THE UNIVERSITY OF CHICAGO

PROGRAMMING THE CONTINUUM: TOWARDS BETTER TECHNIQUES FOR DEVELOPING DISTRIBUTED SCIENCE APPLICATIONS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
JAMES GREGORY PAULOSKI

CHICAGO, ILLINOIS
JUNE 2025

ABSTRACT

Advances in networks, accelerators, and cloud services encourage programmers to reconsider where to compute—such as when fast networks make it cost-effective to compute on remote accelerators despite added latency. Workflow and cloud-hosted serverless computing frameworks can manage multi-step computations spanning federated collections of cloud, high-performance computing, and edge systems, but rely on simple abstractions that pose challenges when building applications composed of multiple distinct software with differing communication and patterns. This dissertation introduces new techniques for programming distributed science applications deployed across the computing continuum—research infrastructure that spans personal, cloud, edge, and high-performance computing (HPC) systems. TaPS, a benchmarking suite for reliable evaluation of parallel execution frameworks, is developed and used to investigate limitations in existing solutions. This investigation motivates the design of ProxyStore, a library that extends the pass-by-reference model to distributed applications with the goal of decoupling data flow from control flow. ProxyStore's object proxy paradigm enables the dynamic selection of different data movement methods, depending on *what* data are moved, *where* data are moved, or *when* data are moved—a long-standing challenge in distributed applications. Three high-level patterns—distributed futures, streaming, and ownership—extend the low-level proxy paradigm to support science applications spanning bioinformatics, federated learning, and molecular design, in which substantial improvements in runtime, throughput, and memory usage are demonstrated. Last, Academy, a federated agents system, supports the creation and deployment of pervasive autonomous agents, decentralizing control flow across stateful entities. These techniques encompass an open-source toolbox for developing novel and performant science applications and federated frameworks for the computing continuum.

"Programming (or problem solving in general) is the judicious postponement of decisions and commitments!" — Edsger W. Dijkstra

"Science is sort of a long, passive-aggressive argument about everything." — ZeFrank

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This work—and, more generally, experience—would not have been possible without the support of many, to whom I owe countless thanks.

To my advisors Kyle Chard and Ian Foster, thank you for this opportunity. My time in Chicago has been formative and rewarding. I tried, failed, learned, and succeeded; I would not have done it any different. The many side quests and paths I traveled down would not have been possible without your guidance and support. I have learned a great deal from both of you, and I hope to pass on those lessons to others.

To my PhD and Masters committee members Michael Franklin and Zhao Zhang, thank you both for your insights, feedback, and mentorship.

To the past and present members of Globus Labs, thank you for our friendships, adventures, and collaborations. I am fortunate to have joined a group that balances ambition with practicality, encourages fun and exploration, and values openness and mutual support. I cannot imagine a more positive environment to have spent the last five years.

To those who have contributed to this dissertation as co-authors on papers, technical support, or mentors, thank you: Yadu Babuji, André Bauer, Matt Baughman, Alex Brace, Ryan Chard, Ben Clifford, Maxime Gonthier, Valerie Hayot-Sasson, Nathaniel Hudson, Alok Kamatar, Haochen Pan, Klaudiusz Rydzy, Charlie Sabino, Tyler Skluzacek, Logan Ward, Sicheng Zhou, and the Globus Team.

To the innumerable colleagues with whom I have collaborated on projects not presented in this dissertation, thank you. I have learned so much through these projects and I would never have imagined the breadth of interdisciplinary collaborations that I would be engaged in.

And finally, to Betsy, my family, and my friends, thank you for your continued encouragement and support.

CHAPTER 1

INTRODUCTION

Computational methods and data-driven approaches—now pervasive across many scientific domains—have contributed to advancements that exceed those possible via traditional experimental methods [148]. These shifts in application requirements, increasing hardware heterogeneity, and faster and more reliable networks have stimulated the desire to execute scientific workflows across the *computing continuum* where applications are seamlessly deployed across federated collections of personal, cloud, edge, and high-performance computing (HPC) systems. Thus, programmers specify *what* task (e.g., command, function, or operation) to perform without regard to *where* they are executed; an execution engine then handles the mechanics of routing each task and data to a suitable processor.

