dynamic* task model [39]: tasks can dynamically invoke exactly-once nested workflows by returning a `WorkflowDAG`.

Task annotations. The application specifies semantics relevant to recovery at task invocation time (Table 1). The workflow system uses these to ensure correctness of: (1) coordination of distributed workflow checkpoints during execution, and (2) output rollback and task re-execution upon failure.

First, the application specifies whether to skip checkpointing a task's output. Note that the workflow system guarantees correctness, so this can be considered an optimization hint, e.g., to avoid recomputation for long tasks,

Next, the application can specify whether a task's outputs (both internal and external) are deterministic. This allows the workflow system to minimize rollback during recovery.

Finally, the application specifies whether a task can be rolled back and if yes, how to do so. Tasks with no external outputs, such as the data processing tasks in Figure 2, should set `can_rollback=True`. Tasks that have external outputs that cannot be rolled back should set `can_rollback=False` and ensure idempotence, as recovery may require re-execution.

Non-idempotent tasks with external outputs that can be rolled back should set `can_rollback=True` and the `rollback` callback. On failure, ExoFlow executes these rollback "tasks" in reverse dependency order before resuming execution. The rollback task can take any arguments available to the original workflow task, but the application must additionally guarantee that the rollback task is idempotent. For example, to implement the transaction in Figure 3, rollback for the `beginTxn` and `reserve` tasks could simply abort.

On `run`, ExoFlow checks the `WorkflowDAG` for specification errors and throws an exception if any are found. In particular, correctness requires the application to set checkpoints between each nondeterministic task and each downstream task with external output. Section 3.3 makes this precise.

Internal outputs. Direct task outputs are subject to limits of the execution backend. For greater flexibility, ExoFlow allows outputs to include `Refs` created by the task. `Refs` are (optionally) pluggable by the execution backend. They are intended to capture volatile outputs that would be expensive or complex to natively support in ExoFlow, e.g., large distributed data or third-party framework context. For an AWS Lambdas backend, for example, values can be stored in an external (volatile) key-value store and the key can be passed in a `Ref`. Other tasks can dynamically `get` the value, which can throw an error if the value is irretrievable due to failure.

`Refs` are uniquely identifiable objects typically containing backend-specific metadata. A task can only return `Refs` that it created or that were passed to it by an upstream task. Then, upon failure, ExoFlow can either restore the `Ref` from a checkpoint, or trace the DAG back to the creating task. On re-execution, the task need not return the same `Refs` as its original execution. For example, with the annotation `deterministic=True`, it is only necessary that the *value* of a returned `Ref` is deterministic; the `Ref` itself may have a nondeterministic ID. This is safe because ExoFlow simply cancels tasks using the previous `Refs` and re-executes with the new `Refs`.

By default, `Ref` values are *immutable*. This improves recovery efficiency, as it simplifies checkpointing and minimizes task rollback. To capture task outputs that are expensive or impossible to materialize, we also support *stateful* references, i.e. *actors* [30]. An `ActorRef` extends `Refs` with application-defined methods that execute on the actor's state (Listing 1). However, mutable state is more complex to recover efficiently and correctly. Thus, compared to `Refs`, we limit how `ActorRefs` can be passed between workflow tasks (Section 3.4).

3.2 Model

We present a formal model of workflows to more precisely capture the API and assumptions. A *workflow* $G = (V, E)$ is a directed acyclic graph with vertices V and edges E. Each vertex v_i has an associated function F_i, a function $\mathcal{N}_i$ representing a (potential) source of nondeterminism, a nullable rollback function $\mathcal{R}_i$, and the annotations described in Table 1.

A workflow execution produces one internal and one

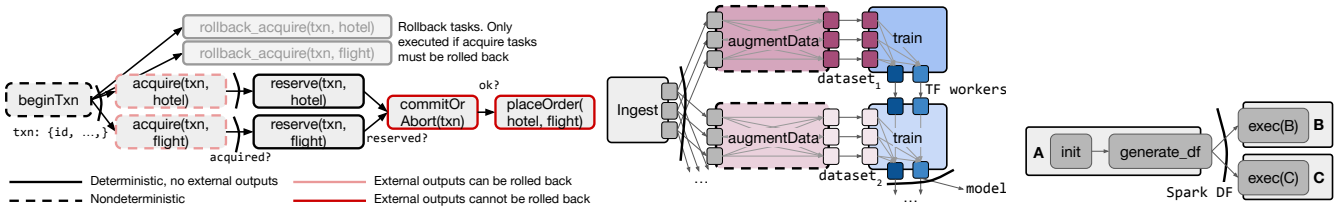


Figure 4: **(a)** Task annotations. Edge cuts represent `checkpoint=True`. **(b)** Passing references (small boxes) in an ML workflow. Blue `Refs` are actors that wrap TensorFlow worker state. **(c)** Passing an `ActorRef` in an ETL workflow. `B` and `C` call read-only methods on the Spark context actor.

external output per vertex, both nullable. For brevity, we do not consider tasks with multiple outputs. We denote an execution's outputs by (O_{Int}, O_{Ext}), both mappings from vertex to a single output o_{Int} or o_{Ext}. F_i outputs o_{Ext} by adding it to a global set $\mathcal{W}$, which can be read by other tasks and external processes. Each F_i takes as inputs:

- I_{Int}: Internal outputs of vertices with an edge to v_i.
- w_i: A read of $\mathcal{W}$, i.e. the external outputs so far.
- n_i: A nondeterministic value returned by $\mathcal{N}_i$.

In other words, an edge (v_i, v_j) indicates that v_i's internal output is passed to task v_j. Internal outputs passed between vertices are analogous to messages passed between processes in a message-passing model [23], except that the application must declare the "messages" (dependencies) before execution.

$\mathcal{N}_i$ captures nondeterministic inputs. For example, if F_i depends on the current time, then $\mathcal{N}_i$ returns the current time. We assume that if $\mathcal{N}_i$ reads some external state, the external state will not be rolled back (unless F_i is also rolled back via $\mathcal{R}_i$).

The correctness condition relates outputs to a failure-free execution. W denotes all possible sequences of reads of $\mathcal{W}$ by an external process.

Definition 1 (Consistency)**.** *O_{Int}, O_{Ext} are consistent with a workflow $G = (V, E)$ if W is monotonic and $\forall w \in W$:*

$$o_{Ext} \in w \Rightarrow \exists i, O_{Ext}[i] = o_{Ext} \bigwedge (O_{Int}[i], o_{Ext}) = F_i\big([O_{Int}[j] \mid (v_j, v_i) \in E], \{O_{Ext}[j] | v_j <_G v_i\}, \mathcal{N}_i()\big) \bigwedge \{O_{Ext}[j] | v_j <_G v_i\} \subseteq w$$

More simply, from an external process's perspective, if it sees an external output, then: (1) the same output was seen in all previous reads, (2) it must correspond to one invocation of some F_i, and (3) it also sees the external outputs of all predecessors of v_i. This is analogous to *global consistency* in message-passing [23], i.e. that every visible output has a corresponding task that created it. The goal is to provide a consistent execution under a crash failure model.

