Providing Efficient Fault Tolerance in Distributed Systems

By

Siyuan Zhuang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Spring 2024

Providing Efficient Fault Tolerance in Distributed Systems

Abstract

Providing Efficient Fault Tolerance in Distributed Systems

by

Siyuan Zhuang
Doctor of Philosophy in Computer Science

University of California, Berkeley
Professor Ion Stoica, Co-Chair
Professor Dawn Song, Co-Chair

The exponential growth in data and computational demands is transforming the approach to system design, particularly in tackling large-scale problems such as training large language models. This shift necessitates the widespread adoption of distributed systems. Simultaneously, systems and applications are becoming increasingly heterogeneous and sophisticated. In this evolving landscape, a critical challenge arises: supporting a wide range of distributed applications while simultaneously achieving computational efficiency and fault tolerance.

This thesis explores the development of universal distributed systems that provide efficient fault tolerance for modern applications. The key idea is to exploit the semantics of workloads at all layers of distributed systems. At the communication layer, we introduce Hoplite, a distributed object store that dynamically exploits data transfer patterns and employs fine-grained pipelining to gain efficiency. Hoplite also reschedules tasks to mitigate the effects of failures. At the task execution layer, ExoFlow leverages the semantics of tasks and data passing between tasks to separate execution and recovery units within workflow systems. This approach ensures exactly-once failure recovery semantics while minimizing checkpointing overhead. Together, these contributions demonstrate a full-stack approach to building universal, efficient, and fault-tolerant distributed systems.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

Behind this dissertation is the effort of many people, and I am very grateful to everyone for their contributions and support.

First and foremost, I would like to express my deepest gratitude to my co-advisors, Ion Stoica and Dawn Song, for their invaluable guidance and unwavering support throughout my PhD journey. Ion taught me the fundamentals of conducting excellent research. I admire his sharp mind, which allows him to transform complex real-world solutions into simple yet powerful ideas, and his determination to push the boundaries of what he believes to have true value. During challenging times, Ion has been incredibly helpful and supportive. Conversing with Dawn is always an exciting experience, as she possesses the remarkable ability to quickly grasp novel ideas. Throughout my Ph.D., Dawn consistently provided me with inspiring insights, steadfast encouragement, and generously shared her extensive knowledge. Her vast experience enriched our research discussions, making them truly invaluable.

I would like to thank my dissertation committee members, Danyang Zhuo and Matei Zaharia. Danyang, who was previously a postdoc in our lab, helped me with my first system paper, and I am very grateful for his mentoring. Matei is a very kind advisor with flexible thinking. I am grateful to him for guiding me on the future directions of my research.

I would like to thank Zhuohan Li, the co-author of my first paper and the person with whom I have published the most papers. His positive mindset and helpful nature have been invaluable to our work together.

I am also grateful to Stephanie Wang, one of the most experienced PhDs in our lab at the time. Stephanie demonstrated her capability as a good mentor for students, and some of my work would not have been possible without her patient guidance.

I would like to express my gratitude to the LM-SYS team, especially Wei-lin Chiang and Lianmin Zheng, for collaborating on the Vicuna and Chatbot Arena projects. I fondly remember the exciting days when we worked overnight to release Vicuna, one of the first chatbots based on open LLMs, which resulted in the founding of the LM-SYS team.

I also want to thank the vLLM team, particularly Zhuohan Li and Woosuk Kwon, for our collaboration. I respect their efforts in making vLLM one of the most popular and recognized open-source LLM serving frameworks.

My appreciation extends to the Skypilot team, especially Zhanghao Wu, for his tremendous effort in helping me push and test a major update - the new provisioner - to the SkyPilot project.

A special nomination goes to Michael Luo, a delightful cook, roommate, colleague, and friend, whose companionship I have greatly enjoyed. I also want to give a special thanks to Sijun Tan, a friend who shares common philosophical ideas and always makes me feel encouraged and cheered up.

I am especially grateful to Jean Nguyen, our CS Graduate Advisor. Jean has always been responsible and responsive to students' requests. It's been great having you around all these years, and best wishes on your new journey!

The lab would not function effectively without the staff of RISELab & Sky Computing Lab - Boban, Dave, Ivan, Jon, Kailee, and Kattt - who are the people behind the scenes keeping the whole lab running. Thank you for organizing all the events and helping us troubleshoot problems in the lab.

Thank you to all my brilliant collaborators. I have enjoyed our collaborations: Frank Sifei Luan, Joseph Gonzalez, Hao Zhang, Dacheng Li, Ying Sheng, Cody Hao Yu, Yonghao Zhuang, Zi Lin, Eric Xing, Shiyuan Guo, Zongheng Yang, Romil Bhardwaj, Gautam Mittal, Scott Shenker, Tianle Li, Eric Liang, Robert Nishihara, Philipp Moritz, Guanhua Wang, Zhuang Liu, Brandon Hsieh, Trevor Darrell, Zi Lin, Yi Cheng, Kaiyuan Zhang, Ang Chen, Fangyu Wu, Kehan Wang, Alexander Keimer, Alexandre Bayen, and Mitar Milutinovic.

Prior to my studies at Berkeley, I was very fortunate to be an undergraduate student at the University of Science and Technology of China (USTC), where I developed my knowledge and skills in computer science. I would like to thank my undergraduate advisors, Chunsheng Li and Jia Xue, for their kind assistance during my four years of study.

Finally, I want to thank my family. First to Xiaoyi He, my dear wife, for her kindness, support over the years, and all the experiences and adventures we have shared. Then to my parents, for their unconditional support, love, and sacrifice, and for all their efforts in trying to provide me with a better environment for education.

Chapter 1

Introduction

In recent years, the exponential growth in data and computational demands has been reshaping the landscape of system design. This trend is particularly evident in tackling large-scale problems, such as training large language models, which require the widespread adoption of distributed systems. For example, in 2023, it cost approximately 21 yottaFLOPs to train GPT-4 [10], which is based on the Transformers architecture. This is 6 orders of magnitude higher than training the original Transformer dated back to 2017. The fundamental limits in compute and memory scalability of commodity hardware make the use of distributed systems inevitable for horizontal scaling.

At the same time, the increasing heterogeneity of applications pose significant challenges for the design and development of distributed systems. Modern applications span a wide range of domains, each with its unique requirements and characteristics. For example, data analytics applications such as graph processing often require a combination of batch and stream processing. Similarly, machine learning (ML) workloads demand extraction, transformation, and loading (ETL) of data on CPUs, as well as model training on GPUs. This heterogeneity creates complexity in the underlying software systems, as they must adapt to support the diverse needs of different application domains. The differences between CPUs and GPUs, for instance, introduce additional challenges in terms of resource management, communication, and fault tolerance. To cope with this heterogeneity, a myriad of specialized distributed execution frameworks have emerged, each designed to support applications within specific domains. While these frameworks provide tailored solutions for standalone applications that fit into their respective domains, they also introduce several problems. First, the lack of interoperability between these frameworks hinders the development of complex end-to-end applications that span multiple domains. Second, different frameworks could have different recovery strategies for fault tolerance, which could not be compatible when combined into an end-to-end application.

Last but not least, the increasing sophistication of applications further complicates the design and development of distributed systems. In the past, distributed

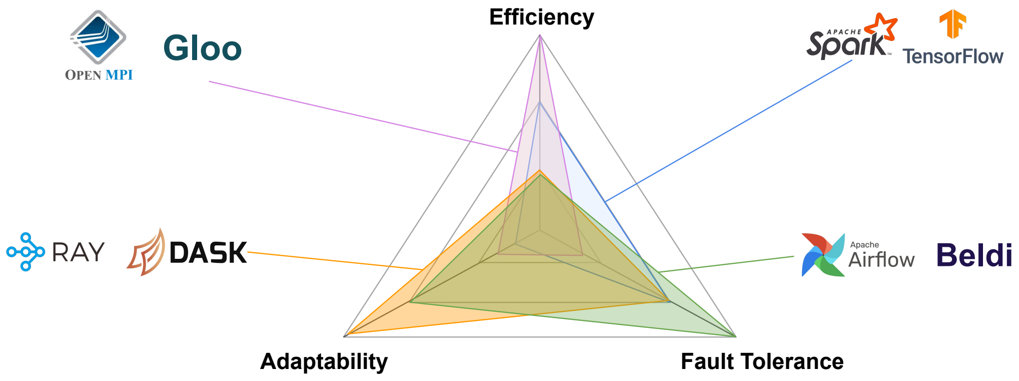


Figure 1.1: A radar chart of existing distributed systems and where they fall short regarding adaptability, efficiency and fault tolerance.

applications often followed the Single Program, Multiple Data (SPMD) model or the Bulk Synchronous Parallel (BSP) model, which had relatively simple and predictable communication patterns. However, emerging applications like distributed reinforcement learning and distributed machine learning model serving require dynamic task scheduling and dynamic communication across multiple nodes in a cluster. The sophistication of these applications introduces new challenges in terms of efficient task scheduling, communication, and fault tolerance. Traditional static collective communication may not be sufficient to handle the dynamic nature of these applications, and fault tolerance mechanisms need to be more flexible and fine-grained to minimize the overhead.

1.1 Overview of Existing Distributed Systems

Before delving into our contributions, we first overview some of our representative distributed systems.

Distributed systems have evolved significantly over the years to cater to the ever-growing demands of data processing, machine learning, and complex application workflows. Among these systems, Apache Spark [103] and TensorFlow [8] have emerged as popular choices for scalable distributed data processing and training. Spark, with its resilient distributed datasets (RDDs) and rich set of APIs, has become a go-to platform for big data analytics and batch processing. TensorFlow, on the other hand, has gained widespread adoption in the machine learning community due to its flexible architecture and extensive ecosystem for building and deploying deep learning models. However, while these systems excel in their respective domains, they fall short in achieving fault tolerance while maintaining efficiency in the new landscape. Spark and TensorFlow are primarily designed as domain-specific distributed systems, focusing on specific applications such as data processing and model training. This specialization limits their adaptability to emerging applications like distributed reinforcement learning, which requires a more dynamic and interactive execution model.

In the realm of distributed communication, Message Passing Interface (MPI) [42] and Gloo [39] have been widely used for efficient communication in high-performance computing and deep learning frameworks. MPI provides a standardized interface for message passing between processes, enabling efficient point-to-point and collective communication. Gloo, developed by Facebook, offers optimized collective communication primitives for machine learning workloads. While these systems excel in static communication patterns, they struggle to support dynamic communication patterns and lack the ability to recover from failures at runtime, which is crucial in modern distributed applications.

Task-based distributed systems, such as Ray [66] and Dask [87], have emerged to address the limitations of domain-specific systems. These systems provide a unified framework for distributed computing, allowing users to define and execute tasks across a cluster of machines. They support dynamic communication patterns and offer runtime fault tolerance mechanisms. However, the trade-off is that they may not be as optimized for efficient communication compared to specialized systems like MPI and Gloo.

Workflow orchestration systems, such as Apache Airflow [3] and Beldi, have gained popularity for managing complex and heterogeneous application pipelines. These systems allow users to define and schedule workflows as directed acyclic graphs (DAGs), enabling the orchestration of diverse tasks and dependencies. They provide strong fault tolerance properties, ensuring the reliable execution of workflows even in the presence of failures. However, this robustness often comes at the cost of efficiency, as the overhead of coordination and checkpointing can impact the overall performance.

To illustrate the strengths and weaknesses of these systems, we present a radar chart (Figure 1.1). The chart visually compares the systems across various dimensions, such as fault tolerance, efficiency, adaptability to emerging applications. It becomes evident that while each system excels in certain aspects, they struggle to achieve a balance among adaptability, fault tolerance and efficiency in the new landscape of distributed computing.

In the following sections, we will introduce our contributions that aim to address these limitations and provide a holistic approach to achieving fault tolerance and efficiency in distributed systems across various application domains.

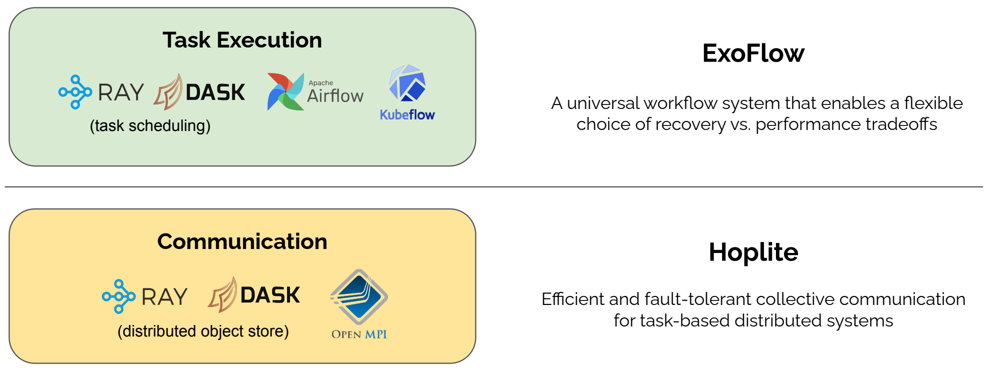

Task Execution
RAY
DASK
(task scheduling)
Apache Airflow
Kubeflow
ExoFlow
A universal workflow system that enables a flexible choice of recovery vs. performance tradeoffs
Communication
RAY
DASK
(distributed object store)
OPEN MPI
Hoplite
Efficient and fault-tolerant collective communication for task-based distributed systems

Figure 1.2: The distributed system stack that is composed of the communication layer and task execution layer.

1.2 Overview and Contributions

In this dissertation, we providing efficient fault tolerance for a wide category of distributed applications, by exploiting the semantics of workloads at all layers of distributed systems. Specifically, we divide distributed system into two layers: the communication layer and task execution layer (Figure 1.2):

1. **Communication Layer**: This layer is responsible for the exchange of data and messages between different nodes in a distributed system. Communication is the foundation of any distributed system, as it enables the coordination and collaboration among distributed components. At this layer, we focus on optimizing data transfer and communication patterns to achieve efficient and fault-tolerant communication.

2. **Task Execution Layer**: This layer includes the actual execution of tasks and computations within a distributed system. It encompasses the scheduling, allocation, and management of resources, as well as the coordination and synchronization of tasks across multiple nodes. At this layer, we explore techniques that exploit the semantics of tasks and their dependencies to achieve efficient and fault-tolerant execution.

At the communication layer, we develop **Hoplite** (Chapter 2). As a distributed object store which is compatible with general task-based distributed systems, Hoplite exploits the dynamic data transfer pattern on-the-fly and transfer data with fine-grained pipelining to gain efficiency and enables fast data transfer reschedule upon task failure to keep making progress. Hoplite speeds up asynchronous stochastic gradient descent, reinforcement learning, and serving an ensemble of machine learning models that are difficult to execute efficiently with traditional collective communication by up to 7.8x, 3.9x, and 3.3x, respectively.

At the task execution layer, we build **ExoFlow** (Chapter 3), a universal workflow system. By exploit the semantics of tasks and data passing between tasks, ExoFlow enables a flexible choice of recovery vs. performance tradeoffs. More specifically, the key insight behind our solution is to decouple execution from recovery and provide exactly-once semantics as a separate layer from execution, with task annotations that specify execution semantics. For generality, workflow tasks can return references that capture arbitrary inter-task communication. ExoFlow generalizes recovery for existing workflow applications ranging from ETL pipelines to stateful serverless workflows, while enabling further optimizations in task communication and recovery.

In Chapter 4, we conclude by discussing future directions for research in this space. We also present the lessons learned and explore alternative approaches for build universal distributed systems that enables efficient fault tolerance. In summary, this dissertation presents novel techniques and frameworks for providing efficient fault tolerance by exploiting the semantics of workloads. The work presented here lays the foundation for further research and development in this area, paving the way for the creation of more efficient, robust, and fault-tolerant distributed systems that can support the ever-evolving landscape of modern applications.

Chapter 2

Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems

We start by investigating the communication layer of distributed systems. Collective communication primitives, such as broadcast and reduce, are widely used in distributed applications to efficiently exchange data among multiple nodes. However, traditional collective communication libraries, such as MPI and Gloo, are designed for static and synchronous workloads and are not well-suited for the dynamic and asynchronous nature of task-based distributed systems.

Task-based distributed systems, such as Ray, Dask, and Hydro, have become increasingly popular for developing and running distributed applications with asynchronous and dynamic computation and communication patterns. These systems rely on a distributed object store to transfer objects between tasks, which allows tasks to be scheduled and executed dynamically based on the availability of resources and data.

In this Chapter, we introduce Hoplite, an efficient and fault-tolerant collective communication layer for task-based distributed systems. Hoplite addresses the challenges of supporting efficient collective communication in dynamic and asynchronous environments by computing data transfer schedules on-the-fly and executing them efficiently through fine-grained pipelining. Hoplite also provides fault tolerance by dynamically adapting the data transfer schedule when failures are detected, allowing live tasks to make progress while failed tasks recover.

We apply Hoplite to a popular task-based framework, Ray, and evaluate its performance on a wide range of existing workloads, including asynchronous stochastic gradient descent (SGD), reinforcement learning (RL), and serving an ensemble of machine learning models. Our evaluations show that Hoplite can significantly speed up these workloads, with improvements of up to 7.8x for asynchronous SGD, 3.9x for RL, and 3.3x for model serving, while requiring only minimal code changes and incurring negligible additional latency in failure recovery.

2.1 Introduction

Task-based distributed systems (e.g., Ray [66], Hydro [48], Dask [87], CIEL [70]) have become increasingly popular for developing and running distributed applications that contain asynchronous and dynamic computation and communication patterns, including asynchronous stochastic gradient descent (SGD), reinforcement learning (RL), and model serving. Today, many top technology companies have started to adopt task-based distributed frameworks for their distributed applications, such as Intel, Microsoft, Ericsson, and JP. Morgan. For example, Ant Financial uses task-based distributed systems to run their online machine learning pipeline and serve financial transactions for billions of users [53].

There are two key benefits of building distributed applications on top of task-based systems. First, it is easy to express asynchronous and dynamic computation and communication patterns. A task-based system implements a *dynamic task* model: a caller can dynamically invoke a task A, which immediately returns an *object future*, i.e. a reference to the eventual return value. By passing the future as an argument, the caller can specify another task B that uses the return value of A even before A finishes. The task-based system is responsible for scheduling *workers* to execute tasks A and B and transferring the result of A to B between the corresponding workers. Second, fault tolerance is provided by the task-based system transparently. When a task fails, the task-based system quickly reconstructs the state of the failed task and resumes execution [99, 94]. Well-behaving tasks do not need to roll back, so failure recovery is low cost.

As a growing number of data-intensive workloads are moving to task-based distributed systems, supporting efficient collective communication (e.g., broadcast, reduce) has become critical. Consider an RL application where the trainer process broadcasts a policy to a set of agents that use this policy to perform a series of simulations. Without the support for collective broadcast, the trainer process needs to send the same policy to every agent which causes a network bottleneck on the sender side.

Efficient collective communication is a well-understood problem in the HPC community and in distributed data-parallel training. Many collective communication libraries exist today, e.g., OpenMPI [42], MPICH [68], Horovod [91], Gloo [39], and NCCL [75]. However, there are two limitations of traditional collective communication implementations that make them an ill fit for dynamic task-based systems.