Accordingly, the design of modern scientific workflows is increasingly shifting from monolithic programs to *applications* that are written as a composition of many distinct components, referred to as *tasks*. These task-centric approaches map well onto distributed and remote execution paradigms, such as function-as-a-service (FaaS) and workflow engines, that provide the computational flexibility, scalability, and reliability vital to abstracting the complexities of executing tasks in parallel. Yet, as the scale and ambition of task-parallel applications grow, they increasingly encounter difficulties managing the exchange of intermediate data among tasks—the *data flow*—and the autonomous coordination of actions and state across different resources—the *control flow*.

This thesis identifies limitations in distributed execution paradigms that inhibit scaling out science applications across the continuum. These limitations stem from the one-size-fits-all approaches employed by state-of-the-art execution engines. That is to say, these systems support a majority of use-cases well—often with impressive performance or simplicity—but designing and deploying more ambitious science applications poses unique challenges not well supported by existing systems. The ultimate goal of this thesis is to augment—not

replace—existing systems with new techniques that better address new challenges.

Paramount to the development of task-centric applications are the frameworks, such as Dask [219], Parsl [27], and Ray [184], that provide distributed and parallel execution. Research into these frameworks has accelerated as computational sciences increasingly need to take advantage of parallel compute and/or heterogeneous hardware. However, the lack of evaluation standards makes it challenging to compare limitations in existing implementations and to evaluate novel advancements. Thus, I begin by introducing TaPS, the Task Performance Suite, to support continued research in parallel task executor frameworks. TaPS provides (1) a unified, modular interface for writing and evaluating applications using arbitrary execution frameworks and data management systems and (2) a set of real-world science applications and synthetic benchmarks that function as benchmarking workloads.

A key insight learned from TaPS is that centralized systems and shared storage, commonly employed both by FaaS systems and workflow engines to facilitate data flow, can fail or become prohibitively expensive as the number of tasks, the geographic distribution of tasks, the volume of data exchanged, and the required speed of data exchange grow. An alternative approach to simplifying data sharing is the *object proxy paradigm*, which provides transparent access and management for shared objects in distributed settings and long used with Java's Remote Method Invocation (RMI) [40]. This paradigm inspires shifting the responsibilities of managing data flow from the execution frameworks to the intermediate objects themselves. This shift is achieved through *transparent object proxies* implemented to act as lightweight, wide-area references to objects in arbitrary data stores—references that can be communicated cheaply and resolved just-in-time via performant bulk transfer methods in a manner that is transparent to the consumer code. In this paradigm, proxies are self-contained and have both pass-by-reference and pass-by-value attributes. The eventual user of the proxied data gets a copy, but unnecessary copies are avoided when the proxy is passed among multiple processes. The goal of this abstraction is to decouple data flow com-

plexities from control flow-optimized execution engines, ultimately allowing developers to focus, when writing and composing distributed applications, on logical data flow rather than physical details of where data reside and how data are communicated. ProxyStore, described here, implements this proxy paradigm, and I apply ProxyStore to accelerate data flow management in high-performance scientific applications and programming frameworks.

This *low-level* paradigm raises a logical followup: What *high-level* patterns can build on the proxy model to accelerate and simplify the development of advanced applications? To this end, I design and implement proxy-based patterns for distributed futures, streaming, and ownership that make the power of the proxy pattern usable for more complex and dynamic distributed program structures. *ProxyFutures* enable seamless injection of data flow dependencies into arbitrary compute tasks to overlap computation and communication; *ProxyStream* decouples event notifications from bulk data transfer such that data producers can unilaterally determine optimal transfer methods; and *Proxy Ownership* provides client-side mechanisms for managing object lifetimes and preventing data races in distributed task-based workflows. These patterns are evaluated extensively through benchmarks and real-world scientific applications, in which we demonstrate substantial improvements in runtime, throughput, and memory usage.