The application assumptions are as follows. For each v_i:

1. If v_j is concurrent with v_i ($v_i \not<_G v_j$ and $v_j \not<_G v_i$), then $F_i(I_{Int}, w_i, n_i) = F_i(I_{Int}, w_i \setminus \{O_{Ext}[j]\}, n_i)$.
2. If `deterministic=True`, then $\{O_{Ext}[j] | v_j <_G v_i\} \subseteq w_i, w_i' \implies F_i(I_{Int}, w_i, n_i) = F_i(I_{Int}, w_i', n_i')$.
3. If the o_{Ext} returned by F_i is not null, then either `can_rollback=False` or $\mathcal{R}_i$ is not null.
   (a) If `can_rollback=False`, then F_i is idempotent. That is, if $(o_{Int}, o_{Ext}) = F_i(I_{Int}, w, n_i)$ and $(o_{Int}', o_{Ext}') = F_i(I_{Int}, w', n_i')$, then $o_{Ext} = o_{Ext}'$.
   (b) If $\mathcal{R}_i$ is provided, then it is a deterministic and idempotent function of the task's internal inputs. If $(o_{Int}, o_{Ext}) = F_i(I_{Int}, w, n_i)$, then $\mathcal{R}_i(I_{Int})$ removes o_{Ext} from all past reads of $\mathcal{W}$.

(1) means that we do not consider cases in which a task v_i depends on a task v_j's external output, where v_i, v_j cannot be ordered in G. To ensure consistency, v_j's external output should be considered part of v_i's nondeterministic input, and v_j must set `can_rollback=False`. Regarding (3b), note that the meaning of removing o_{Ext} from past reads is application-dependent. For example, suppose F_i executes a transaction and $\mathcal{R}_i$ aborts the transaction; if uncommitted reads are allowed, then $\mathcal{R}_i$ does not need to roll back the reader.

Nested tasks and references. While not explicitly captured in the above model, nested tasks can be thought of as tasks that expand into a sub-workflow. `Refs` and `ActorRefs` are native data types that can be returned in a function's internal output. Because actors are mutable, `ActorRefs` are versioned: if a caller writes to an actor by calling a method on its `ActorRef`, the caller's resulting `ActorRef` is of a different version. This becomes relevant in Section 3.4, which discusses the rules that the application must follow to ensure exactly-once semantics when `ActorRefs` are passed between workflow tasks.

3.3 Guaranteeing exactly-once execution

Task annotations simplify the decision of when to commit task outputs. To illustrate this, we use Figure 4a, a modified version of the workflow described in Figure 3. We show the annotations for a workflow using an external two-phase locking (2PL) transaction system. `beginTxn` generates a transaction context with a random `txn_id`. The `acquire` tasks each attempt to acquire a lock on an external table row. If this is successful, we attempt to `reserve` the flight and hotel if available, then finally commit the transaction and place the order if both succeed. The cuts in Figure 4a indicate `checkpoint=True`.

As an example, we first consider the `acquire` and `commitOrAbort` tasks. `acquire` tasks are nondeterministic because they depend on the run-time state of the external table. `commitOrAbort` has `can_rollback=False` because it is impossible to abort a committed transaction and vice versa. Although `ac-`

quire can be rolled back (e.g., by aborting the transaction and releasing the lock), once we have started the commitOrAbort task, it is no longer safe to do so because the transaction may already be committed. Thus, we must ensure that both acquire outputs are saved before commitOrAbort starts. We can generalize this rule for the application as follows:

Invariant 1 (External output commit)**.** *For each workflow task v_i with deterministic=False, let G be the minimal subgraph that contains v_i and all downstream tasks (tasks for which there is a path from v_i). Then, for each workflow task v_j with can_rollback=False in G, there must exist a vertex cut that partitions v_i from v_j such that all tasks in the cut have checkpoint=True.*

Intuitively the vertex cut of the sub-DAG defines a commit point for the nondeterministic output of v_i. There may exist multiple such cuts. For example, another acceptable specification in Figure 4a is to instead checkpoint the reserve outputs.

ExoFlow guarantees that at least one task frontier is fully checkpointed by the time commitOrAbort (v_j) starts. Interestingly, this also tells us that we do not need to commit the acquire outputs synchronously. In particular, the reserve tasks in this case are deterministic, as their outputs depend only on whether the lock was acquired and the value stored in the external table, which cannot be modified while locked. Furthermore, their external outputs are not visible while the lock is held. Thus, in this case, it is safe to annotate the reserve tasks with deterministic=True and can_rollback=True. Together, these annotations allow ExoFlow to *overlap* the checkpoint of acquire's outputs with execution of the reserve tasks, as long as the checkpoints are synchronized before commitOrAbort.

There is a similar requirement for rollback tasks. The rollback tasks in Figure 4a are conditionally invoked by the workflow system to undo external outputs of the acquire tasks. We must ensure that all inputs to the original acquire task are recoverable *before* execution. Otherwise, if the rollback task and its inputs fail simultaneously, it will be impossible to finish rollback. Thus, in Figure 4a, the application must set checkpoint=True for beginTxn, and ExoFlow synchronizes this checkpoint before executing the acquire tasks.

Invariant 2 (Rollback durability)**.** *For each path beginning at a task v_i with deterministic=False and ending at a task v_j that has a rollback function R_j, there must exist at least one vertex along the path with checkpoint=True.*

Unlike Invariant 1, here we only require checkpointing a single task to handle nondeterminism, as the availability of a rollback function R_j means that we do not need to commit to the original output. The checkpointed task can also be a task other than v_i or v_j. For example, if there were additional deterministic tasks between beginTxn (T) and rollback_acquire (R), then checkpointing any is sufficient.

Both invariants can be easily checked by walking the DAG passed to run. If an invariant is not met, the system throws an exception to the user. Annotations do therefore require user cooperation, but note that a user with minimal performance needs can use the defaults in Table 1. This specification trivially satisfies the invariants and indeed corresponds to current workflow systems that commit all task outputs. Section 4 describes how ExoFlow leverages the invariants to improve run-time performance for more sophisticated specifications.

```
@ray.remote
class SparkActor:
  def __init__(self):
    self.spark_context = connect(); self.df = None
  def generate_df(self):
    self.df = generate_df(self.spark_context).cache()
  @const
  def exec(self, seed: int) -> int:
    return exec(self.df, seed=seed).count()
  def _checkpoint(self):
    return self.spark_context.save(self.df)
  def _restore(self, path):
    self.df = self.spark_context.load_df(path)
```

Listing 1: Psuedocode for passing a Spark DataFrame by actor. The execution backend implements the actor. Public methods are user-defined. Methods prepended by _ are called internally by ExoFlow.

Note that the system will not durably record a nested workflow returned by a task with checkpoint=False. To simplify recovery, we disallow sub-tasks with checkpoint=True, as we may lose all references to these checkpoints upon failure. We also disallow can_rollback=False and rollback, as these are challenging to recover without workflow durability.

3.4 References

Immutable Refs enable efficient passing of large and distributed data between workflow tasks. For example, Figure 4b shows how the Ingest task from Figure 2d can use Refs to return distributed in-memory data. ExoFlow tracks inter-task Ref dependencies for recovery purposes, while the execution backend handles intra-task execution (e.g., get).