First, a distributed application using traditional collective communication must specify the communication pattern *before* runtime. This allows the library to compute a *static* and efficient data transfer schedule (e.g., ring-allreduce). For example, for *synchronous* distributed data-parallel training, the application specifies that all workers participate in an allreduce communication, once per training round.

However, in task-based systems, the set of tasks or data objects participating in the collective communication is not known before runtime. One approach would be to wait until all the participating tasks and objects are ready and then compute a static data transfer schedule. Unfortunately, this design misses the opportunity to make partial progress before the entire set of participants are ready, which is critical for the performance of modern *asynchronous* applications, e.g., distributed RL.

Second, because of the synchronous nature of collective communications, one process failure can cause the rest of the processes to hang. Existing solutions leave the recovery up to the application. A typical approach is to checkpoint the state of the application periodically (e.g., every hour), and when a process fails, the entire application rolls back to the previous checkpoint and restarts. Unfortunately, this can be expensive for large-scale asynchronous applications, and does not exploit the ability of tasks that are still alive in the same collective communication group as a failed task to make progress.

This raises an important question: *how can we bring the efficiency of collective communication to dynamic and asynchronous task-based applications?* There are two requirements that are unique to this setting. First, the application must be allowed to specify the participants of a collective communication *dynamically* (i.e., at runtime). Second, the collective communication implementation must be *asynchronous*. This would allow tasks to make progress even if other tasks in the same communication group have failed.

We design and implement Hoplite, an efficient and fault-tolerant collective communication layer for task-based distributed systems. Hoplite combines two key ideas: (1) Hoplite computes data transfer schedule on-the-fly as tasks and objects arrive, and Hoplite executes data transfer schedule efficiently using fine-grained pipelining. Collective communication can make significant progress even if only a fraction of the participants are ready. (2) Hoplite dynamically adapts the data transfer schedule when a failure is detected to alleviate the effects of the failed task in collective communication. This allows the live tasks to make progress. The failed task can rejoin the collective communication after being restarted and complete the communication.

We apply Hoplite to a popular task-based framework, Ray [66]. This allows us to evaluate a wide range of existing workloads on Ray. Our evaluations show that Hoplite can speed up an asynchronous SGD by up to 7.8x, two popular RL algorithms (IMPALA [36], and A3C [65]) on RLlib [62] by up to 1.9x, and 3.9x, respectively, and improve the serving throughput time of an ensemble of ML models on Ray Serve [85] by up to 3.3x, with only minimal code changes and negligible additional latency in failure recovery.

This paper makes the following contributions:

- A distributed scheduling scheme for data transfer that provides efficient broadcast and reduce primitives for dynamic-task systems.
- A fine-grained pipeline scheme that achieves low-latency data transfers between tasks located both on the same node or on different nodes.
- Algorithms to adapt the schedule of the data transfers for broadcast and reduce operations which allows live tasks to make progress when other tasks that participate in the collective communication have failed, and later allow those failed tasks to rejoin.
- We demonstrate the benefits of Hoplite on top of a popular task-based distributed system using several applications, including asynchronous SGD, RL, and serving an ensemble of ML models.

2.2 Background

We first describe task-based distributed systems and their benefits for developing distributed applications. We then describe the challenges of integrating efficient collective communication into them.

2.2.1 Task-Based Distributed Systems

The dynamic task programming model [17, 70, 66, 87, 48] allows applications to express asynchronous and dynamic computation and communication patterns. For instance, Figure 2.1a shows how to implement an *asynchronous* RL algorithm that updates the policy with agent results one at a time, choosing them *dynamically* based on the order of availability. Once a batch of agent results have been applied, the resulting policy is sent to each finished agent to begin the next round of rollout. This allows an agent that has a fast rollout not need to wait for a worker that has a slow rollout (Figure 2.2a). Today, most RL algorithms [36, 65] leverage this type of asynchronous execution for efficient training.

To support this type of asynchronous communication, task-based distributed systems rely on a *distributed object store* to transfer objects between tasks. The object store consists of a set of *nodes*, each of which buffers a (possibly overlapping) set of application objects. Each node serves multiple workers, which can read and write directly to objects in its local node via shared memory. A sender task stores the output into the object store and exits, allowing it to release critical resources (e.g., CPU, GPU, memory) before the receiver tasks are even scheduled. When receiver tasks are ready, they directly fetch the object from the distributed object store. As is standard [70, 66], the object store enforces object immutability and uses a distributed object directory service to map each object to its set of node locations. In addition,

```
def train(policy, num_agents, num_steps, batch_size):
  # Start some rollouts in parallel.
  grad_ids = [rollout.remote(policy)
              for _ in range(num_agents)]
  for _ in range(num_steps):
    for _ in range(batch_size):
      # Wait for the first rollout to finish.
      ready_id = ray.wait(grad_ids)
      # Update the policy with one gradient.
      policy += ray.get(ready_id) / batch_size
      # Remove this gradient from remaining gradients
      grad_ids.remove(ready_id)
    # Once one batch of agents finish, broadcast updated
    # policy to finished agents and start new rollouts.
    for _ in range(batch_size):
      grad_ids.append(rollout.remote(policy))
  return policy
```

(a) Dynamic tasks (Ray).

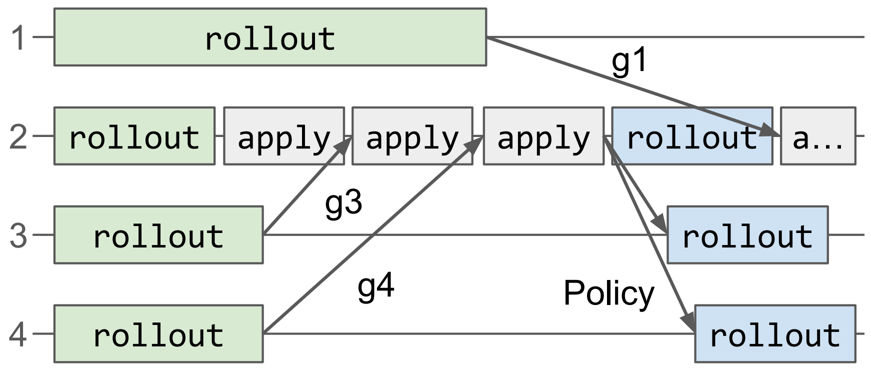
```
 for _ in range(num_steps):
-  for _ in range(batch_size):
-    # Wait for the first rollout to finish.
-    ready_id = ray.wait(grad_ids)
-    # Update the policy with one gradient.
-    policy += ray.get(ready_id) / batch_size
-    # Remove this gradient from remaining gradients
-    grad_ids.remove(ready_id)
+  # Reduce a batch of gradients
+  reduced_grad_id, unreduced_grad_ids = \
+  ray.reduce(grad_ids, num_return=batch_size, op=ray.ADD)
+  # Update the policy with the averaged gradient
+  policy += ray.get(reduced_grad_id) / batch_size
+  # Update remaining gradients
+  grad_ids = ray.get(unreduced_grad_ids)
   # Once one batch of agents finish, broadcast updated
   # policy to finished agents and start new rollouts.
   for _ in range(batch_size):
     grad_ids.append(rollout.remote(policy))
```

(b) Dynamic tasks + collective comm. (Ray + Hoplite).

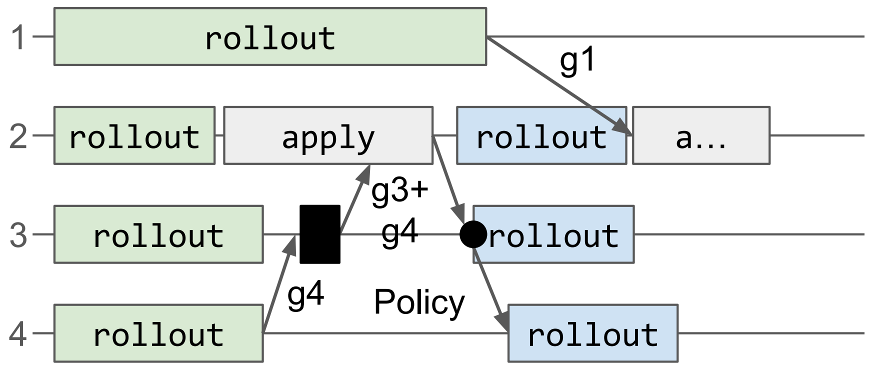
Figure 2.1: Pseudocode for a typical RL algorithm to learn a policy. **(a)** Dynamic tasks with Ray. Each `train` loop waits for a *single* agent to finish, then asynchronously updates the current policy. The new policy is broadcast to a batch of finished agents. **(b)** Modifications to (a) to enable Hoplite. Each step reduces gradients from a subset of agents, updates the current policy, broadcasts the new policy.

task-based distributed systems support fast failure recovery [99, 94] by reconstructing the failed task. Well-behaving tasks do not roll back to keep recovery low cost.

However, if the gradients and the model are large enough in the above RL example, task-based distributed systems incur significant overheads from inefficient communication. For example, the trainer (agent 2) in Figure 2.2a can become a network throughput bottleneck since it has to receive the gradient and also send the new policy from/to each agent individually. This bottleneck becomes more severe when the number of agents increases.



(a) Dynamic tasks (Ray)



(b) Dynamic tasks + collective comm. (Ray + Hoplite).

Figure 2.2: Execution of a distributed RL algorithm. Each row is one agent. Boxes represent computations, and arrows represent data transfers. $g1$-$g4$ are the gradients produced by the agents. **(a)** Dynamic tasks (Ray). Gradients are applied immediately. A batch of three gradients is applied to the current policy before broadcasting. **(b)** Dynamic tasks but with efficient collective communication, in Hoplite. To reduce the network bottleneck at agent 2, agent 3 partially reduces gradients $g3$ and $g4$ (black box), and agent 3 sends the policy to agent 4 (black dot) during the broadcast.

2.2.2 Challenges in Collective Communication

Efficient collective communication has well-known solutions in HPC community and in distributed data-parallel SGD. Many traditional collective communication libraries exist, including Gloo [39], Horovord [91], OpenMPI [42], MPICH [68], and NCCL [75].

They can use efficient data transfer schedule (e.g., ring-allreduce, tree-broadcast) to mitigate communication bottlenecks in distributed applications.

There are two application requirements for using traditional collective communication libraries. First, the communication pattern has to be statically defined before runtime. This is easy for applications that have a bulk-synchronous parallel model. For example, in synchronous data-parallel SGD, all the workers compute on their partitioned set of training data and synchronize the model parameters using allreduce. Second, when any worker fails, all the workers participating in the collective communication hang, and applications are responsible for fault tolerance. For HPC applications, this is typically solved by checkpointing the entire application periodically (e.g., per-hour), and when a process fails, the entire application rolls back to a checkpoint and re-execute.

Unfortunately, these two assumptions are fundamentally incompatible with task-based distributed systems. First, tasks are dynamically invoked by the task-based system's scheduler. This means it is possible that, when collective communication is triggered, only a fraction of the participating tasks are scheduled. For example, on existing task systems, broadcast is implicit: a set of tasks fetch the same object. When only a subset of the receivers are scheduled, it is not possible to build a static broadcast tree without knowing how many total receivers and where and when the receivers will be scheduled. *Therefore, a collective communication layer for a task-based system should adjust data transfer schedule at runtime based on task and object arrivals.*

Second, fast failure recovery is an important design goal for task-based system [99, 94], because many asynchronous workloads have tight SLO requirement (e.g., model serving). In existing task systems, this is done by reconstructing and re-executing failed tasks only. If traditional collective communication libraries are used, a failed task causes the rest of the participating tasks to hang. Thus, *a collective communication layer for task-based systems has to be fault-tolerant: allowing well-behaving tasks to make progress when a task fails and allowing the failed task to rejoin the collective communication after recovery.*

2.3 Design

Hoplite is an efficient and fault-tolerant collective communication layer for task-based distributed systems. At a high level, Hoplite uses two techniques: (1) decentralized fault-tolerant coordination of data transfer for reduce and broadcast, and (2) pipelining of object transfers both across nodes and between tasks and the object store.

We first present a send-receive example workflow using Hoplite's core API (Table 2.1). We then describe Hoplite's object directory service, pipelining mechanism to reduce latency, and fault-tolerant receiver-driven coordination scheme for efficient object transfer in details.

2.3.1 Hoplite's Workflow

```
def application():
  x_id = send.remote()
  recv.remote(x_id)
```

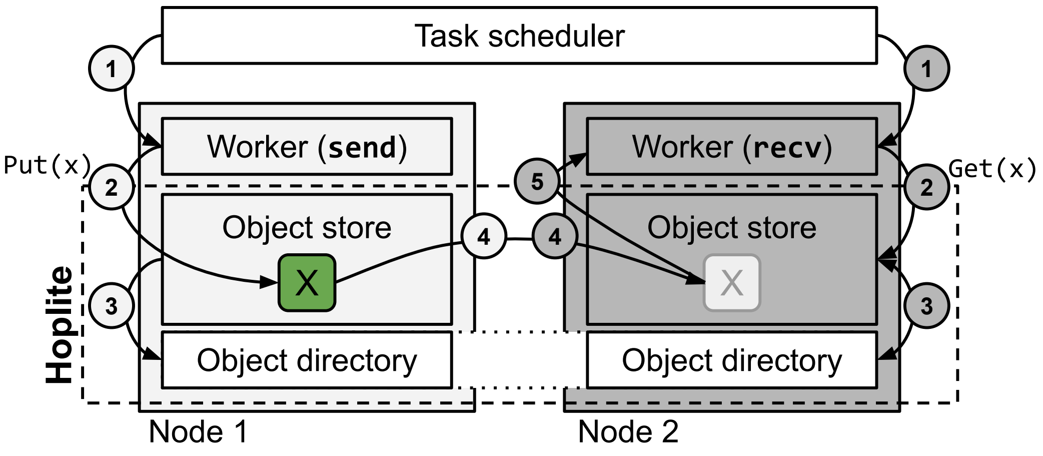


Figure 2.3: Example of a send and receive dynamic task program on a 2-node cluster (N1 and N2). The task-based system consists of a pool of workers per physical node and a scheduler. Hoplite consists of one local object store per node and a global object directory service, which is distributed across physical nodes.

Our example creates a `send` task that returns `x_id` (a future), which is then passed into a `recv` task. In Hoplite, we use an `ObjectID` to represent a future or a reference to an object. During execution, the application first submits the tasks to the task scheduler. The scheduler then chooses a worker to execute each task (step 1, Figure 2.3), e.g., based on resource availability. According to the application, `recv` cannot start executing until it has the value returned by `send`. Note that the task-based system does not require the scheduler to schedule tasks in a particular location or order, i.e. the `recv` task may be scheduled *before* `send`.

In step 2, the task workers call into Hoplite to store and retrieve objects. On node 1, the `send` worker returns an object with the unique ID `x_id`. This object must be stored until the `recv` worker has received it. Thus, the `send` worker calls `Put(x)` on Hoplite, which copies the object from the worker into the local object store (step 2 on N1, Figure 2.3). This frees the worker to execute another task, but incurs an additional memory copy between processes to store objects.

Meanwhile, on node 2, the `recv` worker must retrieve the object returned by `send`. To do this, it calls `Get(x)` on Hoplite, which blocks until the requested object has been copied into the worker's local memory (step 2 on N2, Figure 2.3). In step 3, Hoplite uses the *object directory service* to discover object locations and coordinate

Core Interfaces:	**Description**
Buffer buffer← Get(ObjectID object_id)	Get an object buffer from an object id.
Put(ObjectID object_id, Buffer buffer)	Create an object with a given object id and an object buffer.
Delete(ObjectID object_id)	Delete all copies of an object with a given object id. Called by the task framework once an object is no longer in use.
Reduce(ObjectID target_object_id, int num_objects, {ObjectID source_object_id, ...}, ReduceOp op)	Create a new object with a given object id from a set of objects using a reduce operation (e.g., sum, min, max).

Table 2.1: Core Hoplite APIs. The application generates an `ObjectID` with a unique string and can pass an `ObjectID` by sending the string.

data transfer, in order to fulfill the client's `Put` and `Get` requests. In the example, the Hoplite object store on node 1 publishes the new location for the object `x` to the directory (step 3 on `N1`, Figure 2.3). Meanwhile, on node 2, the Hoplite object store queries the directory for a location for `x` (step 3 on `N2`, Figure 2.3).

Hoplite's object directory service (§2.3.2) is implemented as a sharded hash table that is distributed throughout the cluster (Figure 2.3). Each shard maps an `ObjectID` to the current set of node locations. When there are multiple locations for an object, the directory service can choose a *single* location to return to the client. The object store also maintains information about objects that have only been partially created to facilitate object transfer pipelining (§2.3.3). For example, in Figure 2.3, the object store on node 1 publishes its location to the object directory as soon as `Put(x)` is called, even if the object hasn't been fully copied into the store yet. This allows node 1 to begin sending the object to node 2 while it is still being copied from the `send` worker.

Finally, in step 4, the Hoplite object store nodes execute the data transfer schedule specified by the object directory's reply to node 2. Node 1 is the only location for `x`, so node 2 requests and receives a copy from node 1 (step 4). Node 2 then copies the object from its local store to the `recv` worker (step 5 in Figure 2.3), which again can be pipelined with the copy over the network.

Hoplite provides two efficient collective communication schemes. Hoplite implements efficient broadcast through coordination between the object directory service and the workers (§2.3.4), without an explicit primitive. For reduce, Hoplite exposes an explicit `Reduce` call to the task-based system. It is necessary because this lets Hoplite know that these objects are indeed reducible (i.e., the operation is commutative and associative). Because an `ObjectID` is a future that the object value may not be ready yet, the `Reduce` call also has a `num_objects` input in case the user wants to reduce a subset of the objects, giving Hoplite the flexibility to choose which `num_objects` objects to reduce given their arrival time in the future. Figure 2.1b shows how to modify the RL example to use Hoplite. This allows the trainer to aggregate gradients from a dynamic set of agents efficiently (Figure 2.2b).

Whenever a task fails, Hoplite recomputes a data transfer schedule to avoid using the failed task in the collective communication, and all the rest of the tasks can keep making progress (§2.3.5). Hoplite does not change how task-based distributed system tolerate failures. The underlying task-based distributed system can quickly reconstruct the state of the failed task using their built-in mechanism [99]. Once the state of the task is reconstructed, the task resumes.

2.3.2 Object Directory Service

The object directory service maintains two fields for each object: (1) the size of the object, and (2) the location information. The location information is a list of node IP addresses and the current progress of the object on that node. We use a single bit to represent the object's progress: either the node contains a partial or a complete object. We store both so that partial object copies can immediately act as senders, for both broadcast and reduce (§2.3.4).

Hoplite's directory service supports both synchronous and asynchronous location queries. Synchronous location queries block until corresponding objects are created and locations are known. Asynchronous location queries return immediately, and the object directory service publishes any future locations of the object to the client.

A node writes object locations to the object directory service in two conditions: when a local client creates an object via `Put` and when an object is copied from a remote node. In each case, the node notifies the object directory service twice: once when an object is about to be created in the local store and once when the complete object is ready. We differentiate between partial and complete objects so that object store nodes with complete copies can be favored during a broadcast or reduce (§2.3.4).