Leveraging the computing continuum to execute scientific workflows will accelerate scientific discovery through automation, but at some point humans—rather than compute or data—become the limiting factor. Humans synthesize knowledge from prior research to propose hypotheses; design, debug, and deploy experiments and programs; and interpret results to inform new hypotheses. Intelligent agents, composing larger multi-agent systems, can be the driving entities instead—ultimately replacing the human in the loop. Enabling this new class of autonomous, *agentic workflows* necessitates new middleware and infrastructure solutions for expressing complex agents and orchestrating their deployment and coordination across federated resources. This thesis investigates three aspects of building multi-agent sys-

tems for science: how to program agents, asynchronous communication among agents, and execution of agents across federated resources. The near-term goal is to develop new techniques and tools that can support continued research in agentic workflows with the long-term goal of enabling truly autonomous discovery.

1.1 Thesis Statement

This thesis aims to demonstrate the following research statement:

New programming techniques enable and accelerate task-centric science applications executed across the computing continuum.

The desire to deploy scientific applications across research cyberinfrastructure and—more broadly—the computing continuum is apparent, yet key limitations (e.g., cost, features, and performance) in the frameworks used to coordinate remote execution and distributed data management lead to sub-optimal solutions or inhibit development entirely. I posit that new programming techniques are needed to enhance existing solutions and reduce the friction inherent to developing and deploying sophisticated science applications across diverse, federated resources. Specifically, I propose benchmarks for evaluating execution and data management frameworks, paradigms and patterns for coordinating wide-area data flow, and models and mechanisms for organizing the cooperation of distinct agents across diverse locations. Thus, by showing that these techniques enable and accelerate scientific discovery, then I validated my thesis.

1.2 Thesis Contributions

This thesis proposes new methods for the research and development of distributed and federated applications with the goal of understanding the limitations of existing solutions, introducing new techniques that enable broad patterns in distributed computing, advancing

community tools for the development of novel systems, and applying learnings to accelerate large-scale science. My objective is to develop techniques that enable new endeavors across the computational sciences, so that we may answer more questions, bigger questions, and harder questions. In each chapter, I motivate, describe, and evaluate one project that aims to further this vision.

In Chapter 2, I introduce **TaPS**, the Task Performance Suite. Task-based execution frameworks, such as parallel programming libraries, computational workflow systems, and function-as-a-service platforms, enable the composition of distinct tasks into a single, unified application designed to achieve a computational goal. As computational sciences increasingly rely on parallel computing and heterogeneous hardware, research into these execution frameworks has accelerated. However, the absence of standardized evaluation methodologies makes it difficult to systematically compare new frameworks against existing ones. TaPS supports continued research in parallel task executor frameworks through reference benchmarking workloads spanning real-world and synthetic applications and a robust interface and configuration system for evaluating execution frameworks and data management systems. Using TaPS, I perform an investigation of the performance characteristics of popular frameworks. These learning motivate the contributions of later chapters, which aim to address key challenges in data and control flow management.

In Chapter 3, I introduce **ProxyStore**. Advances in networks, accelerators, and cloud services encourage programmers to reconsider where to compute—such as when fast networks make it cost-effective to compute on remote accelerators despite added latency. Workflow and cloud-hosted serverless computing frameworks can manage multi-step computations spanning federated collections of cloud, high-performance computing (HPC), and edge systems, but passing data among computational steps via cloud storage can incur high costs. Here, I overcome these obstacles with a new programming paradigm that decouples control flow from data flow by extending the pass-by-reference model to distributed applications. Proxy-

Table 1.1: Summary of first-author papers composing the primary artifacts of this dissertation. Asterisks denote papers under review when this dissertation was submitted.