Some cases require stateful actors for performance. For example, the blue boxes passed between train tasks in Figure 4b are ActorRefs representing a training worker's state, e.g., a Distributed TensorFlow session. This helps avoid expensive materialization, such as the worker's local model copy.

Guaranteeing exactly-once semantics for state is challenging. If one task writes the ActorRef's state, the output is visible to any other task holding a reference to the same actor. This can cause cascading rollbacks on failure depending on how ActorRefs are passed. Furthermore, checkpointing is more challenging if multiple tasks write concurrently to the actor, as the system must ensure that the actor checkpoint is consistent.

To simplify recovery, we limit ActorRef passing to two patterns, analogous to a read-write lock. By default, the ActorRef is in "write" mode. In this mode, only one workflow task may have a reference to the actor at a time. That task can call any actor methods as long as they finish before the task returns. For example, in Figure 4b, only one train task refers

to each actor at a time. ExoFlow can then checkpoint the actors' state between tasks, and on failure, roll back the actors with the workflow. This pattern is useful for abstracting and checkpointing distributed workers in third-party frameworks such as Distributed TensorFlow [9] and Flink [20].

If there are multiple concurrent workflow tasks with a reference to the same actor, however, the tasks are restricted to read-only methods annotated by the user, as shown in Listing 1. Figure 4c shows an expanded Figure 2b in which we use an `ActorRef` to capture a Spark DataFrame. Initially, `A` has the only `ActorRef`, so it can write to the actor's state (`generate_df`). `B` and `C` share the actor concurrently, however, and so they are limited to read-only methods (`exec`). Invoking a write method such as `generate_df` would throw a run-time error.

Similar to a read-write lock, ExoFlow can only provide correctness if the application respects certain conditions. In particular, the workflow tasks must explicitly pass `ActorRefs` through their outputs and arguments. Any other `ActorRefs` cannot be tracked by ExoFlow and exactly-once semantics is not guaranteed, similar to reading a variable without holding the lock. Also, while methods may be called asynchronously on an `ActorRef`, a workflow task must synchronize any outstanding calls to an actor before returning.

4 Architecture

We describe the ExoFlow design and the requirements of the pluggable execution backend and persistent storage. The workflow controller is a long-running service that can be sharded by workflow (Figure 5). Persistent storage can be implemented by any durable blob storage supporting puts and gets with read-after-write consistency, such as Amazon S3. The execution backend should implement a *remote function invocation* interface, used by the controller to scale checkpointing and task execution. The backend should provide: (1) ability to detect and report task and `Ref` failures, and (2) guarantee no resource leaks for failed task execution and `Refs`.

The controller runs as an event loop with the following events: task or checkpoint completes, and task or checkpoint failed. All critical workflow state, such as the workflow DAG, is cached by the workflow controller and written-through to persistent storage, making it simple to also recover the workflow controller. On restart, the controller simply scans the storage for any unfinished workflows, and re-`runs` to completion.

4.1 Workflow execution

The workflow control layer is implemented using the system Ray [37]. Ray provides remote task invocation, distributed immutable memory, and distributed actors. However, Ray only provides at-most-once or at-least-once guarantees and lacks built-in persistence for memory and actors. Thus, Ray tasks and actors are distinct from workflow tasks and actors, which execute exactly-once and can be natively checkpointed.

We use Ray actors to implement the workflow controller and task executors (Figure 5). The controller uses Ray's

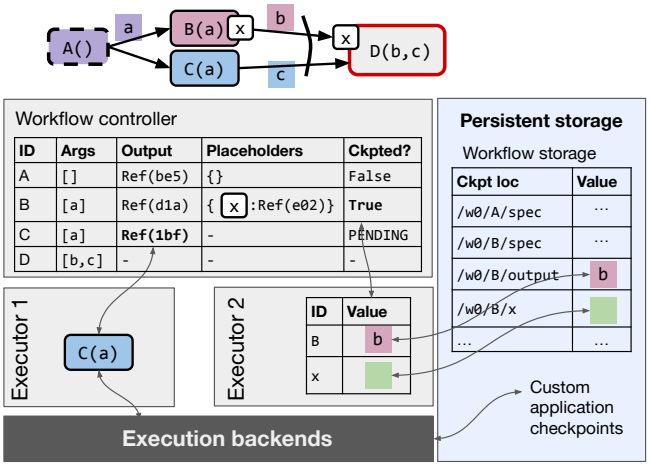


Figure 5: Workflow architecture. The controller and executors are RPC-like services built using Ray actors. Each invocation on these services returns a distributed future (system-internal `Refs`).

distributed futures [47] to coordinate task execution and checkpointing. Distributed futures are an asynchronous extension of RPC where each invocation returns a future pointing to the eventual and possibly remote return value. Ray actors and distributed futures also directly implement application-facing references (Section 3).

We build on Ray for three reasons: (1) futures make it simple for the controller to manage concurrent task execution and checkpointing, (2) passing remote values by reference avoids bottlenecks from large task outputs being passed directly through the centralized controller, and (3) the RPC-like interface straightforwardly and efficiently wraps other execution backends. For example, the Lambdas backend is implemented by wrapping a synchronous Lambda invocation in a Ray task.

The controller is a state machine where the state describes the current execution status of a workflow DAG and is persisted in storage. On `run`, the controller logs the workflow DAG specification (arguments, `Opts`, etc.) to durable storage and triggers execution. On each iteration of the event loop, the controller may select a workflow task whose inputs are ready and submit the task to an executor. For example, in Figure 5, the controller submits `C` to executor 1 and immediately receives back the distributed future `Ref(1bf)`. The controller uses this system-internal `Ref` to wait for task completion, and then passes it to downstream workflow tasks (e.g., `D`).

Checkpointing is carried out asynchronously by background threads on the executors, enabling parallel and distributed checkpoints that are not bottlenecked by the centralized controller. To checkpoint an output, the executor asynchronously writes a copy to a deterministic storage location (e.g., `w0/B/output` in Figure 5). The controller considers the checkpoint done once it is fully written. For convenience, the controller can also synchronize the checkpoint by requesting a signal from the executor (controller to executor 2 in Figure 5).

Checkpoint synchronization is required: (1) at the end

of a workflow, (2) before executing a task with `can_rollback=False`, and (3) before executing a task with a `rollback` option. Section 6 evaluates a simple policy that synchronizes all pending checkpoints for a workflow in any of these cases and shows that this provides sufficient performance for key applications. A more sophisticated policy may synchronize only the minimum necessary.

ExoFlow handles passing and checkpointing application references (Section 3.4). When a task finishes, the executor replaces any `Refs` and `ActorRefs` appearing in the task's output with placeholders, e.g., `x` in Figure 5. When passing the output to another task, the controller also passes a list of concrete references (`Ref(e02)` for `x`) used by the executor to fill the placeholders. Task checkpoints include a list of `Ref` checkpoint locations, which are written in parallel and distributed fashion. The controller restores and swaps `Refs` after a failure.