Optimization for small objects. Querying object location can introduce an excessive latency penalty for fetching small objects, and the overhead of computing efficient object transfer schedule is usually not worthwhile for small objects in our use cases. Therefore, we implement a fast path in the object directory service. For small objects (¡64KB), we simply cache them in the object directory service, and when a node queries for their location, the object directory service directly returns the object buffers. Similar to object in the per-node stores, cached objects must be freed by the application via the `Delete` call when no longer in use.

2.3.3 Pipelining

Hoplite uses pipelining to achieve low-latency transfer between processes and across nodes for large objects. This is implemented by enabling a receiver node to fetch an object that is incomplete in a source node. An object can be incomplete if the operation that created the object, either a `Put` from the client or a copy between object store nodes, is still in progress. To enable fetching incomplete objects, as shown in the previous section (§2.3.2), the object directory service also maintains locations of incomplete copies. Then, when an object store receives a `Get` operation, it can choose to request the object from a store with an incomplete copy.

By pipelining data transfers across nodes using the object directory service as an intermediary, it becomes simple to also pipeline higher-level collective communication primitives, such as a reduce followed by a broadcast (Figure 2.2b). Within the reduce, a node can compute a reduce of a subset of the input objects and simultaneously send the intermediate result to a downstream node. The downstream node can then compute the final reduce result by computing on the intermediate result as it is received and simultaneously send the final result to any broadcast receivers that have been scheduled. A broadcast receiver can then also simultaneously send the final result to any other broadcast receivers.

Piplining between the task worker and local store on the same node is also important to hide `Put` and `Get` latency for large objects (steps 2 and 5 in Figure 2.3). The reason is that using the distributed object store requires two additional data copies other than the minimum needed to transfer data over the network. The sender task worker must copy to its local store, and then the receiving local store must also copy to its local worker. Our observation is that the additional memory copy latency can be masked by the network transfer if the memory copy is asynchronous. When a sender task calls `Put`, Hoplite immediately notifies the object directory service that the object is ready to transfer. A receiver can then fetch the object before the entire object is copied into the sender node's local store. The receiver side's pipelining mechanism is similar. When the receiver task calls `Get`, the receiver task starts to copy the object from the local store before the local store has a complete object.

By combining cross-node and in-node pipelining, Hoplite enables end-to-end object streaming between the sender and receiver tasks, even when there are multiple rounds of collective communication in between.

Optimization for immutable get. Although Hoplite objects are immutable, the receiver task still copies the object data from its local store during a `Get`, in case it modifies the buffer later on. However, if it only needs read access to the object, then Hoplite can directly return a pointer inside the local store. Read-only access can be enforced through the front-end programming language, e.g., with `const` in C++.

2.3.4 Receiver-Driven Collective Communication

Hoplite's receiver-driven coordination scheme optimizes data transfer using distributed protocols. In Hoplite, data transfer happens in two scenarios: either a task calls `Get` to retrieve an object with a given `ObjectID`, or a task calls `Reduce` to create a new object by reducing a set of other objects with a reduce operation (e.g., sum, min, max).

2.3.4.1 Broadcast

Broadcast in a task-based distributed system happens when a group of tasks located on multiple nodes want to get the same object from its creator task. Specifically, a *sender* task from node `S` creates an object with `Put` and a group of *receiver* tasks `R1`, `R2`, ... fetches it using `Get`. For the receiver tasks that locate on different nodes from the sender task, their corresponding receiver nodes will fetch the object from sender node's local object store to the receiver nodes' local object store. To simplify the description of our method, we assume that the sender task and the receiver tasks locate on different nodes and use the sender `S` and the receiver `R1`, `R2`, ... to also refer to the local object store on the nodes.

Broadcast in a task-based distributed system is challenging because we have no knowledge of the tasks, including where these tasks are located and when these tasks fetch the object. If all receivers simply fetch the object from the sender, the performance will be restricted by the sender's upstream bandwidth. Traditional collective communication libraries can generate a static tree where the root is the sender node to mitigate the throughput bottleneck. The goal of Hoplite's receiver-driven coordination scheme is to achieve a similar effect but using decentralized protocols. Inspired by application-level broadcast [25, 26] in peer-to-peer systems that uses high-capacity nodes to serve as intermediate nodes in the broadcast tree, we use receivers who receives the object earlier than the rest as intermediates to construct a broadcast tree.

When a receiver `R` wants to fetch a remote object, it first checks if the object is locally available, or there is an on-going request for the object locally. If so, the receiver just waits until it gets the completed object. This avoids creating cyclic object dependencies. Otherwise, `R` queries the object directory service for the object's location. The object directory service first tries to return *one* location with a complete copy. If none exist, then the object directory service returns one of the locations holding a partial copy. This is so that partial objects can also act as intermediate senders, but locations with complete copies are favored.

When the location query replies, `R` also removes the location returned from the directory and immediately add itself to the object directory as a location with a partial copy to enable pipelining. Once the data transfer is complete, the receiver adds the sender's location back to the object directory service and mark itself as a location with a complete copy. This makes sure that, for each object, a node can only send to one receiver at a time, thus mitigating bottlenecks at any single node.

Figure 2.4 shows an example of a broadcast scenario in Hoplite. In Figure 2.4a, the first receiver `R1` starts to fetch the object from the sender `S`. In Figure 2.4b, `S` is still sending to `R1`, so it does not appear in the object directory when the second receiver

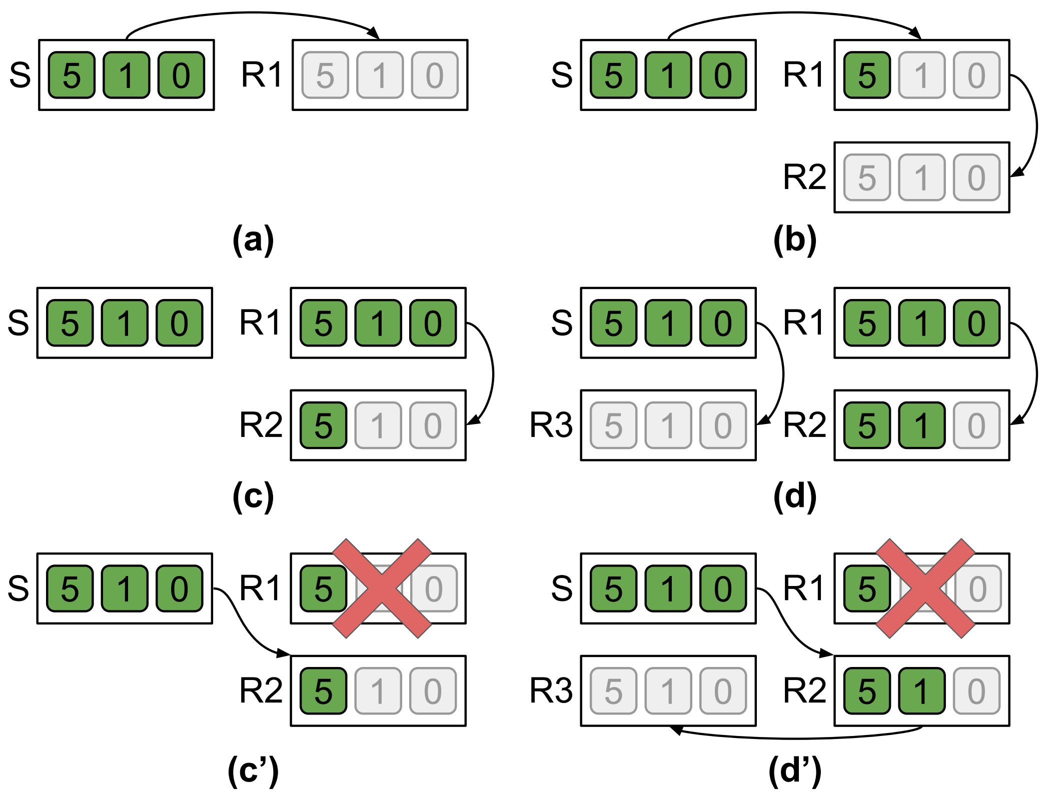


Figure 2.4: An example of broadcasting an object (integer array {5, 1, 0}) from a sender (`S`) in Hoplite, when the receivers (`R1`-`R3`) arrive at different times. **(a)** - **(d)** show the broadcast process without failure. **(c')** and **(d')** show the broadcast process when `R1` fails after **(b)**.

R2 arrives. Thus, R2 fetches the object from R1, the partial copy. In Figure 2.4c, R1 has finished receiving, but is still sending to R2. Then, the object directory contains S and R2 as a complete and partial location, respectively. In Figure 2.4d, R3 queries the object directory, which chooses S over R2 as the sender because S has a complete object.

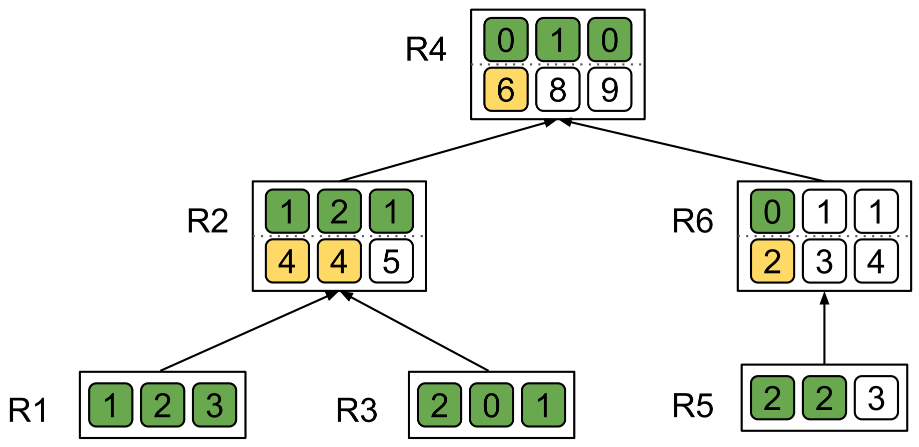
2.3.4.2 Reduce

Reduce happens when a task in a task-based distributed system wants to get a reduced object (e.g., summed or maximal object) from a list of objects. In Hoplite, this happens via a Reduce call. Similar to broadcast, we assume that each object to reduce is located on a separate node and we use R1, R2, ... to represent both the object and the local object store on a node that stores the corresponding object. Note that in a task-based distributed system, the objects to reduce can become ready to reduce in any arbitrary order.

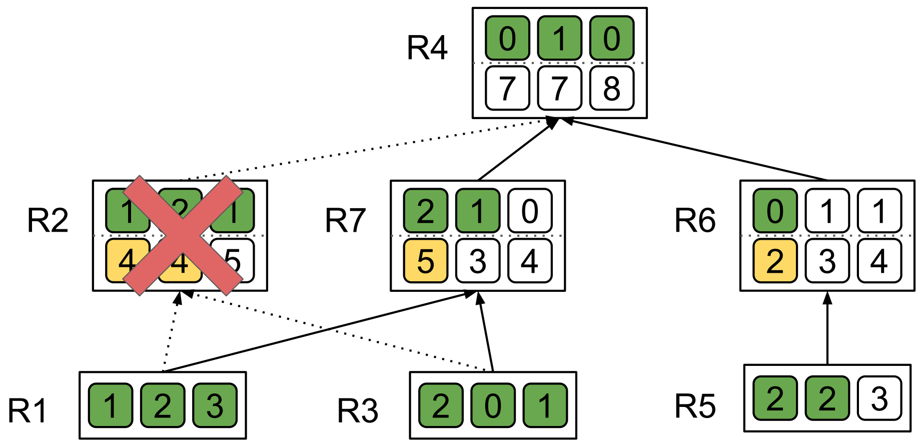
How to reduce objects efficiently to accommodate dynamic object creation is more challenging than broadcast. Broadcast is simpler because a receiver can fetch the object from any sender, and Hoplite thus has more flexibility to adapting data transfer schedule. For reduce, we need to make sure all the objects are reduced once and only once: when one object is added into a partial reduce result, the object should not be added into any other partial results.

In Hoplite, we choose to use a tree-structured reduce algorithm, while the question is what type of tree to use. Let's think about reducing n objects. Without the support of collective communication in task-based distributed systems, each node sends the object to a single receiver. Let's assume that the network latency is L , network bandwidth is B, and the object size is S. This approach's total reduce running time is $L + \frac{nS}{B}$. The L term is due to the network latency, and $\frac{nS}{B}$ is due to the receiver's bandwidth constraint, because every node has to send the object in to it. This is a special kind of tree where the degree of the root is n. When object size is very small (i.e., $\frac{S}{B}$ is negligible), the performance of this kind of tree is the best.

To mitigate the bandwidth bottleneck at the receiver, we can generalize this n-nary tree to a d-nary tree. When we use a d-nary tree, the total running time is $L \log_d n + \frac{dS}{B}$. It reduces the latency due to the bandwidth constraint but incurs additional latency because the height of the tree grows to $\log_d n$. If an object is very large (i.e., $\frac{S}{B} \gg L$), we can set $d = 1$. This means all the nodes are in a single chain, and its running time is $nL + \frac{S}{B}$. Note that we only need to incur $\frac{S}{B}$ for transferring the actual content of the object, because we use fine-grained pipelining, i.e., intermediate nodes send the partially reduced object to the next node. As we can see here, the optimal choice of d depends on the network characteristics, the size of the object, and the number of participants (objects). In other words, we choose the d to minimize



(a) Reduce Tree.



(b) Reduce Tree with failure.

Figure 2.5: Examples of reduce where the objects arrive in the order of `R1`, `R2`, ..., `R6`. The numbers on the top of each node (and the numbers in leaf nodes) represent the object to reduce and green blocks means the fraction of the object that is ready. The numbers on the bottom of each node represent the reduced result and yellow blocks means the fraction of the object that has been reduced. Each intermediate node is responsible to reduce the subtree rooted at it. **(a)** An example reduce tree consists of 6 objects. **(b)** The reconstructed reduce tree after `R2` fails.

the total latency:

$$T(d) = \begin{cases} nL + \frac{S}{B} & \text{if } d = 1; \\ L \log_d n + \frac{dS}{B} & \text{otherwise.} \end{cases} \quad (2.1)$$

During runtime, Hoplite will automatically chooses the optimal d based on an empirical measure of these three factors.

Once the topology of the tree is determined, we need to assign nodes into the tree. Here we want to allow `Reduce` to make significant progress even with a subset of objects. To do so, we assign arriving objects with a generalized version of in-order tree traversal. For a d-nary tree, for each node, we traverse the first child, the node itself, the second child, third child, ..., and the d-th child. Figure 2.5a shows an example for reducing 6 objects with a binary tree. Note that though MPI also supports tree-reduce, our method is completely different: MPI's tree is constructed statically, and our tree is constructed dynamically taking the object arrival sequence into consideration.

If a task only wants to reduce a subset of objects (i.e., `num_object` is smaller than the size of the source object list in `Reduce`), the tree construction process stops when there are `num_object` objects in the tree. For example, if the task wants to reduce 6 out of 10 objects, then the earliest arriving 6 objects are in the reduce tree structured as Figure 2.5a.

An application can also specify the inputs of a `Reduce` incrementally, i.e. by passing the `ObjectID` result of one `Reduce` operation as an input of a subsequent `Reduce` operation. The data transfer for composed `Reduce` operations will naturally compose together. In particular, as soon as the first `Reduce` output is partially ready, it will be added to the object directory service, where it will be discovered by the downstream `Reduce` coordinator. The first output can then be streamed into the downstream `Reduce`.

2.3.4.3 AllReduce

AllReduce is a synchronous collective communication operation that is useful for synchronous data-parallel training. Optimizing allreduce is not our design goal: people usually do synchronous data-parallel training on specialized distributed systems that are optimized for bulk-synchronous workloads (e.g., TensorFlow [8], PyTorch [78]) rather than on task-based distributed systems. In Hoplite, a developer can express allreduce by concatenating reduce and broadcast.

2.3.5 Fault-Tolerant Collective Communication

In the previous subsection, we assume that there is no task failures. However, task failures can happen in a task-based distributed systems for various reasons, including (1) the node that the task is running on crashes, (2) the node runs out of available memory and has to kill the task, and (3) the task encounters a runtime error. Task-based distributed systems already support transparent fault-tolerance to tasks [99], but adding collective communication support requires us to dynamically change data transfer schedule when a fraction of the tasks fail when participating in the collective communication. This is because we do not want a failed task to block collective communication, and we want to allow a recovered task to rejoin an existing collective communication.

2.3.5.1 Broadcast

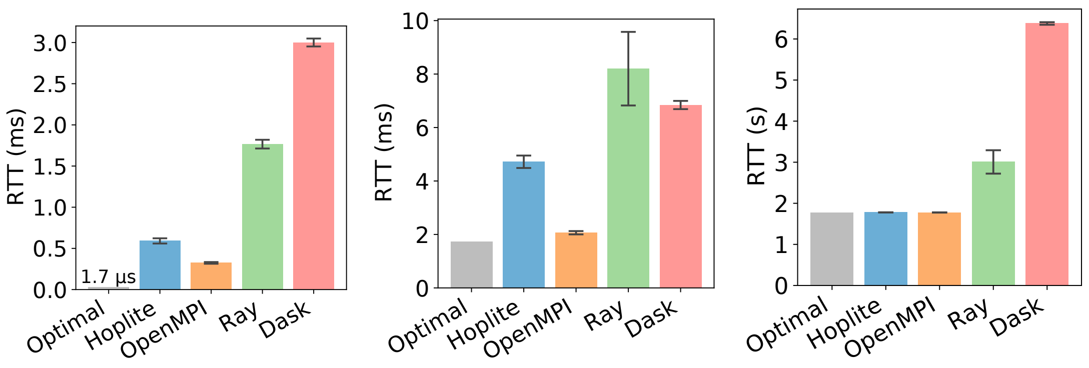
When a sender failure is detected by the receiver in broadcast, the receiver immediately locate another sender by querying the object directory again. The new sender only needs to send the remaining object that the receiver does not have. A failed task can rejoin broadcast transparently because the failed task can simply call `Get` on the same `ObjectID` to fetch the object. Implementing this feature naively would cause cyclic object transfer dependencies. For example, it is possible that two nodes try to fetch the same object from each other. It is because when the a receiver locates an alternative sender, the object directory can return the address of another node which fetches the object from the receiver. To avoid cyclic dependencies, we need to track the dependencies of `Get` if the sender is not the original task that creates the object. If a sender fails, the receiver only resumes if it can find another sender whose dependencies do not include the receiver itself. Figure 2.4c' shows the previous example if `R1` fails. `R2` resumes the fetch from `S`, and when `R3` comes, `R3` can fetch from `R2` (Figure 2.4d').

2.3.5.2 Reduce

When a task fails during `Reduce`, this node is immediately removed from the tree by the coordinator, and will be replaced by the next ready source object. The guarantee is that to reduce n objects from m source objects, as long as at least n objects can be created (i.e., $m-n$ tasks can fail), `Reduce` will return successfully. Otherwise, `Reduce` completes when enough failed tasks are reconstructed by the underlying task-based system's recovery mechanisms. A failed tree node causes its parent, its grandparent, and all its ancestors to clear the reduced object. In the previous example, Figure 2.5b shows the adapted tree after `R2` fails. If the task `Reduce` 6 out of 10 objects and `R2` is recovered after `R7` arrives, `R7` replaces `R2`'s position in the tree. (`R7` can also be the rejoined `R2`.) `R4` has to clear all the current reduced the object, because the final result should be the `Reduce` result of `R1`, `R3`, `R4`, ..., `R7`. Any intermediate result that contains `R2`'s object has to be cleared. Overall, at most $\log_d n$ nodes have to clear the current object.