Title	**Ref.**	**Chapter**
TaPS: A Performance Evaluation Suite for Task-based Execution Frameworks	[196]	Chapter 2
Accelerating Communications in Federated Applications with Transparent Object Proxies	[195]	Chapter 3
Accelerating Python Applications with Dask and ProxyStore	[198]	Chapter 3
Object Proxy Patterns for Accelerating Distributed Applications	[197]	Chapter 4
Agentic Discovery: Closing the Loop with Cooperative Agents	*	Chapter 5
Empowering Scientific Workflows with Federated Agents	*	Chapter 5

Store implements this paradigm by providing object *proxies* that act as wide-area object references with just-in-time resolution. This proxy model enables data producers to communicate data unilaterally, transparently, and efficiently to both local and remote consumers. I demonstrate the benefits of this model with synthetic benchmarks and real-world scientific applications, running across various computing platforms, and through the integration of ProxyStore into frameworks for parallel computing and federated learning.

In Chapter 4, I extend the proxy model to support three high-level **proxy patterns**—distributed futures, streaming, and ownership—that enable the proxy paradigm to support more complex and dynamic distributed program structures. These patterns are motivated via careful review of downstream application requirements, and the implementations are evaluated through a suite of benchmarks. Further, I apply the patterns to three motivating scientific applications, in which substantial improvements in runtime, throughput, and memory usage are demonstrated.

In Chapter 5, I introduce **Academy**. Agentic systems, in which diverse agents cooperate to tackle challenging problems, are exploding in popularity in the AI community. However, the agentic frameworks used to build these systems have not previously enabled use with research cyberinfrastructure, relying on centralized control flow and cloud-centric deployments.

Table 1.2: Summary of papers I co-authored that describe applications and frameworks that utilize the primary artifacts of this thesis.

Title	**Ref.**
Colmena: Scalable Machine-Learning-Based Steering of Ensemble Simulations for High Performance Computing	[249]
GenSLMs: Genome-scale Language Models Reveal SARS-CoV-2 Evolutionary Dynamics	[267]
Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources	[251]
Employing Artificial Intelligence to Steer Exascale Workflows with Colmena	[248]
Flight: A FaaS-Based Framework for Complex and Hierarchical Federated Learning	[138]
Establishing a High-Performance and Productive Ecosystem for Distributed Execution of Python Functions Using Globus Compute	[18]
MOFA: Discovering Materials for Carbon Capture with a GenAI- and Simulation-Based Workflow	[263]
WRATH: Workload Resilience Across Task Hierarchies in Task-based Parallel Programming Frameworks	[266]
DynoStore: A Wide-Area Distribution System for the Management of Data over Heterogeneous Storage	[233]

ACADEMY is a modular and extensible middleware designed to deploy autonomous agents across the federated research ecosystem, including HPC systems, experimental facilities, and data repositories. To meet the demands of scientific computing, ACADEMY supports asynchronous execution, heterogeneous resources, high-throughput data flows, decentralized control flow, and dynamic resource availability. It provides abstractions for expressing stateful agents, managing inter-agent coordination, and integrating computation with experimental control. I present microbenchmark results that demonstrate high performance and scalability in HPC environments. To demonstrate the breadth of applications that can be supported by agentic workflow designs, I also present case studies in materials discovery, decentralized learning, and information extraction in which agents are deployed across diverse HPC systems.

The contents of each chapter correspond to one or two first-author papers, summarized in Table 1.1. The artifacts associated with these papers—the code, experiments, and analyses—

are open source on GitHub: `https://github.com/proxystore`. These artifacts have enabled numerous scientific applications spanning diverse domains and resulting in many papers, summarized in Table 1.2. These applications are used as motivation and evaluation throughout the chapters and highlight the impact that this work has had across the computational sciences.