If a workflow task returns a `WorkflowDAG` as its output, the controller simply records the sub-workflow (if `checkpoint=True`), points the output of the parent task to the output of the sub-workflow, then resumes execution.

4.2 Workflow recovery

The controller handles task and checkpoint failures. In both cases, the protocol rolls back any previous outputs as needed, then rolls "forward" by re-executing workflow tasks.

The first step is to determine the re-execution task frontier. For example, suppose `C` in Figure 5 fails because we lost `A`'s cached output `Ref(be5)`. Then, we walk the DAG backwards from `C` and add each visited task node to the re-execution set. For each task, we check argument availability, i.e. whether the value has a checkpoint or a live `Ref`. If all arguments are available, then we terminate. Else, we add the tasks that create the arguments (`A`) to the re-execution set. If a visited task has `deterministic=False`, then we also add all tasks downstream to the re-execution set. Thus, if `C` fails and we need to re-execute *A*, we also re-execute *B*, even though it has a checkpoint.

From the re-execution task set, we carry out rollback. In reverse-topological order of the re-execution set, we first clear any cached output `Refs` and output checkpoints, e.g., `/w0/B/output` and `/w0/B/x` for `B`. If it has a `rollback` task, then we re-execute this task, using the same protocol as normal task execution. Finally, we resume workflow execution as normal, starting from the earliest task frontier of the re-execution set.

Critical controller state is persisted, so recovering from controller failure is straightforward. On failure, all in-memory controller state (the table in Figure 5) is wiped, including any `Refs`. On restart, the controller simply scans persistent storage for incomplete workflows, rebuilds its in-memory table, then re-executes them using the described protocol.

Correctness. We provide informal proofs that the final outputs are consistent (Definition 1). During normal execution, this follows from the execution protocol: starting from a consistent prefix of outputs, executing a task will produce another consistent prefix.

For recovery, we first consider reconstruction of internal outputs, i.e. values returned by workflow tasks. If the task is deterministic, then the reconstructed output will match the original. If the task is nondeterministic, then the described rollback procedure returns execution to a consistent prefix that does not include any results downstream to the original output.

Next, we consider external outputs: tasks with `can_rollback=False` or `rollback` defined. For a task T with `can_rollback=False`, the application guarantees idempotence, so it is enough to show that once T begins, the failure-free execution will include the same inputs for T. To show this, we rely on Invariant 1 (Section 3.3) and checkpoint synchronization (Section 4.1). The system synchronizes the partition provided by Invariant 1 before submitting T; thus once T begins, any future recovery procedure will never add T to the rollback set.

If T instead has `rollback` defined, we must show that if T fails, `rollback` will complete with the same view of inputs as T's previous execution, before re-executing T. Invariant 2 and checkpoint synchronization guarantee that we can deterministically and idempotently recreate `rollback`'s original inputs.

Correctness also requires preventing conflicts between different executions of the same task. For task checkpoints, if the backend's failure detection for executors is reliable, then by the time we re-execute T, we can be sure that there is no concurrent checkpoint in progress. Under unreliable failure detection, the ExoFlow controller assigns unique checkpoint locations to prevent races between concurrent executions. This requires one extra durable write before each task execution to record the expected checkpoint location.

For a task that returns `Refs` or `ActorRefs`, the execution backend can provide reliable failure detection for references by killing all copies of a `Ref` *before* reporting failure to ExoFlow. Alternatively, a safe and efficient method that works for both crash and fail-stop failures is to generate unique references for each execution.

4.3 Execution backends

Integration. ExoFlow references are compatible with existing third-party mechanisms for task communication and recovery. For example, Ray does not provide exactly-once semantics, but it does automatically reconstruct `Refs` created by deterministic (at-least-once) tasks [47]. ExoFlow encourages hierarchical recovery, wherein the execution backend can attempt to handle `Ref` failures first, then throw unrecoverable errors up to the workflow controller.

ExoFlow is compatible with backends that use logging and checkpointing. In general, log-based tasks would use `deterministic=True` and `can_rollback=False` annotations, while checkpoint-based tasks would use `deterministic=False` and `can_rollback=True`. The backend can also directly leverage ExoFlow for checkpointing instead of supplying a user-defined `rollback` function; this shifts the responsibility of checkpoint coordination to ExoFlow and automatically enables optimizations such as overlapping with execution.

Preventing leaks. The workflow layer ensures that previous `Refs` and pending checkpoints do not leak; invalid `Refs` and checkpoints are dropped during rollback. The execution backend must additionally prevent resource leaks for dead `Refs`. Dead `Refs` can be deleted via reference counting (the controller calls back to the backend once a `Ref` goes out of scope) or garbage collection (the backend scans the controller's in-memory state for dead `Refs`).

5 Implementation

ExoFlow is built on Ray v2.0.1, which uses gRPC [6] for tasks and actors and a custom shared-memory object store for `Ref` storage [37]. ExoFlow is implemented as a Ray Python program in 4k LoC.

We implemented two execution backends for ExoFlow: Ray itself ("ExoFlow-Ray") and the serverless FaaS offering AWS Lambdas ("ExoFlow-Lambdas"). In each case, a typical deployment would use one Ray node to host the ExoFlow controller. In ExoFlow-Lambdas, the controller node takes the place of the gateway provided by AWS for their proprietary serverless workflow offering (Step Functions).

We chose to implement ExoFlow on Ray for three reasons:

1. Support for first-class references to immutable data, which we use to implement `Refs`.
2. Support for actors (stateful workers), which we use to implement `ActorRefs`.
3. Low task and actor overhead, similar to pure RPC.

We also use Ray actors to implement executors. Workflow tasks are stateless, but we use actors to store execution state about checkpoints that are pending after task completion.

To build ExoFlow on another actor system such as Akka [2] or Orleans [13], we must implement `Refs`. This is straightforward for workloads that only pass small data. For data-intensive workflows, one can build a custom in-memory store that is tightly coupled to executors, as in Ray, or use an external key-value store. The latter requires low implementation effort, but may result in poor locality. It is ideal if the execution backend cannot be modified, e.g., to support values larger than the Lambdas response size in Exoflow-Lambdas.

6 Evaluation

Our evaluation covers the following questions:

1. What overheads does ExoFlow add to at-least-once or at-most-once execution backends?
2. How can applications leverage first-class references and task annotations to have greater flexibility in recovery?
3. How does this flexibility in recovery strategy affect performance during execution and recovery?

We compare primarily against these baselines: (1) exactly-once workflow systems: Airflow [3], "standard mode" AWS Step Functions [14], and the serverless workflow system Beldi [49]; and (2) at-least-once distributed DAG systems: "express mode" AWS Step Functions [14] and Ray [37].

Given the high execution overheads of exactly-once workflow systems such as Airflow (Section 6.4), to fairly address questions (1) and (3), we also compare against the following ExoFlow modes:

1. `SyncCkpt`: Task outputs are synchronized before executing downstream tasks. This is used to simulate the recovery strategy of exactly-once workflow systems such as Airflow.
2. `NoCkpt`: All task outputs except the final are skipped. This is used to simulate the recovery strategy of an at-least-once or at-most-once system. The application must guarantee that all tasks are deterministic and idempotent to achieve exactly-once semantics.
3. `AsyncCkpt`: The default mode of ExoFlow. Task outputs are only synchronized where necessary, to provide exactly-once semantics.