2.4 Evaluation

We first microbenchmark Hoplite on a set of popular traditional network primitives (e.g., broadcast, reduce, allreduce). We then evaluate Hoplite using real applications on Ray [66], including asynchronous SGD, reinforcement learning, and serving an ensemble of ML models. We also test Hoplite with synchronous data-parallel training workloads to estimate how much performance is lost if people choose to run these static and synchronous workloads on task-based distributed systems. Each application requires less than 100 lines of code changes, most of which are for object serialization. All experiments are done on AWS EC2. We use a cluster of 16 m5.4xlarge nodes (16 vCPUs, 64GB memory, 10 Gbps network) with Linux (version 4.15). We run every test 10 times, and we show standard deviations as error bars.



(a) 1 KB (b) 1 MB (c) 1 GB

Figure 2.6: Round trip latency for point-to-point data communication on Hoplite, OpenMPI, Ray, and Dask. We also include the theoretical optimal RTT (i.e. total bytes transferred divided by the bandwidth).

2.4.1 Microbenchmarks

We use two popular task-based distributed systems, Ray [66] (version 0.8.6) and Dask [87] (version 2.25), as our baselines. In addition, we compare Hoplite with OpenMPI [42] (version 3.3) and Gloo [39]. We chose OpenMPI because OpenMPI is the collective communication library recommended by AWS. We did not choose Horovod because Horovod has three backends: OpenMPI, Gloo, and NCCL. We have already tested OpenMPI and Gloo individually. We currently do not support GPU, so we do not test NCCL.

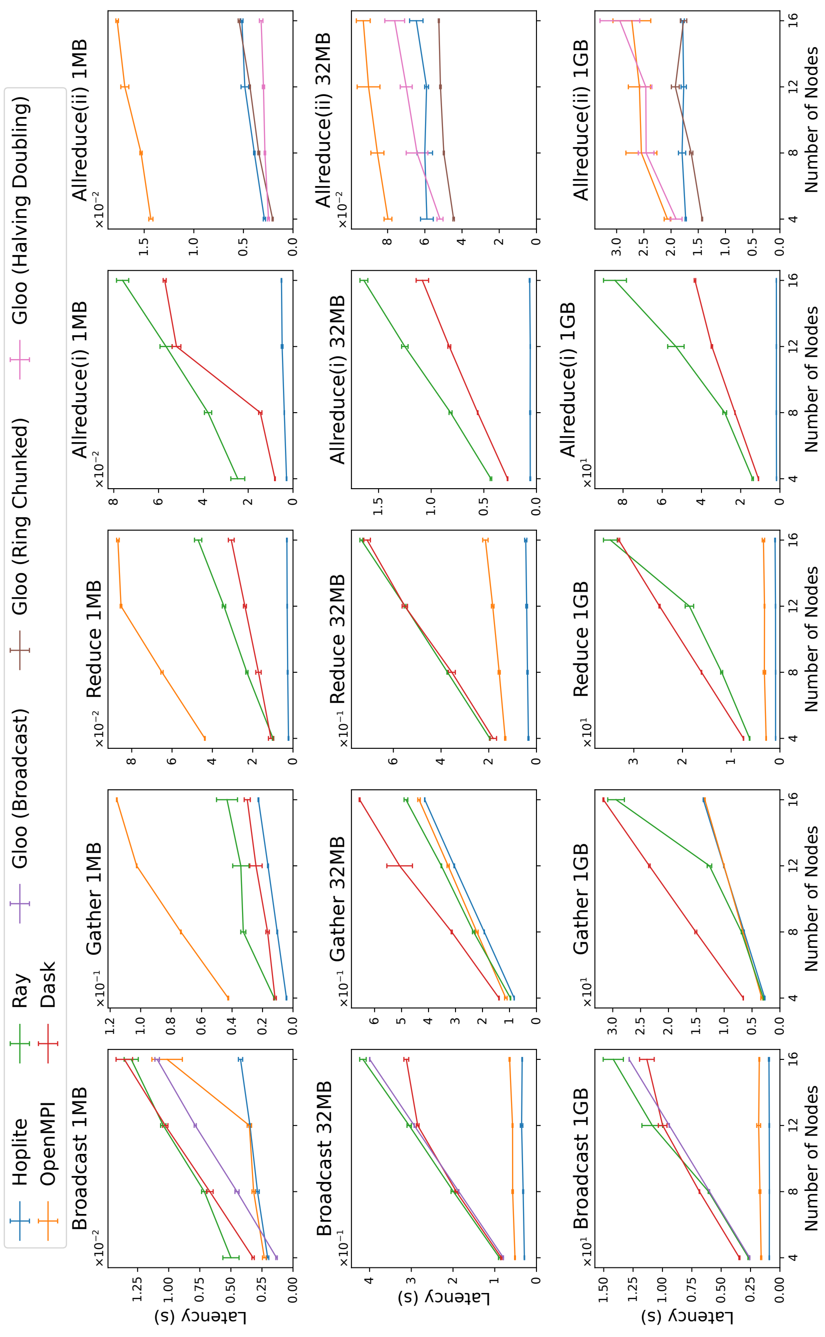


Figure 2.7: Latency comparison of Hoplite, OpenMPI, Ray, Dask, and Gloo on standard collective communication primitives (e.g., broadcast, gather, reduce, allreduce). To show the results more clearly, we split the results of Allreduce into two groups: group (i) includes Hoplite, Ray, and Dask, and group (ii) includes Hoplite, OpenMPI, and two different allreduce algorithms in Gloo.

2.4.1.1 Point-to-Point Data Communication

We first benchmark direct point-to-point transfer. On our testbed, writing object locations to the object directory service takes 167 µs (standard deviation = 12 µs), and getting object location from the object directory service takes 177 µs (standard deviation = 14 µs).

Hoplite's point-to-point communication is efficient. We test round-trip time for different object sizes using OpenMPI, Ray, Dask, and Hoplite. Figure 2.6 shows the result. We also include the optimal RTT, which is calculated by $\text{object\_size}/\text{bandwidth} \times 2$.

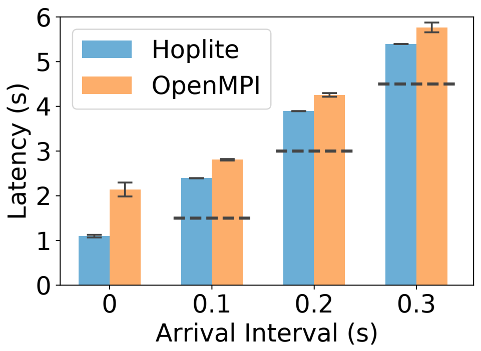
For 1 KB and 1 MB object, OpenMPI is 1.8x and 2.3x faster than Hoplite. For 1 GB objects, Hoplite is 0.2% slower than OpenMPI. Ray and Dask are significantly slower. OpenMPI is the fastest because MPI has the knowledge of the locations of the processes to communicate. Ray, Dask, and Hoplite need to locate the object through an object directory service. Hoplite outperforms Ray and Dask because (1) Hoplite stores object contents in object directory service for objects smaller than 64 KB (§2.3.2) and (2) Hoplite uses pipelining (§2.3.3) to reduce end-to-end latency. Ray does not support pipelining, so it suffers from the extra memory copy latency in the object store. Our pipelining block size is 4 MB, and thus larger object (1 GB) has better pipelining benefits. On 1 GB object, Hoplite achieves similar performance as the underlying network bandwidth despite it has additional memory copies. This is because fine-grained pipelining successfully overlaps memory copying and data transfer.

2.4.1.2 Collective Communication

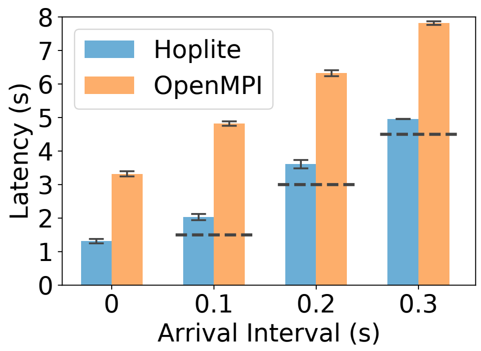
Next, we measure the performance of collective communication on OpenMPI, Ray, Dask, Gloo, and Hoplite, with arrays of 32-bit floats and addition as the reduce operation (if applicable). We measure the time between when the input objects are ready and when the last process finishes. For both Hoplite and Ray, we assume that the application uses a read-only `Get` to avoid the memory copy from the object store to the receiver task (§2.3.3). Gloo only implements broadcast and allreduce. For allreduce, Gloo supports several algorithms. We evaluated the performance for all of them, and for presentation simplicity, we only show the two algorithms with the best performance on our setup: (1) ring-chunked allreduce and (2) halving doubling allreduce.

Figure 2.7 shows the results for medium (1MB) to large (1GB) objects.[1] We present the results for small objects (1KB, 64KB) in §A.2 because small objects are cached in object directory service in Hoplite, and there is thus no collective communi-

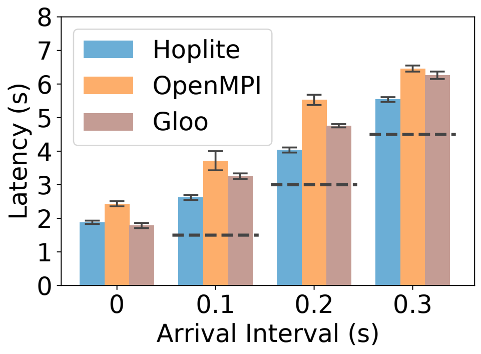
[1]OpenMPI's latency does not increase monotonically because OpenMPI chooses different algorithms on different conditions (e.g., number of nodes, whether the number of nodes is a power of two, object size).



(a) Broadcast



(b) Reduce



(c) AllReduce

Figure 2.8: Latency of 1 GB object broadcast/reduce/allreduce on 16 nodes when tasks start sequentially with a fixed arrival interval. Arrival interval equals to 0 means that all the tasks start at the same time. The dashed lines denote the time the last task arrives.

cation to begin with. In summary, Hoplite achieves a similar level of performance as traditional collective communication libraries, such has OpenMPI and Gloo. Hoplite significantly outperforms Ray and Dask, because Ray and Dask do not support efficient collective communication. Gloo's ring-chunked allreduce is the fastest allreduce implementation for large objects in our tests.

Broadcast. We let one node first `Put` an object, and after the `Put` succeeds, other nodes `Get` the object simultaneously. The latency of broadcast is calculated from the time all nodes call `Get` to the time when the last receiver finishes. Hoplite and OpenMPI achieve the best performance for all object size and node configurations. This is because Ray, Dask, and Gloo do not have collective communication optimization for broadcast. Hoplite slightly outperforms OpenMPI because of fine-grained pipelining.

Gather. We let every node first `Put` an object, and after every node's `Put` succeeds, one of the nodes `Get` all the object via their `ObjectIDs`. The latency of gather is the `Get` duration. OpenMPI and Hoplite outperforms the rest for all object size and node configurations. This is because both Ray and Dask need additional memory copying between workers and the object store. Hoplite also needs additional memory copying, but the latency is masked by fine-grained pipelining between workers and the object store.

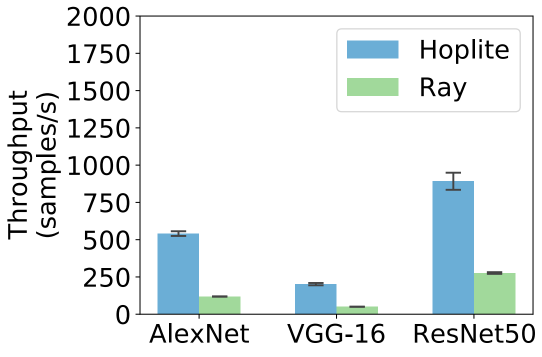
Reduce. We let every node first `Put` an object, and after every node's `Put` succeeds, one of the nodes `Reduce` the objects via their `ObjectIDs` to create a new `ObjectID` for the result. The node then calls `Get` to get the resulting object buffer. The latency of reduce is calculated from the time the node calls `Reduce` to the time the node has a copy of the reduce result. OpenMPI and Hoplite consistently outperform the rest for all object size and node configurations since Ray and Dask do not support collective communication. Hoplite can slightly outperform OpenMPI because of fine-grained pipelining.

AllReduce. In Hoplite, we simply concatenate reduce and broadcast to implement allreduce. The latency of allreduce is calculated from the time a node starts to `Reduce` all the objects to the last node `Get` the reduce result. We divide the results into two groups in Figure 2.7. Hoplite significantly outperforms Ray and Dask because of the collective communication support of broadcast and reduce in Hoplite. Note that efficient allreduce is not our design goal since allreduce is a static and synchronous collective communication operation. However, Hoplite still achieves comparable performance with static collective communication libraries such as OpenMPI and Gloo.

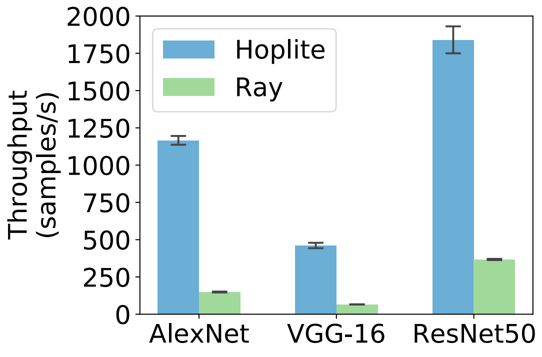
2.4.1.3 Asynchrony

Hoplite's performance is robust even when processes are not synchronized, which is typical in task-based distributed systems. We measure broadcast, reduce, and allreduce latencies when the participating tasks arrive sequentially with a fixed arrival interval. For broadcast (Figure 2.8a), OpenMPI makes some progress before the last receiver arrives (§2.6). However, the algorithm is static (i.e. based on process *rank*

[42]), while Hoplite achieves a lower latency with a dynamic algorithm that does not depend on the particular arrival order. We do not include Gloo because it does not optimize its broadcast performance (Figure 2.7). For reduce (Figure 2.8b) and allreduce (Figure 2.8c), both OpenMPI and Gloo have to wait until all processes are ready, while Hoplite can make significant progress before the last object is ready. This allows Hoplite to even outperform Gloo's ring-chunked allreduce when objects do not arrive at the same time.



(a) Number of Nodes = 8



(b) Number of Nodes = 16

Figure 2.9: Training throughput (number of training samples per second) for asynchronous SGD.

2.4.2 Asynchronous SGD

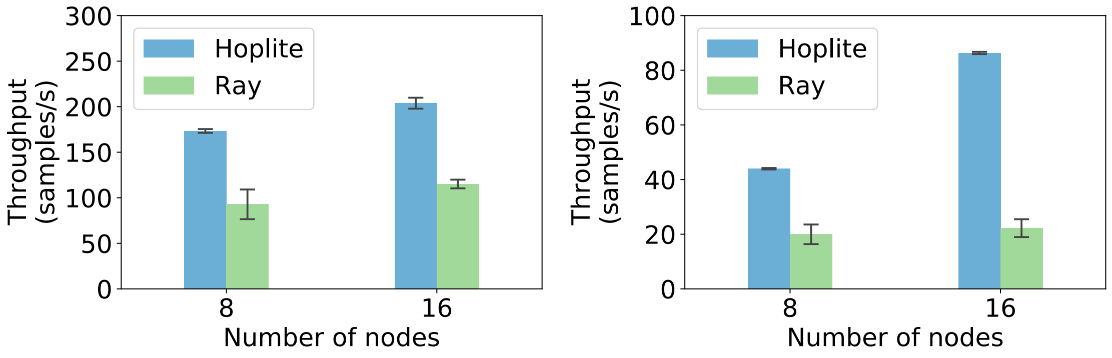
Asynchronous stochastic gradient descent (SGD) is one way to train deep neural networks efficiently, and it usually uses a parameter server framework [60, 59, 32, 60, 59]: clients fetch the parameters from a centralized server, evaluate the parameters on its own portion of data (e.g., performing forward and backward propagation on a neural network), and send the updates (e.g., gradients) back to the server independently. The parameter server needs to broadcast parameters to and reduce from an uncertain set of workers.

Here we evaluate Hoplite with Ray's example implementation of an asynchronous parameter server [84]. We use three widely-used standard deep neural networks, AlexNet [55] (model size = 233 MB), VGG-16 [92] (model size = 528 MB), and ResNet-50 [44] (model size = 97 MB). We test two cluster configurations: 8 p3.2xlarge nodes and 16 p3.2xlarge nodes on AWS. p3.2xlarge instance has the same network performance as m5.4xlarge instance but with an additional NVIDIA V100 GPU to accelerate the execution of the neural networks. The asynchronous parameter server collects and reduces the updates from the first half of worker nodes that finish the update and broadcast the new weights back to these nodes.

We show the results in Figure 2.9. Hoplite improves the training throughput of the asynchronous parameter server. Comparing to Ray, it speedups training on asynchronous parameter server for 16 nodes by 7.8x, 7.0x, and 5.0x, for AlexNet, VGG-16, and ResNet-50, respectively. Ray is slow because the parameter server has to receive gradients from each worker and send the updated model to each worker one by one. This creates a bandwidth bottleneck at the parameter server. In Hoplite, these operations are optimized by our broadcast and reduce algorithms.

2.4.3 Reinforcement Learning

RL algorithms involve the deep nesting of irregular distributed computation patterns, so task-based distributed systems are a perfect fit for these algorithms. We evaluate Hoplite with RLlib [62], a popular and comprehensive RL library on Ray. Distributed RL algorithms can be divided into two classes: In *samples optimization* (e.g., IMPALA [36], Asynchronous PPO [90]), a centralized trainer periodically broadcasts a policy to a set of workers and gather the rollouts generated by the workers to update the model. In *gradients optimization* (e.g., A3C [65]), the workers compute the gradient with their rollouts, and the trainer updates the model with the reduced gradients from the workers.



(a) IMPALA (b) A3C

Figure 2.10: RLlib's training throughput (number of training samples per second) on Ray and Hoplite.

We evaluate two popular RL algorithms, IMPALA [36] and A3C [65], one from each class. We test two cluster configurations: 8 nodes (1 trainer + 7 workers) and 16 nodes (1 trainer + 15 workers). The trainer broadcast a model to the first half workers that have finished a round of simulation (in IMPALA) or gradient computation (in A3C). We use a two-layer feed-forward neural network with 64 MB of parameters. Figure 2.10 shows the training throughput. Training throughput is calculated by the number of simulation traces (in samples optimization) or gradients (in gradients optimization) the RL algorithm can process in a second.

Hoplite significantly improves the training throughput of both IMPALA and A3C. Hoplite improves the training throughput of IMPALA by 1.9x on an 8-node cluster and 1.8x on a 16-node cluster. The reason Hoplite outperforms Ray is because IMPALA has to broadcast a model of 64 MB frequently to the workers. We expect more improvement when the number of nodes is higher, but we already achieve the maximum training throughput: IMPALA is bottlenecked by computation rather than communication using Hoplite with 16 nodes (15 workers). For A3C, Hoplite improves the training throughput by 2.2x on the 8-node configuration and 3.9x on the 16-node configuration. Unlike IMPALA, A3C achieves almost linear scaling with the number of workers. A3C on Ray cannot scale linearly from 8 nodes to 16 nodes because of the communication bottleneck.