CHAPTER 2

BENCHMARKING TASK-BASED PARALLEL EXECUTORS

Task-based execution frameworks, such as Dask [219], Parsl [27], and Ray [184], have enabled many advances across the sciences. These *task executors* manage the complexities of executing the tasks comprising an application in parallel across arbitrary hardware. Decoupling the application logic (e.g., what tasks to perform, how data flow between tasks) from the execution details (e.g., scheduling systems or communication protocols) simplifies development and results in applications which are portable across diverse systems. Task executors come in many forms, from a simple pool of processes to sophisticated workflow management systems (WMSs), and the rapid increase in the use of task-based applications across the computational sciences has spurred further research in the area.

Consistent and reliable benchmarking is fundamental to evaluating advances within a field over time. Benchmarks and other performance evaluation systems offer a common ground and objective metrics that enable researchers to assess the efficiency, performance, scalability, and robustness of their solutions under controlled conditions. Benchmarks foster transparency and reproducibility, ensuring that results can be consistently replicated and verified by others in the field. This, in turn, accelerates the pace of innovation as researchers can identify best practices, optimize existing methods, and uncover new areas for improvement. Benchmarks facilitate meaningful comparisons between competing approaches—a valuable aspect for researchers, reviewers, and readers alike.

Access to open source benchmarks democratizes research, and many fields have found great success through the creation of standards. LINPACK [92], for example, is used to evaluate the floating point performance of hardware systems. The Transaction Processing Performance Council (TPC) [236] provides a variety of standard benchmarks for database systems, and UnixBench [241] can evaluate basic performance of Unix-like systems from file copies to system call overheads. Machine learning (ML) has demonstrated this success with

benchmarks for every level of the ML stack. For example, MLPerf [173, 217] has continued to support the development of ML hardware and frameworks, and novel algorithms are compared against prior work by using open source datasets, as exemplified by the Papers with Code Leaderboards [192] that comprise results of tens of thousands of papers across thousands of datasets.

However, the parallel application and workflows communities lack such established benchmarks. The NAS parallel benchmarks date back to the 1990s [29]. For workflows, with the exception of a few common applications (e.g., Montage [181, 84]), papers typically evaluate their solutions on purpose-built synthetic benchmarks or forks of real world science applications. Unfortunately, the *ad hoc* nature of these solutions means that the code is often not open sourced, not maintained beyond publication of the corresponding paper, or so specific to an implementation that it is challenging to appropriately compare against in later works. Recent work has introduced a standard for recording execution traces and tools for analyzing those traces [69], but there remains a need for realistic reference applications for benchmarking.

To address these challenges, we introduce TaPS, the Task Performance Suite, a standardized framework for evaluating task-based execution frameworks against synthetic and realistic science applications. With TaPS, applications can be written in a framework-agnostic manner, thus turning them into suitable benchmarking workloads. Then, the performance of task executors and data management systems can be compared using these applications. We make the following contributions:

1. TaPS, a standardized benchmarking framework for task-based applications with an extensible plugin system for comparing task executors and data management systems. TaPS is available on GitHub [234].

2. Support for popular task executors (Dask, Globus Compute, Parsl, Ray, and TaskVine) and data management systems (shared file systems and ProxyStore, introduced in

Chapter 3).

3. Reference implementations within TAPS for seven real (Cholesky factorization, protein docking, federated learning, MapReduce, molecular design, Montage, and physics simulation) and two synthetic applications.

4. Insights into the performance of the reference implementations across the supported task executor and data management systems.

The rest of this chapter is as follows: Section 2.1 discusses related work; Section 2.2 describes the design and implementation details of the TAPS framework; Section 2.3 introduces the initial set of applications provided by TAPS; Section 2.4 presents our experiences using TAPS to evaluate system components; and Section 2.5 summarizes our contributions and future directions.

2.1 Background and Related Work

Task executors, which manage the execution of tasks in parallel across distributed resources, come in many forms (see Figure 2.1). A *task* refers to discrete unit of work, and tasks are combined into a larger *application*. Tasks can take data as input, produce output data, and may have dependencies with other tasks; i.e., a dependent task cannot start until a preceding tasks completes. Dask Distributed, Python's `ProcessPoolExecutor`, Globus Compute [61], Radical Pilot [16], and Ray all provide mechanisms for executing tasks in parallel across distributed systems.