We conduct all of the experiments using the AWS cloud, specifically in the us-east-1 region. ExoFlow and execution backends are hosted on EC2 and use Amazon S3 (or EFS in Section 6.2) for persistent storage.

6.1 ML training pipelines

We show how ExoFlow enables a flexible recovery-performance tradeoffs for the workflow in Figure 2d. We use an image classification example adapted from Azure MLOps [4]. An ETL `Ingest` task (1 r3.2xlarge node) downloads the compressed data from S3. "1×" in Figure 6 indicates one data copy with 569 raw image files and total size 225MB. The task loads the images into memory, and performs data cleaning and normalization with at-least-once parallel Ray tasks. The dataset (1.4GB of memory per data copy) is partitioned and passed using `Refs` to the dataset augmentation tasks, via Ray's shared-memory object store. Dataset augmentation again uses Ray at-least-once tasks to apply random cropping, flipping, and color adjustments to the base dataset, once per epoch. Dataset augmentation requires repeatedly processing the same dataset in a tight loop with training. Therefore, the dataset augmentation stage accumulates a total intermediate and checkpoint size of 67GB and 18GB respectively, per data copy. Training tasks are colocated and pipelined with dataset augmentation (1 g4dn.12xlarge node, 4 NVIDIA T4 GPUs). We use PyTorch data-parallel distributed training and the ConvNeXt Tiny (28.6M parameters) model. PyTorch workers are passed using `ActorRefs`.

Figure 6L shows end-to-end duration of 25 epochs without failures of different ExoFlow recovery modes, as a function of dataset size. Here, we also include `Selective AsyncCkpt` (skip checkpointing dataset augmentation outputs) and `Workflow Tasks` (include at-least-once Ray tasks for data processing in the workflow DAG instead of passing volatile `Refs`).

Duration predictably grows approximately linearly with the dataset size for all strategies. The overhead of `Workflow Tasks` is high because each data processing task is durably (and unnecessarily) logged as part of the workflow. For the same

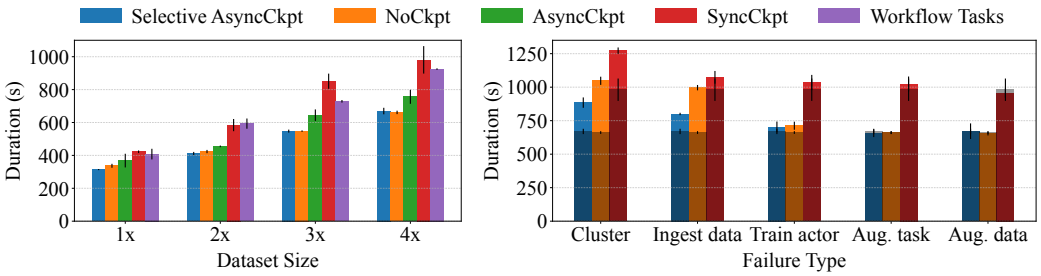


Figure 6: End-to-end duration for the ML workflow application shown in Figures 2d and 4b. **Left:** End-to-end duration without failure. **Right:** End-to-end duration with different failure types. The shadow represents the execution time without failure.

workflow graph, the overhead for larger data varies depending on the recovery strategy. `NoCkpt` represents the best possible performance, where only the final model is checkpointed. `SyncCkpt` represents existing workflow systems (Figure 2c) and its overhead grows the most because checkpointing overhead grows faster than computation overhead. `AsyncCkpt`'s overhead grows less because checkpointing of augmented datasets is overlapped with training tasks. `Selective AsyncCkpt` has nearly identical duration as `NoCkpt` because the `Ingest` checkpoint is perfectly overlapped with training tasks.

Meanwhile, Figure 6R shows end-to-end duration in different failure scenarios compared to normal run-time execution (dark): whole cluster failure (including the ExoFlow controller); in-memory ingest data lost; PyTorch worker actor lost; augmentation task lost; and in-memory augmented data lost. Here, we see the tradeoff between recovery and performance. `SyncCkpt` has similar or better recovery time overhead than `NoCkpt` for cluster and ingest data failures because it avoids re-executing the `Ingest` task, but overall it does worse because of high normal run-time overhead. `Selective AsyncCkpt` checkpoints the `Ingest` data asynchronously, so recovering from cluster and ingest data failures is fast because it simply restores the `Refs` from the checkpoint. Together, Figure 6L and R demonstrate how *the developer can flexibly choose the best recovery strategy*.

Figure 6R also demonstrates ExoFlow's *broad failure coverage and ability to integrate with Ray's built-in recovery*: Ray automatically reconstructs deterministic data processing results but does not handle persistence or actor recovery [47]. Thus, ExoFlow handles the first four failures, while Ray handles the last. Recovery for the last two failures is fast because rollback and checkpoint restore are unnecessary.

6.2 Stateful serverless workflows

We compare ExoFlow on a travel reservation benchmark [26] to Beldi [49], a recent system for fault-tolerant and transactional stateful serverless workflows that uses *intent logging* to ensure exactly-once semantics. Our implementation uses Beldi's APIs for reading and writing state but the ExoFlow controller with an AWS Lambdas backend for workflow execution and recovery. We use a single m5.16xlarge instance to host ExoFlow and EFS for persistent storage, which provides lower latency than S3. The benchmark procedure follows [49], and we report response latency in Figure 7a.

ExoFlow achieves about 51% lower p50 latency than Beldi for request rates up to 400, despite using the same execution system (AWS Lambdas) and state APIs (Beldi). This is because most of the workflows have deterministic computation and no external effects (i.e. read-only), so the additional logging used by Beldi is unnecessary for correctness. Furthermore, Beldi schedules an additional Lambda function to orchestrate others, while ExoFlow directly schedules Lambdas. When requests/s is higher than 700, ExoFlow's median latency is greater than Beldi's. This is due to the Lambdas invocation bottleneck at the ExoFlow controller node and can be easily removed through sharding across workflows. The Lambdas gateway used in Beldi is likely sharded internally.

The use of ExoFlow as a Lambdas gateway has benefits in recovery time. Figure 7a also shows latency with a 10% failure rate for all Lambdas. ExoFlow directly invokes Lambdas, so it can detect failures and recover virtually instantaneously, resulting in 0-31% extra overhead in p99 latency. In contrast, Beldi is fully decentralized and relies on timeouts for recovery correctness. Thus, although Beldi-style logging may reduce re-execution on recovery, the actual recovery time would be lower-bounded by a timeout ([49] evaluates 1min as a possible lower bound).