2.4.4 ML Model Serving

Machine learning is deployed in a growing number of applications which demand real-time, accurate, and robust predictions under heavy query load [31, 15, 77]. An important use case of task-based distributed system is to serve a wide range of machine learning models implemented with different machine learning frameworks [66].

We evaluate Hoplite with Ray Serve [85], a framework-agnostic distributed machine learning model serving library built on Ray. We set up an image classification service with a majority vote-based ensemble of the following models: AlexNet [55], ResNet34 [44], EfficientNet-B1/-B2 [96], MobileNet V2 [88], ShuffleNet V2 x0.5/x1.0 [63], and SqueezeNet V1.1 [49]. We test two cluster configurations: 8 p3.2xlarge nodes and 16 p3.2xlarge nodes on AWS. For 8 nodes setting, we serve a different model on each node. For 16 nodes setting, each model is served by two different nodes and the two nodes serve the model with two different versions of weight parameters. Each query to the service includes a batch of 64 images of size 256×256. During serving, the service will broadcast the query to all the nodes to evaluate on different models, gather the classification results, and return the majority vote to the user.

We visualize the results in Figure 2.11. Hoplite improves the serving throughput for serving an ensemble of image classification models. Comparing to Ray, it speedups the serving throughput by 2.2x for 8 nodes and 3.3x for 16 nodes. This shows that the optimized broadcast algorithm in Hoplite helps Ray Serve to improve the serving throughput.

2.4.5 Fault Tolerance

We evaluate the failure recovery latency before and after we apply Hoplite to Ray. We rerun our model serving with 8 models and async SGD workloads with 6 workers, and we manually trigger a failure. We do this experiment 10 times. Figure 2.12 shows one particular run. The y-axis shows the latency per query (in model serving) or per iteration (in async SGD), and the x-axis shows the index of the query or the iteration.

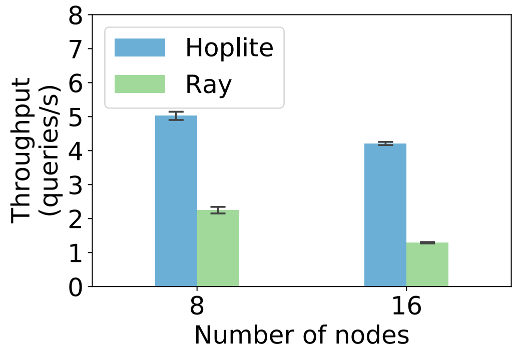
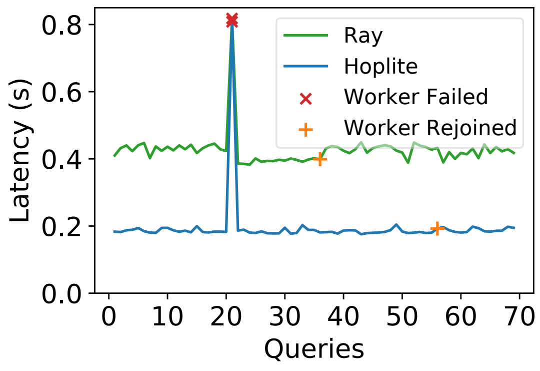
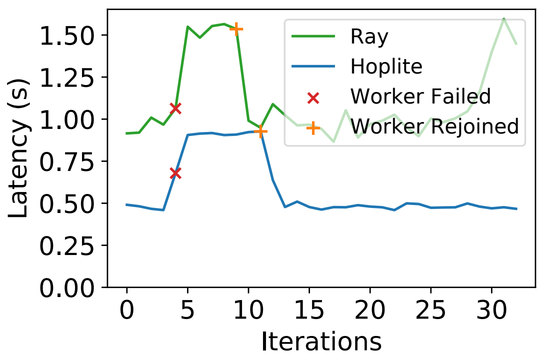


Figure 2.11: Ray Serve's performance (queries per second) on Ray and Hoplite for an ensemble of image classification models.



(a) Ray Server latency.

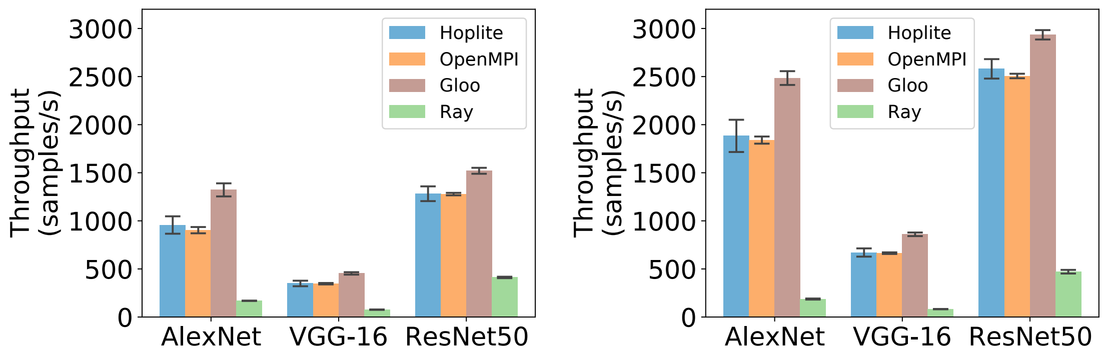


(b) Async SGD latency.

Figure 2.12: Latency when a pariticipating task fails and rejoins on (a) Ray Serve and (b) async SGD.

Hoplite significantly improves Ray's performance. Ray's failure detection latency is 0.58 ± 0.13 second, and after we apply Hoplite to Ray, Ray's failure detection latency increases to 0.74 ± 0.05 second. The additional 28% latency introduced by Hoplite is because Hoplite has a different failure detection mechanism. Ray detects failure by monitoring the liveness of the worker process. Hoplite detects failure by checking the liveness of a socket connection.

After the failure, Ray Serve's latency drops because it only needs to broadcast to less receivers. The latency comes back to normal after the failed worker rejoins. For Hoplite, the latency difference is negligible because of the efficient broadcast algorithm. Hoplite takes more queries between the task fails and the task rejoins. This is because Hoplite is efficient and has processed more queries during the recovery window (the time between the failure and task rejoin). In async SGD, latency for training each iteration increases in the recovery window because of the temporary loss of a worker. The difference in recovery latency (duration of the recovery window) between Ray and Hoplite is negligible because both use Ray's mechanism to reconstruct the failed task.



(a) Number of Nodes = 8 (b) Number of Nodes = 16

Figure 2.13: Training throughput (number of training samples per second) for synchronous data-parallel training.

2.4.6 Synchronous Data-Parallel Training

Synchronous data-parallel training involves a set of workers, each runs on a partition of training data, and the workers synchronize the gradients each round using allreduce [41]. Speeding up synchronous data-parallel training workloads is not our design goal, and they do not require the flexibility provided by task-based systems. Instead, they can run directly on specialized distributed systems that are optimized for static and synchronous workloads (e.g., TensorFlow [8], PyTorch [78]). These systems rely on efficient allreduce implementations in traditional collective communication frameworks (e.g., OpenMPI, Gloo).

However, an interesting question to ask is how much performance developers have to pay if they choose to run these static and synchronous workloads on task-based distributed systems. Our cluster setup is the same as the asynchronous parameter server experiment. In addition to Ray, we evaluate Gloo and OpenMPI. We evaluate the Gloo baseline through PyTorch, which chooses ring-chunked allreduce as its choice for Gloo's algorithm.

We show the results in Figure 2.13. Hoplite significantly improves the synchronous data-parallel training for Ray. Ray is slower than Hoplite, OpenMPI, and Gloo, with the similar reason as in asynchronous parameter server. Hoplite achieves similar speed with OpenMPI. However, Hoplite is 12-24% slower than Gloo. This is expected because ring-allreduce is more bandwidth efficient than the tree-reduce plus broadcast in Hoplite.

2.5 Discussion

Garbage collection. Hoplite provides a `Delete` call (Table 2.1) that deletes all copies of an object from the store. This can be used to garbage-collect an object whose `ObjectID` is no longer in scope in the application. However, it is still the task framework or application's responsibility to determine when `Delete` can and should be called, since only these layers have visibility into which `ObjectIDs` a task has references to. The guarantee that Hoplite provides is simple: when `Put` is called on an `ObjectID`, the object copy that is created will be pinned in its local store until the framework calls `Delete` on the same ID. This guarantees that there will always be at least one available location of the object to copy from, to fulfill future `Get` requests. Meanwhile, Hoplite is free to evict any additional copies that were generated on other nodes during execution, to make room for new objects. The overhead of eviction is very low, since Hoplite uses a local LRU policy per node that considers all unpinned object copies in the local store.

Framework's Fault tolerance. Hoplite ensures that collective communication can tolerate task failure. A task-based distributed system has a set of control processes that can also fail, and they usually require separate mechanisms to tolerate failures. For example, the object directory service can fail and requires replication for durability. These failures are handled by the underlying framework independent of whether Hoplite is used.

Network Heterogeneity. The design of Hoplite assumes that the network capacity between all the nodes is uniform. Accommodating heterogeneous network can achieve higher performance (e.g., using high bandwidth nodes as intermediate nodes for broadcast, fetching objects from a node which has lower latency). This can be done by monitoring network metrics at run time. We do not need this feature for our use cases because our cloud provider ensures uniform network bandwidth between our nodes.

Integration with GPU. Hoplite currently does not support pipelining into GPU memory. If training processes need to use GPU, the application has to copy data between GPU and CPU memory. In the future, we want to extend our pipelining mechanism into GPU memory.

2.6 Related Work

Optimizing data transfer for cluster computing. Cluster computing frameworks, such as Spark [103] and MapReduce [33], have been popular for decades for data processing, and optimizing data transfer for them [29, 30, 28, 80, 56] has been studied extensively. AI applications are particularly relevant because they are communication-intensive, and traditional collective communication techniques are widely-used [98, 91, 39]. Pipelining is also a well-known technique to improve performance [74, 79]. Our work focuses on improving task-based distributed systems [66, 87, 48]. Applications on these frameworks have dynamic and asynchronous traffic patterns. To the best of our knowledge, Hoplite is the first work to provide efficient collective communication support for task-based distributed systems.

Using named objects or object futures for data communication. Using named objects or object futures for data communication is not new. In serverless computing, tasks (or functions) cannot communicate directly. As a result, tasks communicate through external data stores [81], such as Amazon S3 [11] or Redis [86]. There, the storage and compute servers are disaggregated, and computer servers do not directly communicate. We target a standard cluster computing scenario, where data is directly transmitted between compute servers. Object futures are a useful construct for expressing asynchronous computation. Dask, Ray, Hydro, and PyTorch [78] all use futures to represent results of remote tasks. Our work is complementary to them, showing that efficient collective communication can co-exist with named objects or object futures.

Asynchronous MPI. MPI supports two flavors of asynchrony. First, similar to a non-blocking POSIX socket, MPI allows an application to issue asynchronous network primitives and exposes an `MPI_Wait` primitive to fetch the result. Second, depending on the MPI implementation, some collective communication primitives can make some progress with a subset of participants. For example, in `MPI_Bcast`, the sender generates a static broadcast tree. If the receivers arrive in order from the root of the tree to the leaves of the tree, the receivers can make significant progress before the last receiver arrives. If not, then a receiver must wait until all its upstream ancestors are ready before making any progress (evaluated in Figure 2.8). In Hoplite, the broadcast tree is generated dynamically at runtime, so the arrival order does not matter. In addition, asynchronous MPI primitives still require applications to specify all the participants before runtime. In Hoplite, the communication pattern can be expressed dynamically and incrementally, allowing Hoplite to work with existing task-based distributed systems.

Collective communication in other domains. Optimizing data transfer has been studied extensively in other domains. Application-level multicast [25, 26] for streaming video on wide-area networks. IP multicast [50] enables a sender to send simultaneously to multiple IP addresses at the same time. These work mostly focus entirely on multicast rather than general-purpose collective communication in distributed computing frameworks.

2.7 Conclusion

Task-based distributed computing frameworks have become popular for distributed applications that contain dynamic and asynchronous workloads. We cannot directly use traditional collective communication libraries in task-based distributed systems, because (1) they require static communication patterns and (2) they are not fault-tolerant. We design and implement Hoplite, an efficient and fault-tolerant communication layer for task-based distributed systems that achieves efficient collective communication. Hoplite computes data transfer schedules on the fly, and even when tasks fail, Hoplite can allow well-behaving tasks to keep making progress while waiting for the failed tasks to recover. We port a popular distributed computing framework, Ray, on top of Hoplite. Hoplite speeds up asynchronous SGD, RL, model serving workloads by up to 7.8x, 3.9x, and 3.3x, respectively. Hoplite's source code is publicly available (`https://github.com/suquark/hoplite`). This work does not raise any ethical issues.

Chapter 3

Exoflow: A Universal Workflow System for Exactly-Once DAGs

Moving up to the task execution layer of distributed systems, we observe that existing frameworks often impose a fixed recovery strategy, such as synchronous checkpointing of all task outputs, to ensure exactly-once semantics. While this approach guarantees correctness, it can lead to significant performance overheads, especially for data-intensive applications. Moreover, it fails to leverage the diverse recovery requirements and semantics of different tasks within a workflow.

In this chapter, we argue that distributed systems should provide a flexible choice of recovery strategies, enabling applications to optimize the tradeoff between performance and fault tolerance based on their specific requirements. To this end, we introduce ExoFlow, a universal workflow system that decouples execution from recovery and allows applications to specify task semantics through annotations.

ExoFlow employs a novel architecture that is able to scale recovery and execution independently, and to make the storage and execution backends pluggable. This allows ExoFlow to support a wide range of execution environments, from serverless functions to distributed data processing frameworks, while providing consistent and efficient recovery guarantees.

To enable flexible recovery strategies, ExoFlow introduces two key interfaces: references and task annotations. References allow tasks to efficiently pass large and distributed data as internal outputs, without materializing them or involving the workflow system in the physical communication. Task annotations, such as determinism and rollback behavior, enable the workflow system to make informed decisions about checkpointing and recovery, minimizing unnecessary overheads.

We demonstrate the benefits of ExoFlow on a diverse set of applications, including an ML pipeline, serverless transactions, and graph processing that mixes stream and batch execution. Compared to existing workflow systems, ExoFlow achieves significant performance improvements, such as a 5x speedup for Spark data processing workflows and a 51% reduction in latency for transactional serverless workflows, while maintaining exactly-once semantics.

3.1 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 [35]. 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 [102], and Apache Flink uses checkpointing for stream processing [23].

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 [12]. On the other hand, Flink has added a batch processing mode, but this required building an entirely separate recovery system from the streaming path [24].

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 chapter, 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 [67] 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 [104]. 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.

3.2 Motivation

3.2.1 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 [35] or linearizability in storage systems [46].

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 3.1a 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 3.1b). For example, consider the cases if `A` is nondeterministic and we do not checkpoint `a_out` in Figure 3.1a. 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 $\texttt{a\_out}_2$.

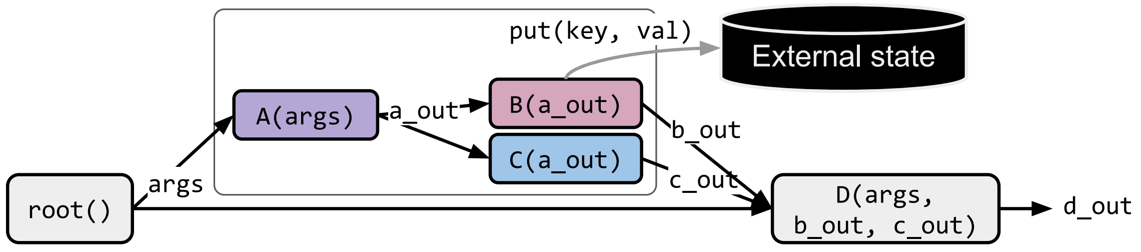
We encounter additional problems in the opposite case where `B` finishes and we then lose $\texttt{a\_out}_1$. `B` has already made `(key,val)` external and these values may depend on $\texttt{a\_out}_1$. If we execute `C` on $\texttt{a\_out}_2$, `c_out` will be inconsistent with `(key,val)`. Thus, the only way to guarantee correctness in this case is to either: (1) "commit" $\texttt{a\_out}_1$ 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 3.1a 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 [35]. 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



(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 3.1: **(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.

approach does not leverage application-specific recovery methods (Figure 3.1b). 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 [37]. Together, these force developers to use external outputs for much of their task communication [37, 82].

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 3.1a 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.

Second, we support *annotations* to specify task semantics (checkpointing, nondeterminism, output visibility). These allow the system to determine recovery cor-

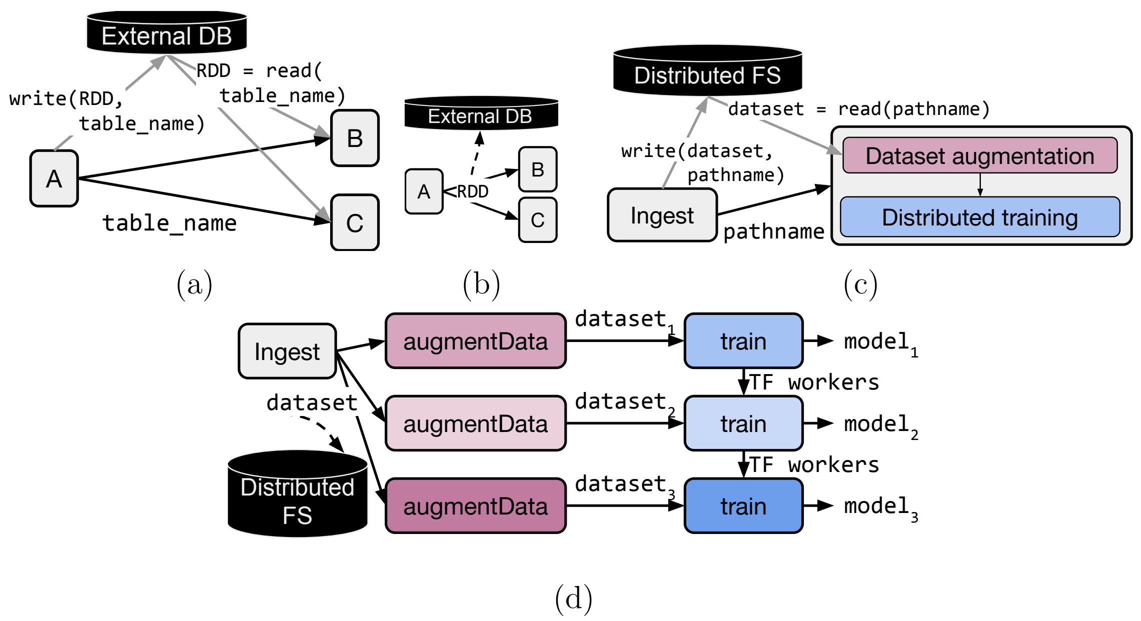


Figure 3.2: **(a)** ETL workflow today, using external outputs for communication. **(b)** The same ETL workflow with internal outputs only. **(c)** ML training workflow today, with external outputs and manual orchestration within a task. **(d)** The same ML workflow with internal outputs only, and orchestration is handled by the workflow system. Third-party framework state (`TF workers`) can be passed between workflow tasks.

rectness 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. The annotations are set to a safe default, i.e. each task's output(s) must be checkpointed, is assumed to be nondeterministic, and any external outputs must be made idempotent. This produces write-ahead logging behavior equivalent to that of a workflow system such as Apache Airflow.