Figure 7b further demonstrates the performance benefit of exposing application semantics to the workflow system. We report latency of the most complex workflow in the benchmark, the trip reservation request described in Figure 3. Beldi implements the transaction using two-phase locking (2PL). We demonstrate progressive improvement over the original solution by varying the execution and recovery strategy. First, we eliminate Beldi logs for dynamic task invocation, as the DAG can be easily specified upfront, reducing p50 and p99 latency by 17% and 25% respectively (`-WAL`). Next, we parallelize the hotel and flight reservation tasks, further reducing p50 and p99 latency by 17% and 15% respectively (`+parallel`). Beldi executes these tasks sequentially because asynchronous invocation does not allow retrieval of the reply. Finally, we split each reservation task into two steps: lock acquisition and reservation, as seen in Figure 4a. `-async` shows that with synchronous checkpoints,

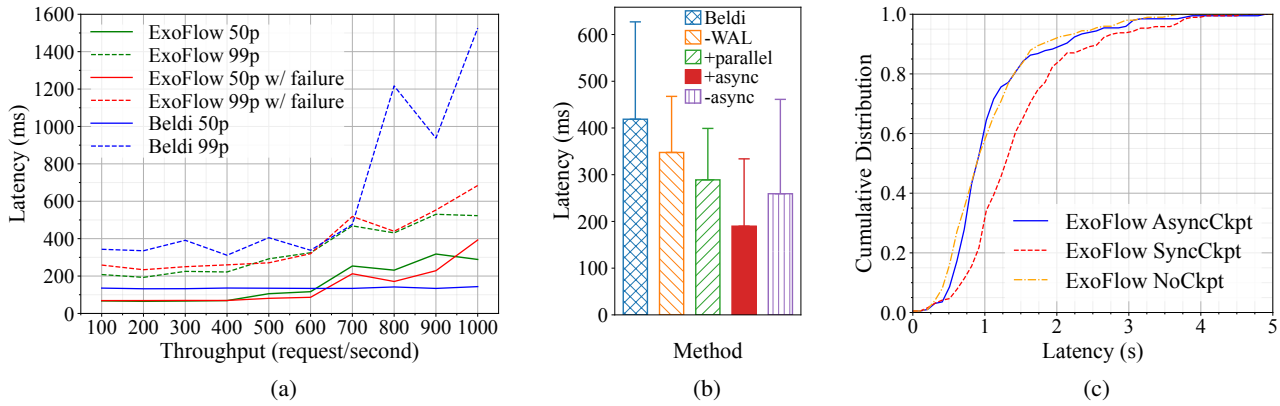


Figure 7: **(a)** Response latency percentile for a serverless travel reservation benchmark [25]. **(b)** Median latency of the trip reservation request from the travel reservation benchmark. Error bar represents 99-percentile latency. **(c)** Latency CDF of online-offline graph processing.

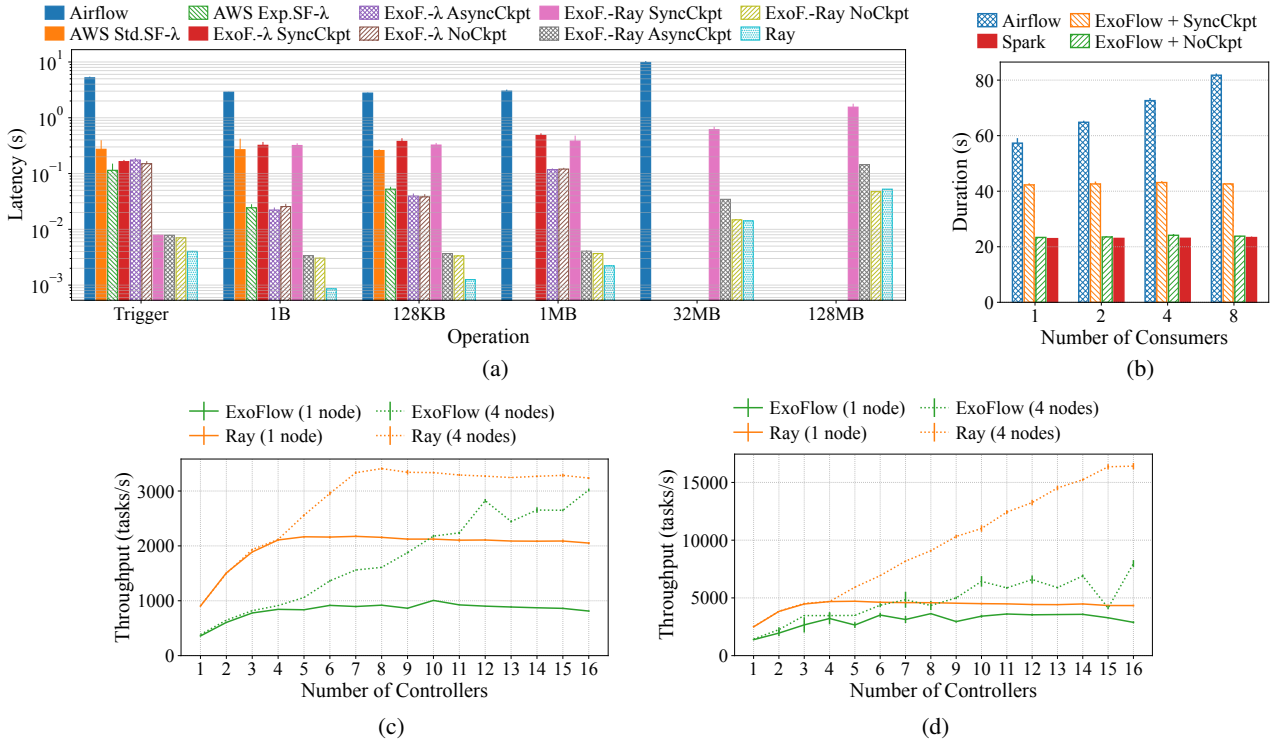


Figure 8: Microbenchmarks. **(a)** Triggering and data passing latency of ExoFlow and other workflow systems, using AWS Lambda (λ) and Ray as execution backends. Missing bars indicate limitations in inter-task communication. **(b)** End-to-end run time for the ETL workflow shown in Figures 2b and 4c, compared with Airflow and native Spark. **(c, d)** Maximum task throughput (c: 1 task/DAG; d: 100 tasks/DAG) of 10k tasks, compared against Ray as an optimal baseline, on 1 node and 4 nodes.

this actually increases latency due to the added task. However, `+async` shows that by overlapping checkpointing with execution, we can further reduce p50 and p99 latency by 34% and 16% respectively, without compromising correctness.

6.3 Online-offline graph processing

Distributed graph processing systems can be generally divided into stream vs. batch processing [36]. Streaming systems can handle continuous updates and produce timely results, but may not offer the same precision as batch systems.

We use references in ExoFlow to link stream and batch graph processing, producing a single application that can both handle online queries and produce periodic exact results. We use Ray actors to implement a version of Kineograph [21], a streaming graph processing system that uses distributed snapshots for consistency. Each workflow task ingests one epoch of incoming graph updates to compute a graph snapshot and an online approximate result, and we periodically pass the snapshot in-memory to another workflow task that uses Spark to compute the full result.