3.2.2 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 3.2a 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 [13]. 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 [102] to `B` and `C` (Figure 3.2b), avoiding the round trip to external storage, allowing `B` and `C` to share 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 3.2b 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 3.2c 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 [72]. 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 3.2d. 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

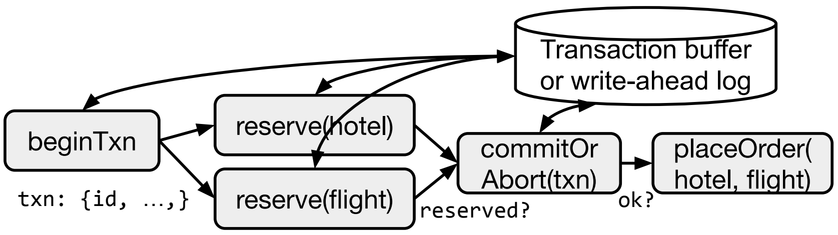


Figure 3.3: Serverless workflow systems [94, 104, 52] guarantee exactly-once semantics by interposing on all communication to external storage, e.g., through a transaction buffer, and explicitly managing visibility of these external effects.

FaaS [45].

Recently, *serverless workflow* systems [20, 94, 104] 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 3.3 shows an example of a trip reservation workflow [38] that places the order if and only if both the hotel and flight were successfully reserved. Systems such as Aft [94], Beldi [104], and Boki [52] guarantee exactly-once semantics by providing a transactional key-value store to manage external output visibility.

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 3.3 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.

3.3 API

3.3.1 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 3.1) 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.

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 3.1: 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.

DAG interface. The application invokes workflow tasks and specifies arguments using `f.bind` (Table 3.1). 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 interface. Within a task, the application can invoke arbitrary local or distributed execution. For greater generality, we also adopt the *dynamic* task model [71]: 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 3.1). 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 3.2, 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.

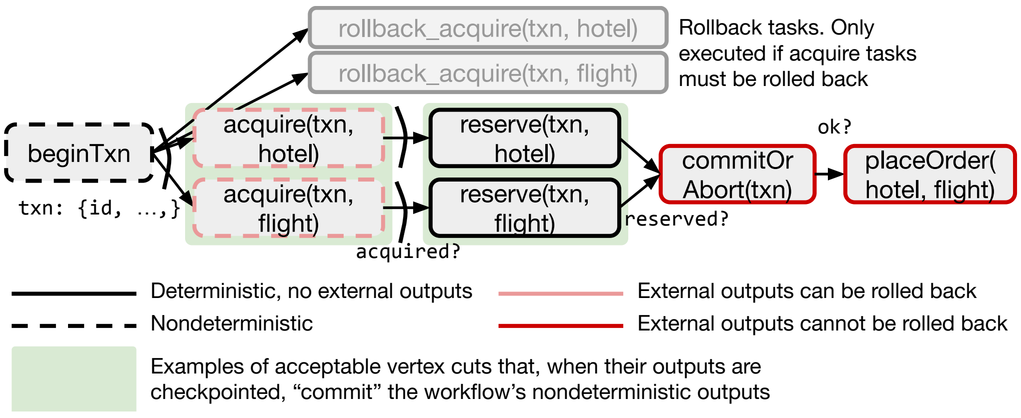
Non-idempotent tasks with external outputs that can be rolled back should set `can_rollback=True` and the `rollback` callback. On failure, Exoflow executes these rollback "tasks" in reverse dependency order before resuming execution. The rollback task can take any arguments available to the original workflow task, but the application must additionally guarantee that the rollback task is idempotent. For example, to implement the transaction in Figure 3.3, 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 3.3.3 makes this precise.

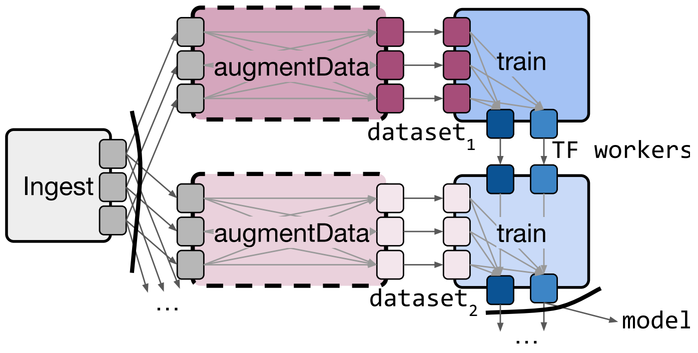
Internal outputs. Direct task outputs are subject to limits of the execution backend. For greater flexibility, Exoflow allows outputs to include `Refs` created by the task. `Refs` are (optionally) pluggable by the execution backend. They are intended to capture volatile outputs that would be expensive 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`.

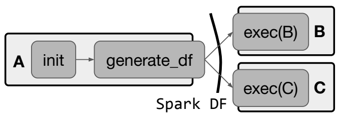
By default, `Ref` values are *immutable*. This improves recovery efficiency, as it simplifies checkpointing and minimizes task rollback. To capture task outputs that are expensive or impossible to materialize, we also support *stateful* references, i.e. *actors* [47]. An `ActorRef` extends `Refs` with application-defined methods that


rollback_acquire(txn, hotel)
rollback_acquire(txn, flight)
Rollback tasks. Only executed if acquire tasks must be rolled back
beginTxn
txn: {id, ...,}
acquire(txn, hotel)
acquire(txn, flight)
acquired?
reserve(txn, hotel)
reserve(txn, flight)
reserved?
ok?
commitOr Abort(txn)
placeOrder(hotel, flight)
Deterministic, no external outputs
Nondeterministic
External outputs can be rolled back
External outputs cannot be rolled back
Examples of acceptable vertex cuts that, when their outputs are checkpointed, "commit" the workflow's nondeterministic outputs

(a)



(b)



(c)

Figure 3.4: **(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.

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 3.3.4).

3.3.2 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:

- F_i: An associated function
- $\mathcal{N}_i$: A function representing a (potential) source of nondeterminism
- $\mathcal{R}_i$: An optional rollback function

- The set of annotations described in Table 3.1.

A workflow execution produces one internal and one external output per vertex, both optional. For brevity, the presented model only considers tasks with single outputs, although the system in reality supports multiple outputs.

We denote an execution's outputs by O_{Int} and O_{Ext}. O is a mapping from vertex to a single output value o, and the subscripts Int and Ext denote internal and external outputs, respectively. F_i outputs o_{Ext} by adding it to a global set $\mathcal{W}$, which can be read by other tasks and by external processes.

Each F_i takes as inputs:

- $args_i$: Direct arguments, one for each vertex with an edge to v_i.
- w_i: A set of external outputs.
- n_i: A nondeterministic value. If F_i's output does not depend on n_i, then F_i is deterministic.

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 [35], 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 $\mathcal{R}_i$).

We define a *failure-free* execution of G as one where the individual output of each task v_i corresponds to an execution of F_i over inputs such that:

- The direct arguments are the internal outputs produced by vertices with an edge to v_i. Formally, this can be written as $args_i = \{O_{Int}[j] \mid (v_j, v_i) \in E\}$.
- The set of external outputs is equal to the external outputs of all tasks that precede v_i in G. Formally, this can be written as $w_i = \{O_{Ext}[j] \mid v_j <_G v_i\}$.
- The nondeterministic value is one returned by $\mathcal{N}_i$, i.e. $n_i = \mathcal{N}_i()$.

The correctness condition says that to an external process, it must appear as if the DAG has executed failure-free. Thus, we also define W: a sequence of snapshots of the external outputs produced so far by the DAG execution. W represents a series of reads of $\mathcal{W}$ made by an external process during execution. Then, we just need to make sure that once an external output is visible, i.e. it appears in a snapshot w in W, it should be visible in all following snapshots in W. In other words, W must be monotonic.

This definition is analogous to *global consistency* in message-passing [35], 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. Formally, we can define this as:

Definition 3.1 (Consistency). *O_{Int}, O_{Ext} are consistent with a workflow $G = (V, E)$ if for all possible W, W is monotonic and the outputs O_{Int} and O_{Ext} correspond to a failure-free execution of G.*

The application assumptions are as follows. For each v_i:

1. We assume that if v_i and v_j cannot be ordered in the graph, then they cannot read each other's external outputs. Formally, if $v_i \not<_G v_j$ and $v_j \not<_G v_i$, then $F_i(I_{Int}, w_i, n_i) = F_i(I_{Int}, w_i \setminus \{O_{Ext}[j]\}, n_i)$. If the application requires v_i to depend on a task v_j's external output, then the ordering should be specified as part of the task graph. If this is not possible, then to ensure consistency, v_j's external output should be considered part of v_i's nondeterministic input, and the application must set `can_rollback=False` for v_j.

2. Tasks that set `deterministic=true` must produce outputs that are a deterministic function of their internal and external outputs, i.e. F_i is not dependent on the value returned by $\mathcal{N}_i$.

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 an invocation of F_i produces an external output o_{Ext}, and F_i is run again on the same internal outputs and a later snapshot of $\mathcal{W}$, then F_i should still produce the same external output.

   (b) If $\mathcal{R}_i$ is provided, then it is a deterministic and idempotent function of the task's *internal inputs* only. Intuitively, $\mathcal{R}$ removes the previous external output from the external world. Formally, this means that if the first invocation of $F_i(I_{Int}, w, n_i)$ produces (o_{Int}, o_{Ext}), then $\mathcal{R}_i(I_{Int})$ removes o_{Ext} from all past reads of $\mathcal{W}$.

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 3.3.4, which discusses the rules that the application must follow to ensure exactly-once semantics when `ActorRefs` are passed between workflow tasks.

3.3.3 Guaranteeing exactly-once execution

Task annotations simplify the decision of when to commit task outputs. To illustrate this, we use Figure 3.4a, a modified version of the workflow described in Figure 3.3. 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 3.4a 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 `acquire` 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 3.1 (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 (green shaded box in Figure 3.4a) 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 3.4a is the righthand vertex cut, which instead checkpoints 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 3.4a 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 3.4a, the application must set `checkpoint=True` for `beginTxn`, and Exoflow synchronizes this checkpoint before executing the `acquire` tasks.

Invariant 3.2 (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 3.1, 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 3.1. This specification trivially satisfies the invariants and indeed corresponds to current workflow systems that commit all task outputs. Section 3.4 describes how Exoflow leverages the invariants to improve run-time performance for more sophisticated specifications.

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.

3.3.4 References

Immutable `Refs` enable efficient passing of large and distributed data between workflow tasks. For example, Figure 3.4b shows how the `Ingest` task from Figure 3.2d 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 3.4b 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.