We evaluate on the SNAP Twitter follower network

dataset [33] (41M nodes and 1.5B edges), with each input record representing an edge insert event. We run the push-model TunkRank algorithm used by Kineograph to compute Twitter user influences on a 3-node r3.8xlarge cluster, 1 for streaming and 2 for the Spark cluster. We use two Ray actors to process the input stream and use ExoFlow to checkpoint and pass the `ActorRefs` between streaming tasks. Each streaming task represents a 10-second epoch and also returns 4 `Refs` that represent the partitioned graph snapshot. These `Refs` are passed to a Spark task every 20 epochs. Latency is reported for 200 epochs, after an initial warmup of 150 epochs. The average digestion rate is 44.94k tweets per second with our dataset. Kineograph achieves about 40k tweets per second with 2 ingest node + 48 graph nodes with a similar setting. We outperform Kineograph likely because we utilize shared memory for data passing, with more powerful hardware, which significantly reduces overhead of data pushing.

Figure 7c shows a CDF for latency from input event to the earliest time that the event is reflected in a streaming task's output (although inconsistent results can be returned earlier by querying the ingest actors directly). `AsyncCkpt` allows the snapshot to be viewed before it is checkpointed. `NoCkpt` has impractical recovery overhead, but we use it as a performance baseline. `AsyncCkpt` achieves similar latency as `NoCkpt`, meaning that checkpointing overhead remains stable as the graph grows larger; this is because streaming tasks pass through previous `Refs` that are already checkpointed, so ExoFlow only checkpoints new data on each epoch. `SyncCkpt` is similar to Kineograph, checkpointing the snapshot before making it visible, and adds less than 1s latency. Finally, the error rate of the online results and the batch processing task duration both grow linearly over time, confirming the tradeoffs between batch vs. stream processing.

6.4 Microbenchmarks

Latency. With equivalent backends, ExoFlow matches or reduces execution overheads of existing workflow systems while enabling more flexible inter-task communication. Figure 8a (1 m5.8xlarge instance) shows the latency of workflow execution ("Trigger") and task execution with different size arguments. We use exactly-once systems (Airflow [3], AWS Standard Step Function [14]) and at-least-once systems (AWS Express Step Function [14], Ray [37]) as baselines. Airflow is an industrial custom-built workflow system while Step Functions are the AWS-native workflow offering for Lambdas.

First, with the Lambdas backend, ExoFlow has similar trigger latency as AWS Standard Step Function. Airflow has generally high overhead due to coordinating execution through a database, which can easily lead to inefficient scans.

"1B" in Figure 8a compares minimum task execution latency. ExoFlow-Lambdas achieves comparable latency as AWS Step Functions, as the primary overheads for exactly-once and at-least-once execution come from durability and Lambdas invocation, respectively. ExoFlow-Ray improves upon the latter as it uses Ray for execution.

Finally, we compare the ability to pass large data between tasks. AWS Step Functions limit data passing to 256KB, but plain Lambdas have a size limit of 6MB. Thus, ExoFlow-Lambdas can actually support larger data sizes. This could be improved further with `Refs`, e.g., with Redis [44] for distributed memory. Airflow's XCom [1] can support slightly larger data but is fundamentally limited by its database-centric design. Meanwhile, ExoFlow-Ray uses Ray `Refs` for efficient data passing. The gap between `AsyncCkpt` and `NoCkpt` latency is small but grows with data size; although the checkpoint is asynchronous, ExoFlow synchronously copies the data to guard against concurrent writes.

In summary, ExoFlow's low execution overheads make it a practical replacement for existing workflow systems, and it enables greater flexibility in task communication and recovery.

Data sharing for ETL. We evaluate ExoFlow against Airflow for a Spark workflow similar to Figure 2b (1 m5.8xlarge instance, 4GB Spark memory). Figure 8b measures total run time for a workflow that uses Spark to generate a 1GB random dataset, followed by multiple downstream tasks that consume the data with data sampling Spark jobs. Such a workflow requires orchestration across Spark jobs, which Spark does not provide, and is therefore often run on a workflow orchestrator such as Airflow.

Airflow run time grows proportionately with the number of consumers because they cannot share data in-memory. Meanwhile, ExoFlow scales well even with synchronous checkpointing because consumers share data via Spark's native cache. Furthermore, ExoFlow runs as fast as native Spark alone, while facilitating composition with other systems as well.

Throughput and Scalability. We measure maximum throughput with varying numbers of controllers, (AWS m5.2xlarge) nodes, and tasks per DAG. We use Ray as the optimal baseline, as Ray is also the execution backend.

Figure 8c (1 task/DAG) shows that ExoFlow and Ray both reach saturation after 4 controllers on one node. With 4 nodes, scalability continues, and the gap between ExoFlow and Ray narrows at around 16 controllers. Figure 8d (100 parallel tasks/DAG) shows that throughput overall improves via task batching. Again, with four nodes, both ExoFlow and Ray scale linearly with the number of controllers. ExoFlow achieved roughly 50% of Ray's throughput, due to additional overhead from workflow orchestration and ensuring exactly-once semantics.

7 Related Work

Workflow systems. Industry workflow systems [3, 5, 7, 14] orchestrate execution and recovery for distributed applications by durably logging the workflow, checkpointing task outputs and replaying failed tasks. However, they require external outputs to be idempotent and significantly limit how tasks can pass data to each other (Section 2).

Many workflow systems for FaaS focus on stateful serverless workflows. Several provide a fault-tolerant transactional key-value store interface [45, 46, 49]. ExoFlow is agnostic to external state APIs and implementation and factors out execution and recovery orchestration from such systems.

Some stateful workflow systems offer a fault-tolerant actor programming model [8, 15, 16]. A common recovery technique is *event sourcing*, i.e. durably logging nondeterministic events. However, this requires the developer to use special APIs for nondeterministic code and can add higher overheads than necessary when deterministic replay is not required for application correctness [23, 38]. ExoFlow also supports pluggable actors but only with coarse-grained logging (i.e. recording the workflow DAG) and checkpoint-based recovery (Section 3.4). This is intentionally minimal, as it enables composition of both log- and checkpoint-based actor implementations.

ExoFlow is similar to DARQ [35]: both use composable atomic steps (tasks) and asynchronous checkpointing. Unlike DARQ, ExoFlow exposes references and annotations to avoid materializing and/or persisting outputs where possible.

Dataflow systems. Many dataflow systems use the DAG model [22, 31, 48]. Several use *lineage reconstruction* for recovery, a form of logging that records the DAG but not the data, to reduce run-time overhead. CIEL [39, 41] also introduces *dynamic tasks*, which we adopt. However, these systems target data processing applications in which all tasks are stateless and deterministic. Ray proposes a unified API for DAGs and actors [37], which we also adopt, but cannot support exactly-once semantics or persistence [47]. Tachyon [34] proposes a method of optimizing checkpoints for lineage-based systems; this could be applied to a future version of ExoFlow.

Other systems such as Naiad [38], Apache Flink [18] and Canary [43] implement both batch and streaming dataflow with message passing and global checkpoints at run time for recovery. This produces lower latency but requires more rollback on failure; it can also add more overhead for applications with frequent external outputs [23]. ExoFlow augments log- and checkpoint-based systems by orchestrating recovery across systems with different internal strategies (Section 6.3).