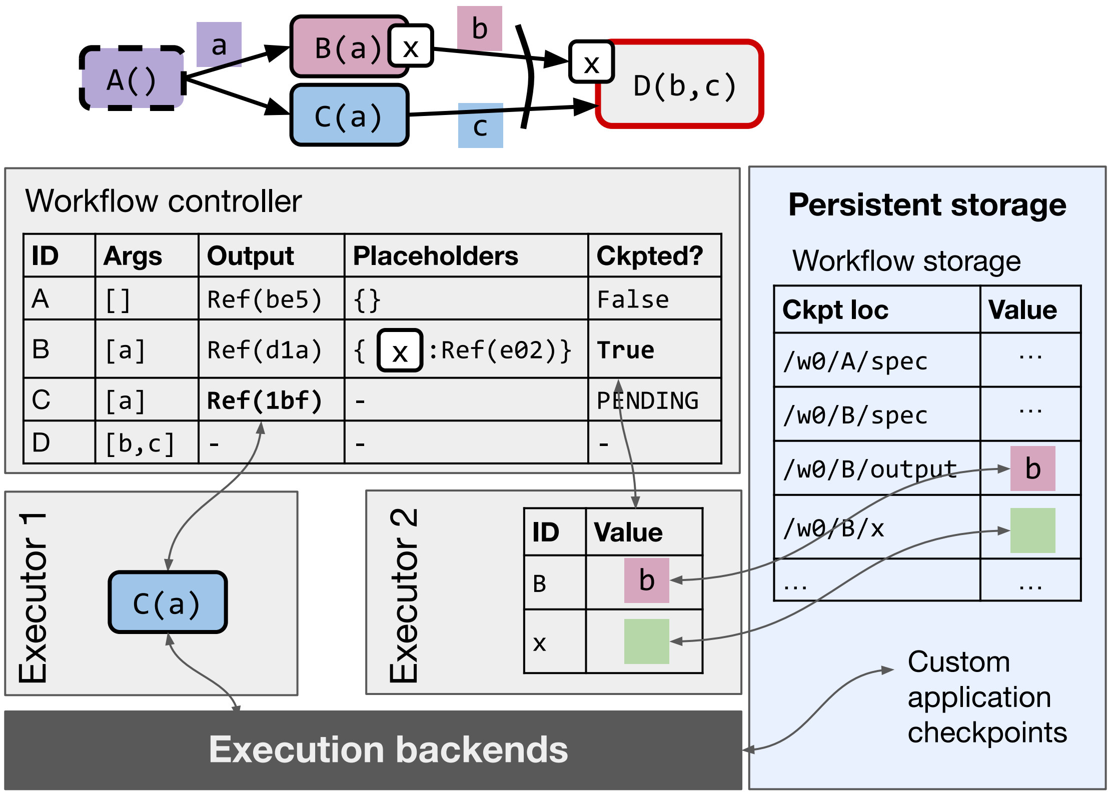
```
@ray.remote
class SparkActor:
  def __init__(self):
    self.spark_context = connect(); self.df = None
  def generate_df(self):
    self.df = generate_df(self.spark_context).cache()
  @const
  def exec(self, seed: int) -> int:
    return exec(self.df, seed=seed).count()
  def _checkpoint(self):
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

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 3.4b, 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 [24].

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 3.4c shows an expanded Figure 3.2b 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.



ID	Args	Output	Placeholders	Ckpted?
A	[]	Ref(be5)	{}	False
B	[a]	Ref(d1a)	{ x :Ref(e02)}	**True**
C	[a]	**Ref(1bf)**	-	PENDING
D	[b,c]	-	-	-

ID	Value
B	b
x	🟩

Ckpt loc	Value
/w0/A/spec	…
/w0/B/spec	…
/w0/B/output	b
/w0/B/x	🟩
…	…

Figure 3.5: 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`).

3.4 Architecture

The Exoflow architecture (Figure 3.5) comprises a logically centralized workflow controller, a pluggable execution backend, and a pluggable persistent storage system.

The Exoflow controller is a long-running service that can be sharded by workflow (Figure 3.5). 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. Checkpointing is carried out asynchronously by background threads on the executors, enabling parallel and distributed checkpoints that are not bottlenecked by the centralized controller. The Exoflow controller coordinates checkpoint synchronization during execution as needed, according to the user-defined annotations. Then, on restart, the controller simply scans the storage for any unfinished workflows, coordinates rollback as needed, and re-`runs` to completion.

See Appendix B for a full description of the execution and recovery procedures, including correctness arguments and implementation details.

3.5 Evaluation

Our evaluation covers the following questions:

1. How can applications leverage first-class references and task annotations to have greater flexibility in recovery?
2. How does this flexibility in recovery strategy affect performance during execution and recovery?

Appendix B includes additional end application evaluation, as well as microbenchmarks evaluating:

1. What overheads does Exoflow add to at-least-once or at-most-once execution backends?

We compare primarily against these baselines: (1) exactly-once workflow systems: Airflow [3], "standard mode" AWS Step Functions [18], and the serverless workflow system Beldi [104]; and (2) at-least-once distributed DAG systems: "express mode" AWS Step Functions [18] and Ray [67].

Given the high execution overheads of exactly-once workflow systems such as Airflow (Appendix B.3.2), 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 3.5.2) for persistent storage.

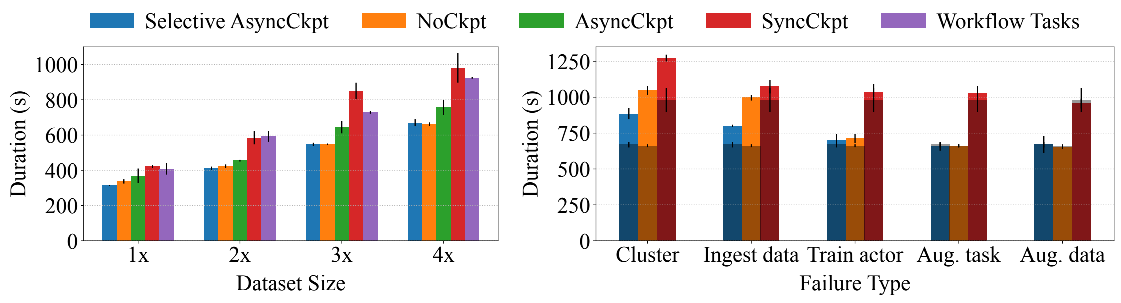


Figure 3.6: End-to-end duration for the ML workflow application shown in Figures 3.2d and 3.4b. **Left:** End-to-end duration without failure. **Right:** End-to-end duration with different failure types. The shadow represents the execution time without failure.

3.5.1 ML training pipelines

We show how Exoflow enables a flexible recovery-performance tradeoffs for the workflow in Figure 3.2d. 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 3.6 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 3.6L 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 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 3.2c) 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 3.6R 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 3.6L and R demonstrate how *the developer can flexibly choose the best recovery strategy.*

Figure 3.6R 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 [100]. 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.

3.5.2 Stateful serverless workflows

We compare Exoflow on a travel reservation benchmark [38] to Beldi [104], 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 [104], and we report response latency in Figure 3.7a.

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[1]. When requests/s is higher than 700, Exoflow's median latency is greater than Beldi's. This is due to

[1]Note that unlike Beldi, Exoflow requires a server. However, because Exoflow's controller is fault-tolerant and horizontally scalable, it would be straightforward to deploy Exoflow as a serverless system using any autoscaling container orchestrator.

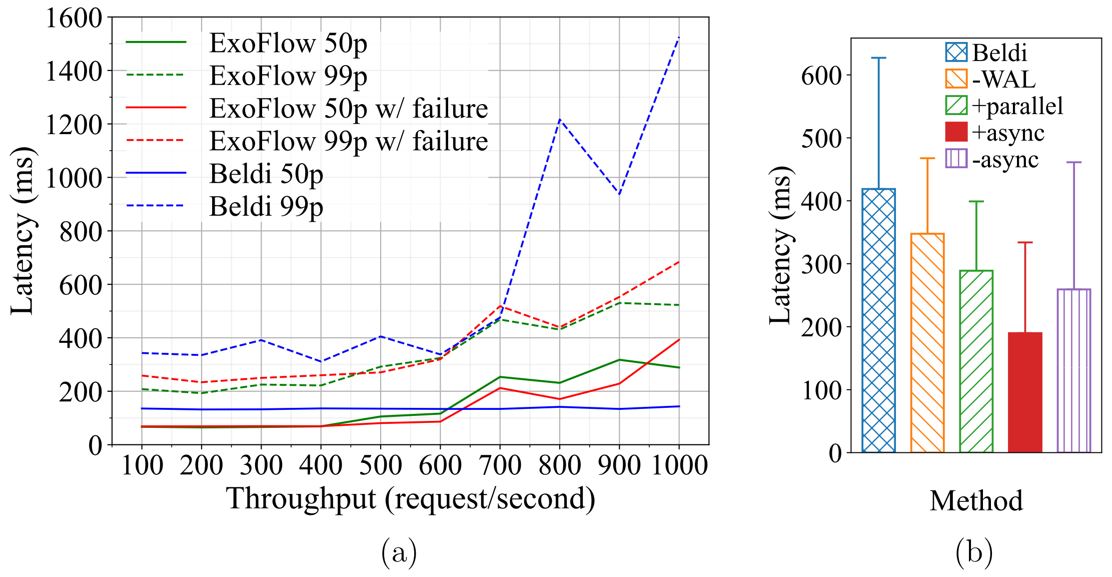


Figure 3.7: **(a)** Response latency percentile for a serverless travel reservation benchmark [38]. **(b)** Median latency of the trip reservation request from the travel reservation benchmark. Error bar represents 99-percentile latency.

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 3.7a 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 ([104] evaluates 1min as a possible lower bound).

Figure 3.7b 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 3.3. 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 3.4a. `-async` shows that with synchronous checkpoints, 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.

3.6 Related Work

Workflow systems. Industry workflow systems [6, 3, 5, 18] 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 3.2).

Many workflow systems for FaaS focus on stateful serverless workflows. Several provide a fault-tolerant transactional key-value store interface [104, 94, 93]. 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 [20, 7, 19]. 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 [35, 69]. Exoflow also supports pluggable actors but only with coarse-grained logging (i.e. recording the workflow DAG) and checkpoint-based recovery (Section 3.3.4). This is intentionally minimal, as it enables composition of both log- and checkpoint-based actor implementations.

Exoflow is similar to DARQ [61]: 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 [34, 51, 102]. Several use *lineage reconstruction* for recovery, a form of logging that records the DAG but not the data, to reduce run-time overhead. CIEL [71, 73] 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 [67], which we also adopt, but cannot support exactly-once semantics or persistence [100]. Tachyon [58] 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 [69], Apache Flink [22] and Canary [83] 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 [35]. Exoflow augments log- and checkpoint-based systems by orchestrating recovery across systems with different internal strategies (Appendix B.3.1).

Falkirk Wheel [40] 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 [47]. Most systems do not guarantee exactly-once semantics [14, 2, 21, 100]. 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 [35]. Our work adapts these techniques to the distributed workflow setting and aims to compose log- and checkpoint-based applications.

3.7 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 data processing abstractions such as RDDs [102] or timely dataflow [69] that are tightly coupled to a specific execution 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.

These decisions are intentional. Pluggability for data movement is important for allowing applications to decide the best way to move data from one place to another. Actors are important because many execution frameworks have some type of context that should be passed between logical steps of an end-to-end workflow, e.g., the driver state in Spark. Neither of these is necessary in an execution framework that natively handles all worker communication and process state.

Serializability is of course important for moving any type of data across process boundaries. Supporting serializable references further allows moving large and potentially distributed data by reference instead of needing to first copy the values into one central location. In contrast, serializing an RDD or timely dataflow graph makes little sense; the deserialized copy may be useless if the receiver is not in the same cluster.

Finally, using a generic task parallelism model allows references to be flexibly passed between applications. In contrast, consuming data within a typical dataflow system often requires the consumer to be expressed as part of the dataflow graph, or else for the system to provide special data connectors to third-party systems.

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.

3.8 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.

Chapter 4

Conclusion

In this dissertation, we have addressed the challenge of providing efficient fault tolerance for a wide range of distributed applications by exploiting the semantics of workloads at all layers of distributed systems. We have focused on two key layers: the communication layer and the task execution layer.

At the communication layer, we introduced Hoplite, a distributed object store that exploits the dynamic data transfer patterns on-the-fly and transfers data with fine-grained pipelining to gain efficiency. Hoplite also enables fast data transfer rescheduling upon task failure to keep making progress. We demonstrated that Hoplite significantly speeds up applications like asynchronous stochastic gradient descent, reinforcement learning, and serving an ensemble of machine learning models, which are difficult to execute efficiently with traditional collective communication.

At the task execution layer, we presented ExoFlow, a universal workflow system that decouples execution from recovery by allowing applications to specify task semantics through annotations. By exploiting the semantics of tasks and data passing between tasks, ExoFlow enables a flexible choice of recovery strategies, optimizing the performance-fault tolerance tradeoff for heterogeneous applications. We showed that ExoFlow generalizes recovery for existing workflow applications ranging from ETL pipelines to stateful serverless workflows, while enabling further optimizations in task communication and recovery.

The techniques and systems presented in this dissertation demonstrate the potential of a semantics-aware approach to building efficient and fault-tolerant distributed systems. By leveraging a general set of application-specific knowledge at both the communication and task execution layers, we can achieve significant performance improvements while maintaining strong fault tolerance guarantees, for a wide range of emerging distributed applications.

In building these systems, there are some lessons I would like to share. Also I will discuss some future directions for universal, efficient and fault tolerant distributed systems.

4.1 Lessons Learned

Choosing the right application semantics is crucial Throughout this dissertation, we have demonstrated the importance of leveraging application semantics to optimize performance and fault tolerance in distributed systems. However, deciding which semantics to utilize in a system is both tricky and interesting. Incorporating too many semantics can complicate the system design and limit its universality, while using too few semantics leaves little room for optimization. From my experience, a helpful approach is to compare with existing systems and consider the minimal set of semantics to inherit or add, enabling key optimizations. In Hoplite, we compared with peer-to-peer (P2P) networking systems and object stores, inheriting the semantics of data transfer from P2P systems and the put/get semantics from object stores. We then added a *reduce* operation to enable efficient optimization for other types of collective communication. Similarly, in ExoFlow, we inherited the data passing semantics of previous workflow or DAG systems, such as *internal and external outputs*, while introducing *non-determinism and rollback semantics* for optimizing checkpointing.

Focusing on systems rather than specific techniques or applications Initially, ExoFlow targeted only data-intensive workloads, making it challenging to differentiate from existing works that focus on checkpoint optimization. However, as we shifted our focus to workflow systems, we gained a broader perspective. Despite significant differences in programming interfaces, we realized that data workflow systems like Apache Airflow were not fundamentally different from serverless workflow systems, as both are based on dataflow graphs and provide exactly-once semantics. This realization led to the introduction of user annotations, resulting in a more universal system applicable to a wider range of applications.

Collaborating with other system builders is invaluable When I was a first-year PhD student, deciding on a research direction was challenging. Besides reading papers, I struggled with finding real problems to solve. The answer came from collaborating with the Ray team. By investigating and addressing issues in the Ray project, I learned that some challenges go beyond mere engineering efforts and are worthy of research exploration. Both Hoplite and ExoFlow were partially inspired by the real challenges I encountered while working with the system builders of Ray. This collaboration provided invaluable insights and helped shape the direction of my research.

4.2 Limitations and Future Work

Automating application annotation One common limitation for both Hoplite and ExoFlow is their requirement of manual annotations, which hinders the adoption of the systems. For example, Hoplite requires explicit declaration of the *reduce* collective communication pattern, while common distributed object stores only support *get* and *put* operations. In ExoFlow, users have to manually annotate whether a task is deterministic or not. Automating application annotations is an important area for future research.

There are several potential approaches to automating application annotations. One approach is to use static program analysis to identify and match corresponding patterns in the application code. By analyzing the code structure, data dependencies, and communication patterns, it may be possible to infer the appropriate annotations automatically. Another promising direction is to leverage the recent advancements in large language models (LLMs) and AI agents to predict annotations for new applications. AI agents could also be employed to interact with the application, observe its behavior, and infer the necessary annotations based on the observed semantics.

Currently, the task annotations in Hoplite and ExoFlow are quite coarse-grained, which makes the systems general-purpose but limits the opportunities for optimization. With automated annotation, it becomes feasible to support more fine-grained annotations, enabling further performance improvements. For example, instead of annotating an entire task as deterministic or not, automated annotation could identify specific parts of the task that are deterministic, allowing for more targeted optimizations. Fine-grained annotations could also capture more detailed information about data dependencies, communication patterns, and resource requirements, enabling more efficient scheduling and resource management.

Efficient fault tolerance for AI agents The emergence of large language model (LLM) based AI agents has opened up new possibilities for intelligent autonomy in distributed systems. These AI agents are capable of interacting with the environment, reasoning about complex tasks, and collaborating with other agents to solve problems [101, 105]. As the scale and complexity of these systems grow, ensuring their reliability and fault tolerance becomes increasingly important. However, the failure scenarios in AI agent-based systems differ from traditional distributed systems. In traditional systems, failures are typically associated with clear system-level events, such as node crashes or network partitions, and the recovery strategies are well-defined, such as checkpoint-restart or state machine replication. In contrast, failures in AI agent-based systems often manifest as semantic errors in the outputs of the LLMs. These errors can be subtle, context-dependent, and difficult to detect using traditional failure detection mechanisms. Detecting and correcting semantic errors in AI agent-based systems is a challenging problem. The outputs of LLMs are often probabilistic and can vary based on the input context and the underlying training data. Defining what constitutes an error can be subjective and application-specific. Moreover, the complexity and scale of these systems make it infeasible to manually inspect and verify the correctness of every output.

One promising approach to address this challenge is to exploit the semantics of the workloads and use them to verify the correctness of the LLM outputs. This could involve using another set of domain experts specifically designed for error detection and correction in specific areas. Figure 4.1 illustrates our proposed architecture. On the left side, we have an AI agent application where we have access to the outputs of internal LLMs and the application semantics during interactions, such as using tools. On the right side, we have a verification leader for checking the correctness of the output. Our design is inspired by meta-prompting [95], where we have a task-agnostic leader ("verification leader") that automatically chooses experts that match the application semantics for verifying the AI agent output. The verification leader and domain experts combined form a multi-agent architecture. The insight behind this architecture is that we can leverage AI agents to unleash the potential of LLMs, which can have stronger capabilities than a single LLM and may be better at verifying the output than the LLM that generated it in the application. This multi-agent framework naturally introduces sparsity, making the entire process more efficient. By distributing the verification tasks among multiple specialized agents, we can achieve a higher degree of parallelism and reduce the computational burden on any single agent. Each domain expert can focus on a specific aspect of the application semantics, allowing for more targeted and efficient error detection and correction. The verification leader acts as a coordinator, dynamically assigning tasks to the most suitable domain experts based on the nature of the input and the application context.

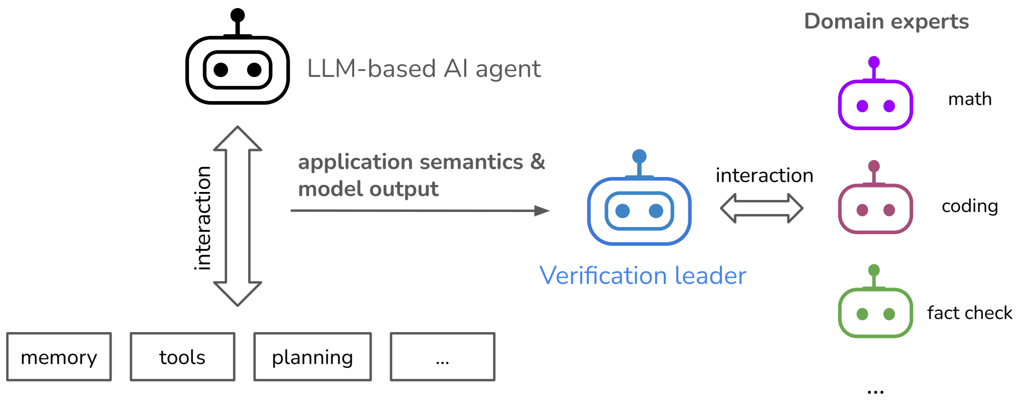


Figure 4.1: A multi-agent architecture for efficient fault tolerance in AI agent-based systems.

Recently, there have been a few closely related works that can be integrated with the AI agent approach. Prometheus 2 [54] trains models with a new dataset that enables a more powerful evaluator LM than its predecessor, closely mirroring human and GPT-4 judgments with custom evaluation criteria. Similarly, we can train our agents on the expected semantics of the workloads and use this knowledge to identify anomalies or inconsistencies in the outputs of the primary agents. PoLL [97] evaluates LLM generations with a panel of diverse models to reduce costs and intra-model bias.

We can achieve the same goal by comparing the outputs of multiple agents and cross-referencing them with the expected semantics.

Developing efficient fault tolerance mechanisms for AI agent-based systems is an exciting area for future research. It requires a deep understanding of the characteristics of LLMs, the semantics of the workloads, and the unique failure modes in these systems. Collaboration between researchers in distributed systems, machine learning, and natural language processing will be essential to address these challenges and build reliable and fault-tolerant AI agent-based systems.

In conclusion, we believe that in the future, heterogeneous distributed systems running sophisticated tasks will be the norm, with the exponential growth of computational demand. This dissertation demonstrates the power of exploiting application semantics for efficient fault tolerance, while keeping the system universal. We hope the insights and lessons learned from this dissertation will serve as valuable resources for researchers and practitioners and play a crucial role in shaping the future of the field.

Bibliography

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

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

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[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] EPOCH AI. Training compute of notable machine learning systems over time. https://epochai.org/data/epochdb/visualization, 2024. (Accessed on 05/08/2024).

[11] Amazon s3. object storage built to store and retrieve any amount of data from anywhere. https://aws.amazon.com/s3/, 2020.

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

[13] 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.

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

[15] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1387–1395, 2017.

[16] 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.

[17] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] M. Castro, P. Druschel, A. . Kermarrec, and A. I. T. Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, Oct 2002.

[26] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth multicast in cooperative environments. *SIGOPS Oper. Syst. Rev.*, 37(5):298–313, October 2003.

[27] 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.

[28] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 393–406, New York, NY, USA, 2015. Association for Computing Machinery.

[29] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *SIGCOMM Comput. Commun. Rev.*, 41(4):98–109, August 2011.

[30] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 443–454, New York, NY, USA, 2014. Association for Computing Machinery.

[31] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 613–627, 2017.

[32] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[33] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. volume 51, page 107–113, New York, NY, USA, January 2008. Association for Computing Machinery.

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

[35] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[36] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1407–1416. PMLR, 2018.

[37] 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.

[38] 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.

[39] Collective communications library with various primitives for multi-machine training. `https://github.com/facebookincubator/gloo`, 2020.

[40] 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.

[41] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[42] Richard L Graham, Timothy S Woodall, and Jeffrey M Squyres. Open mpi: A flexible high performance mpi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 228–239. Springer, 2005.

[43] gRPC. `https://grpc.io/`, 2020.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[45] 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. In *9th Biennial Conference on Innovative Data Systems Research (CIDR 2019)*, 2019.

[46] 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.

[47] 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, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[48] Hydro. `https://github.com/hydro-project`, 2020.

[49] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[50] Ip multicast technology overview . `https://www.cisco.com/c/en/us/td/docs/ios/solutions_docs/ip_multicast/White_papers/mcst_ovr.html`, 2020.

[51] 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.

[52] 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.

[53] Keynote: Building a fusion engine with ray. `https://ray2020.sched.com/event/eGOL/keynote-building-a-fusion-engine-with-ray-dr-charles-he-chief-architect-of-storage-and-compute-ant-group`, 2020.

[54] Seungone Kim, Juyoung Suk, Shayne Longpre, Bill Yuchen Lin, Jamin Shin, Sean Welleck, Graham Neubig, Moontae Lee, Kyungjae Lee, and Minjoon Seo. Prometheus 2: An open source language model specialized in evaluating other language models. *arXiv preprint arXiv:2405.01535*, 2024.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[56] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. *SIGCOMM Comput. Commun. Rev.*, 44(4):467–478, August 2014.

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

[58] 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.

[59] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[60] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[61] 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.

[62] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[63] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018.

[64] 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.

[65] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd*

International Conference on International Conference on Machine Learning - Volume 48, ICML'16, page 1928–1937. JMLR.org, 2016.

[66] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and et al. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.

[67] 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.

[68] MPICH. `https://www.mpich.org/`, 2020.

[69] 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.

[70] Derek G Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: a universal execution engine for distributed data-flow computing. 2011.

[71] 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.

[72] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

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

[74] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[75] The nvidia collective communication library (nccl). `https://developer.nvidia.com/nccl`, 2020.

[76] Numpy. `https://numpy.org/`, 2020.

[77] Christopher Olston, Fangwei Li, Jeremiah Harmsen, Jordan Soyke, Kiril Gorovoy, Li Lao, Noah Fiedel, Sukriti Ramesh, and Vinu Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[78] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[79] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[80] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 421–434, New York, NY, USA, 2015. Association for Computing Machinery.

[81] 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, Boston, MA, February 2019. USENIX Association.

[82] 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.

[83] Hang Qu, Omid Mashayekhi, David Terei, and Philip Levis. Canary: A scheduling architecture for high performance cloud computing. *arXiv preprint arXiv:1602.01412*, 2016.

[84] Parameter server. `https://ray.readthedocs.io/en/latest/auto_examples/plot_parameter_server.html`, 2020.

[85] Ray serve. `https://docs.ray.io/en/master/serve/`, 2021.

[86] Redis. `https://redis.io/`, 2020.

[87] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, number 130-136. Citeseer, 2015.

[88] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

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

[90] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[91] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow, 2018.

[92] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[93] 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.

[94] 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.

[95] Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

[96] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[97] Pat Verga, Sebastian Hofstatter, Sophia Althammer, Yixuan Su, Aleksandra Piktus, Arkady Arkhangorodsky, Minjie Xu, Naomi White, and Patrick Lewis. Replacing judges with juries: Evaluating llm generations with a panel of diverse models. *arXiv preprint arXiv:2404.18796*, 2024.

[98] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 172–186, 2020.

[99] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: Fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 338–352, New York, NY, USA, 2019. Association for Computing Machinery.

[100] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for Fine-Grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686, Virtual, April 2021. USENIX Association.

[101] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[102] 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.

[103] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, and et al. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.

[104] 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.

[105] Mingchen Zhuge, Haozhe Liu, Francesco Faccio, Dylan R Ashley, Róbert Csordás, Anand Gopalakrishnan, Abdullah Hamdi, Hasan Abed Al Kader Hammoud, Vincent Herrmann, Kazuki Irie, et al. Mindstorms in natural language-based societies of mind. *arXiv preprint arXiv:2305.17066*, 2023.

Appendix A

Hoplite system design and evaluation

A.1 Implementation

The core of Hoplite is implemented using 3957 lines of C++. We provide a C++ and a Python front-end. The Python front-end is implemented using 645 lines of Python and 275 lines of Cython. We build the Python front end because it is easier to integrate with Ray [66] and other data processing libraries (e.g., Numpy [76], TensorFlow [8], PyTorch [78]). The interface between the Python front-end and the C++ backend is the same as Hoplite's API (Table 2.1).

We implement the object directory service using a set of gRPC [43] server processes distributed across nodes. Each directory server can push location notifications directly to an object store node. Each object store node in Hoplite is a gRPC server with locally buffered objects. Upon a transfer request from a remote node (e.g., during `Get`), the node sets up a direct TCP connection to the remote node and pushes the object buffer through the TCP connection.

In our experiments, we observe that setting d to $1, 2$, or n in the tree reduce algorithm is enough for our applications. When a task calls `Reduce`, Hoplite picks d from $1, 2$ and n that minimizes the estimated total latency based on the network latency L, bandwidth B, and the object size S. §A.3 shows the effect of different choices of d.

A.2 Microbenchmarks on Small Objects

We present the microbenchmarks for multiple collective communication primitives for small objects (1KB, 32KB) in Figure A.1. Note that Hoplite stores object contents in object directory service for objects smaller than 64 KB (§2.3.2), so there is no collective communication for Hoplite. Again, we compare with Ray, Dask, OpenMPI,

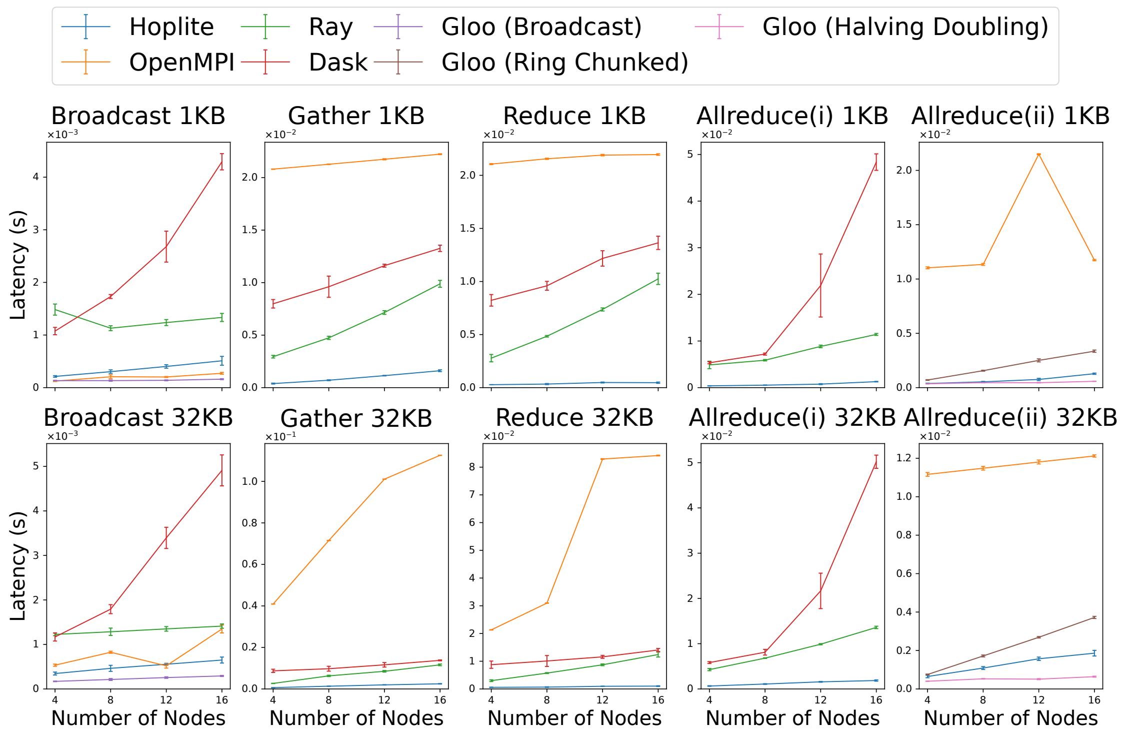


Figure A.1: Latency comparison of Hoplite, OpenMPI, Ray, Dask, and Gloo on standard collective communication primitives (e.g., broadcast, gather, reduce, allreduce) on 1KB and 32KB objects. To show the results more clearly, we split the results of Allreduce into two groups: group (i) includes Hoplite, Ray, and Dask, and group (ii) includes Hoplite, OpenMPI, and two different allreduce algorithms in Gloo.

and Gloo. We do not compare with Horovod for the same reason that Horovord has three backends: OpenMPI, Gloo, and NCCL. We have already compared with OpenMPI and Gloo. NCCL is for GPU, and Hoplite currently does not support GPU.

Hoplite is the best or close to the best among all these alternatives. Gloo has the best performance for broadcast and allreduce. Hoplite is more efficient than Ray, and Dask because Hoplite uses stores the object data directly in object directory service.

A.3 Ablation Study on Reduce Tree Degree

Here we study the choice of d in the AWS EC2 setting (§2.4). The best choice of d depends on network characteristics, the size of the object to reduce, and the number of participants. We compare three choices of d: 1 (a single chain), 2 (a binary tree), and n (a root connects everyone else). The results are in Figure A.2. As expected from our analysis in (§2.3.4), when the object size is small, $d = n$ is the best because the main bottleneck is the network latency. When the object size is medium (256KB, 1MB), $d = n$ becomes unstable for reduce. We suspect that this is due to incast or due to gRPC characteristics. When object size is 4MB or 8MB, we need to choose between $d = 1$ and $d = 2$ based on the number of participants. This is because both network latency and network throughput can be a bottleneck in tree reduce. When object size is 16MB or larger, we choose $d = 1$ to mitigate the throughput bottleneck in reduce.

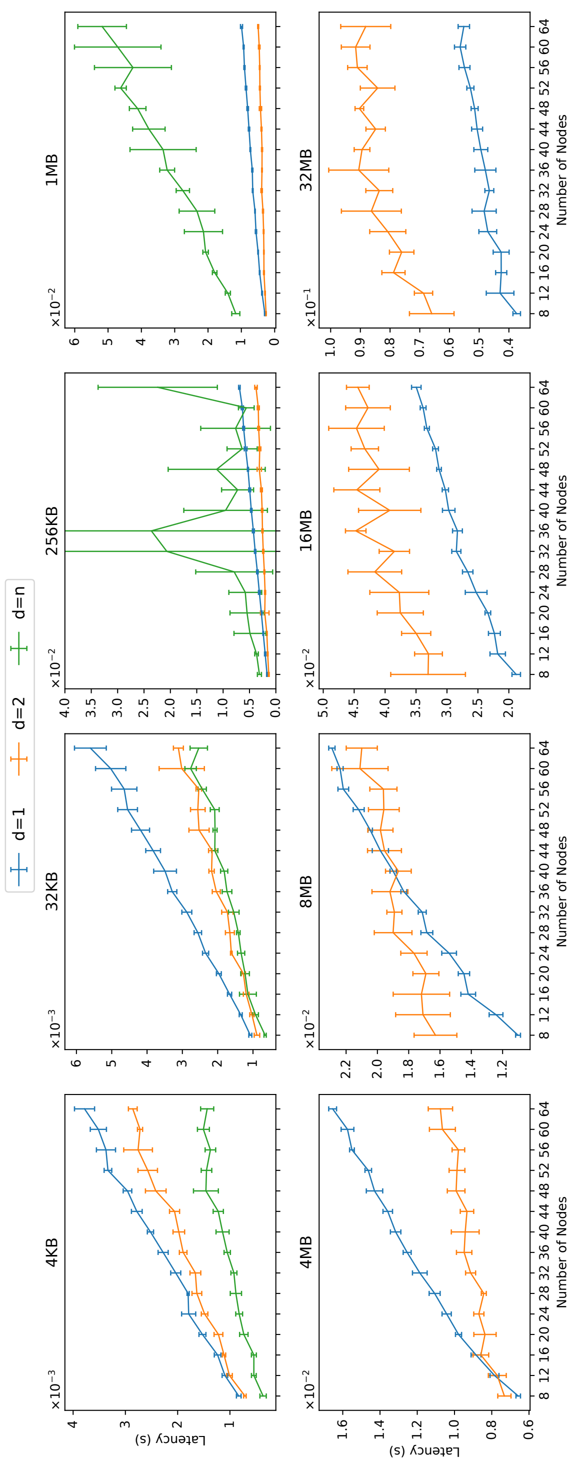


Figure A.2: Ablation study of reduce latency on the reduce tree degree d with different object size and number of participants.

Appendix B

Exoflow system design and evaluation

B.1 Architecture

We further describe the Exoflow design and the requirements of the pluggable execution backend and persistent storage.

B.1.1 Workflow execution

The workflow control layer is implemented using the system Ray [67]. 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 3.5). The controller uses Ray's *distributed futures* [100] 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 3.3).

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 3.5, 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 3.5). 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 3.5).

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 3.5 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 3.3.4). When a task finishes, the executor replaces any `Refs` and `ActorRefs` appearing in the task's output with placeholders, e.g., `x` in Figure 3.5. 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.

B.1.2 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 3.5 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 3.5) 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 3.1). 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 3.1 (Section 3.3.3) and checkpoint synchronization (Appendix B.1.1). The system synchronizes the partition provided by Invariant 3.1 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 3.2 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.

B.1.3 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 [100]. 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`).

B.2 Implementation

Exoflow is built on Ray v2.0.1, which uses gRPC [43] for tasks and actors and a custom shared-memory object store for `Ref` storage [67]. 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`.

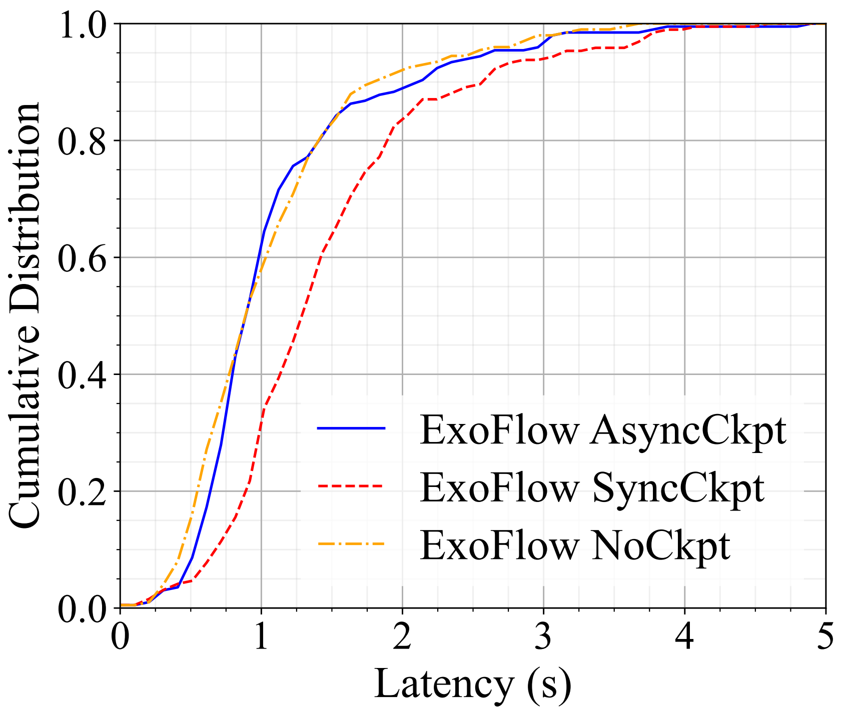


Figure B.1: **(c)** Latency CDF of online-offline graph processing.

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 [16], 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.

B.3 Evaluation

B.3.1 Online-offline graph processing

Distributed graph processing systems can be generally divided into stream vs. batch processing [64]. 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 [27], 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 [57] (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 B.1 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.

B.3.2 Microbenchmarks

Latency. With equivalent backends, Exoflow matches or reduces execution overheads of existing workflow systems while enabling more flexible inter-task communication. Figure B.2a (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 [18]) and at-least-once systems (AWS Express Step Function [18], Ray [67]) 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.

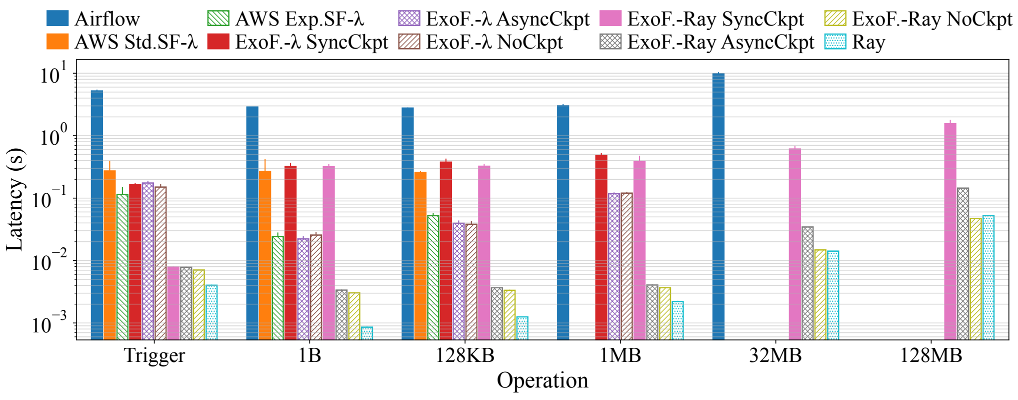
"1B" in Figure B.2a compares minimum task execution latency. Exoflow-Lambdas achieves comparable latency as AWS Step Functions, as the primary overheads for exactly-once and at-least-once execution come from durability and Lambdas invocation, respectively. Exoflow-Ray improves upon the latter as it uses 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 [89] 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.

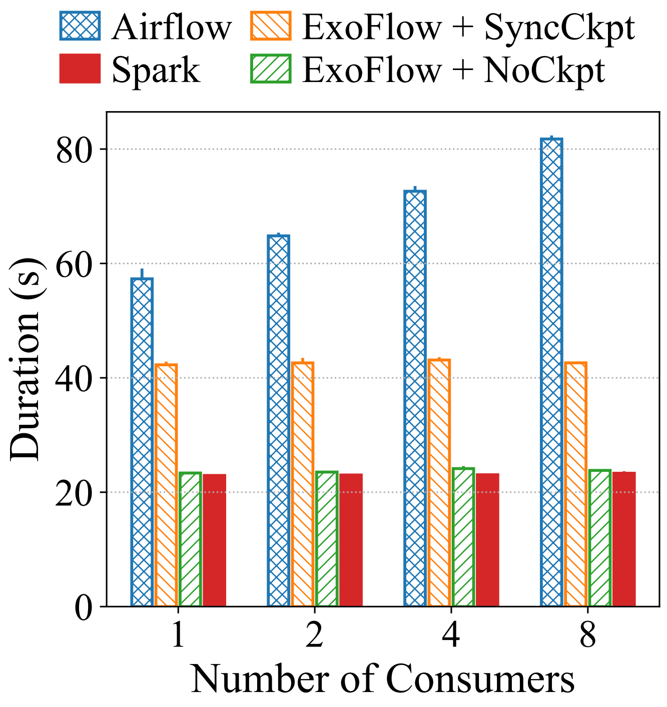
In summary, Exoflow's low execution overheads make it a practical replacement for existing workflow systems, and it enables greater flexibility in task communication and recovery.

Data sharing for ETL. We evaluate Exoflow against Airflow for a Spark workflow similar to Figure 3.2b (1 m5.8xlarge instance, 4GB Spark memory). Figure B.2b 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.

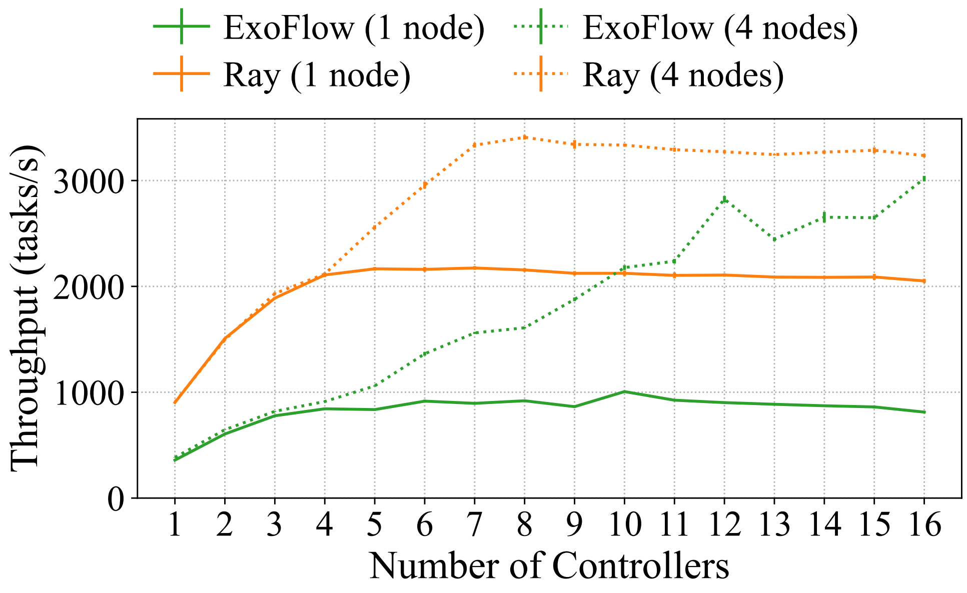


(a)

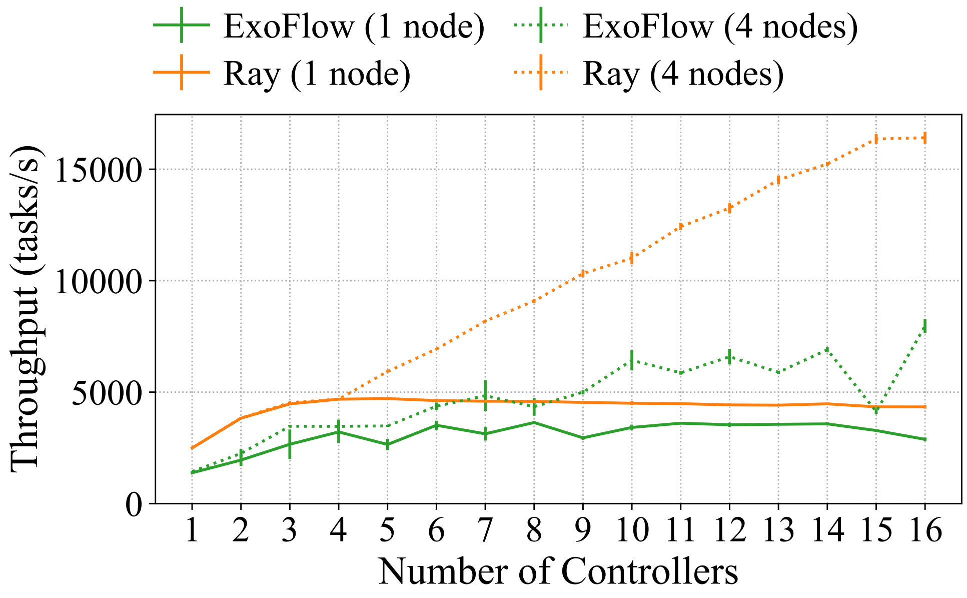


(b)

Figure B.2: 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 3.2b and 3.4c, compared with Airflow and native Spark.



(a)



(b)

Figure B.3: Microbenchmarks, cont. Maximum task throughput (a: 1 task/DAG; b: 100 tasks/DAG) of 10k tasks, compared against Ray as an optimal baseline, on 1 node and 4 nodes.

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 B.3a (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 B.3b (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.