Falkirk Wheel [27] targets efficient and flexible recovery for batch *and* streaming. It uses logical message timestamps to transparently determine the minimum to roll back on failure. ExoFlow provides practical recovery for black-box functions (tasks) by asking semantics from the developer through references and task annotations.

Actor systems. The actor model is a distributed programming model where processes communicate through asynchronous method calls [30]. Most systems do not guarantee exactly-once semantics [2, 12, 17, 47]. ExoFlow provides a limited exactly-once actor model to support workflows that pass actors between tasks. Meanwhile, the application has full flexibility of existing actor systems within a task.

Message-passing systems. Message-passing systems are a generalization of actors in which processes communicate through message sends and receives. There is a large body of work on recovery for message passing, primarily focusing on logging vs. checkpointing [23]. Our work adapts these techniques to the distributed workflow setting and aims to compose log- and checkpoint-based applications.

8 Discussion

References for framework interoperability. Like other dataflow systems, ExoFlow captures the *logical* data movement in an application. ExoFlow also aims to enable *interoperability* across distributed execution frameworks, unlike abstractions such as RDDs [48] or timely dataflow [38] that are tightly coupled to a specific framework. This motivates some of the differences between `Refs` and `ActorRefs` vs. other dataflow abstractions: they can be used to capture third-party data and context, they are serializable, and they do not impose a particular model of parallelism.

Limitations. Using ExoFlow effectively requires developer effort. ExoFlow offers recovery flexibility but the developer must choose the right tradeoff for their application. For example, the developer must decide how large a workflow task should be, and whether checkpointing the output is desirable. Currently task annotations are also very coarse-grained, which makes the system general-purpose but also makes it more challenging for an application to achieve optimal performance and recovery overheads.

There are a number of future directions towards improving ExoFlow's interfacing with external systems. First, while `Refs` allow the application to efficiently pass data between workflow tasks, reading and writing a `Ref`'s data may still require data movement to or from an external framework. Second, currently ExoFlow does not support transactions, i.e. there is no way to specify that a task should be rolled back if another task fails. In this case, the developer must manually roll back the effects of both tasks, e.g., in a final `commitOrAbort` task. Finally, for cases where tasks read and write external state, capturing more fine-grained semantics could reduce developer burden and improve performance. For example, native support for popular types of external state (e.g., a database) could be added.

9 Conclusion

Many existing distributed systems provide specialized, efficient, and transparent recovery for specific application domains. ExoFlow has an orthogonal and complementary goal. To unify heterogeneous applications, we must provide *general* and *interoperable* recovery methods. The greatest challenge is to gain sufficient application semantics without sacrificing flexibility. ExoFlow presents one approach that strikes a balance between usability (minimal annotations, compile-time safety checks) and functionality (flexible `Refs`, automatic recovery). In doing so, we hope to provide universal recovery that matches a universal API: the workflow DAG.

Acknowledgement

We thank the OSDI reviewers and our shepherd, Steven Hand, for their valuable feedback. We also thank Haoran Zhang and Vincent Liu for their insightful discussions and help with Beldi. This work is in part supported by NSF CISE Expeditions Award CCF1730628 and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Nexla, Samsung SDS, Uber, and VMware.

References

[1] Airflow XComs. https://airflow.apache.org/docs/apache-airflow/stable/concepts/xcoms.html. Accessed: 2022-12-13.

[2] Akka. https://akka.io/.

[3] Apache Airflow. https://airflow.apache.org/.

[4] End-to-end mlops pipeline example on azure. https://github.com/microsoft/MLOps/tree/master/examples/KubeflowPipeline.

[5] Google Cloud Composer. https://cloud.google.com/composer.

[6] gRPC. https://grpc.io.

[7] Kubeflow. https://www.kubeflow.org/.

[8] Temporal. https://temporal.io/.

[9] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[10] Michael Armbrust. SPARK-20928: Continuous Processing Mode for Structured Streaming. https://issues.apache.org/jira/browse/SPARK-20928, 2017.

[11] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

[12] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[13] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[14] Jyothi Prasad Buddha and Reshma Beesetty. Step functions. In *The Definitive Guide to AWS Application Integration*, pages 263–342. Springer, 2019.

[15] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 15(8):1591–1604, 2022.

[16] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021.

[17] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 16. ACM, 2011.

[18] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.

[19] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603*, 2015.

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[21] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 85–98, 2012.

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[23] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[24] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia,

and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, 2019.

[25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[27] Ionel Gog, Michael Isard, and Martín Abadi. Falkirk wheel: Rollback recovery for dataflow systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 373–387, 2021.

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

[29] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[30] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[31] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[32] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.

[33] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[34] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15, 2014.

[35] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. Darq matter binds everything: Performant and composable cloud programming via resilient steps. In *Proceedings of the ACM on Management of Data*, 2023.

[36] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):1–39, 2015.

[37] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[38] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[39] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[40] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.

[41] D.G. Murray. *A Distributed Execution Engine Supporting Data-dependent Control Flow*. University of Cambridge, 2012.

[42] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, 2019.

[43] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Salvatore Sanfilippo. Redis: An open source, in-memory data structure store. https://redis.io/, 2009.

[45] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[46] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[47] Stephanie Wang, Eric Liang, Edward Oakes, Benjamin Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In *NSDI*, pages 671–686, 2021.

[48] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[49] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, 2020.

A Artifact Appendix

Abstract

Our artifact includes a comprehensive guide and the source code of the project that allows the evaluators to validate the claims made in ExoFlow. Our artifact runs on Amazon AWS without additional requirements or dependencies. Deploying the code, performing the measurements, generating the plots, and running the benchmarks depend on some third-party frameworks including Anaconda, awscli, and Ray. Please refer to our Github repository https://github.com/suquark/ExoFlow for the latest instructions on reproducing the results.

Scope

The artifact allows the evaluators to validate the claims made in the ExoFlow paper (mostly in the figures) and provides a means to replicate the experiments described. The artifact can be used to set up the necessary environment, execute the main results, and perform microbenchmarks, thus providing a comprehensive understanding of ExoFlow's capabilities.

Contents

Our artifact includes a comprehensive guide designed to assist the evaluator in setting up and running experiments for the ExoFlow paper. It is organized into three primary sections: Local Setup, Main Results, and Microbenchmarks.

The Local Setup section provides instructions to set up an initial AWS EC2 instance. All subsequent experiments will be conducted within that instance.

The Main Results section contains instructions to reproduce the main experiments (ML training pipelines, Stateful serverless workflows, Online-offline graph processing) presented in our paper. These experiments may take a significant amount of time to run (>30 hours) for evaluation. Therefore, we provide options for both batch running experiments and testing individual data points.

The Microbenchmarks section includes instructions for running microbenchmarks, which take a shorter time to complete.

Hosting

You can obtain our artifacts from GitHub: https://github.com/suquark/ExoFlow. The Github repository may be updated later, but we will maintain clear and accessible instructions about our artifacts in an easily identifiable "README" file.

Requirements

ExoFlow is developed and tested on AWS, and we use some AWS services as the baseline. Thus, an AWS account and quota for certain experiments are required.