Workflow management systems (WMSs), a subset of task executors, are designed to define, manage, and execute workflows represented by a directed acyclic graph (DAG) of tasks. WMSs commonly provide mechanisms for automating and optimizing task flow, monitoring, and resource management. WMSs can be categorized as supporting explicit or implicit dataflow patterns. Explicit systems, such as Apache Airflow [22], Fireworks [141],

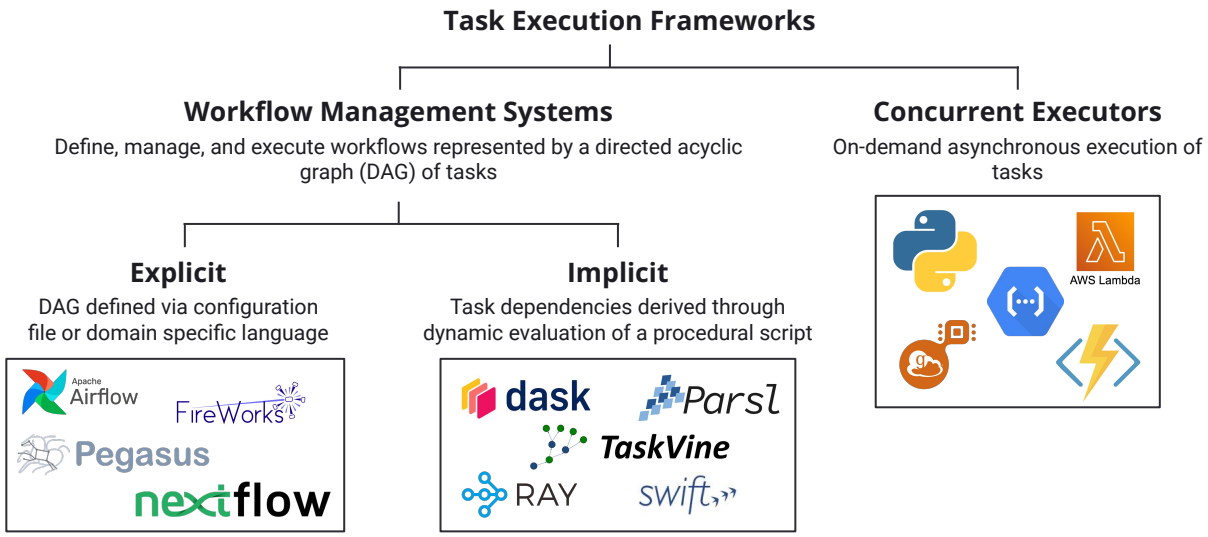


Figure 2.1: A simple taxonomy of task execution frameworks.

Makeflow [13], Nextflow [89], Pegasus [86], and Swift [256], rely on configuration files or domain specific languages (DSLs) to statically define a DAG before execution. Implicit systems, such as Dask Delayed, Parsl, Swift/T [258], and TaskVine [228], derive the application's dataflow through the dynamic evaluation of a procedural script.

Performance evaluation of task executors is challenging due to a lack of standards. Frameworks provide examples designed to aid in learning the framework, but these are often too trivial to be used in benchmarking. Pegasus provides a catalogue of real, end-to-end scientific workflows in AI, astronomy, and bio-informatics which are suitable for benchmarking [200]; Dask maintains a repository of performance benchmarks [79]; WorkflowHub provides a service for sharing scientific workflows [119]; and Workbench [13], designed for analyzing workflow patterns, was released alongside Makeflow. However, these reference applications and benchmarks are typically valid only for evaluating optimizations within the framework they were implemented in. In other words, the majority of these code bases are not suitable for comparing different task executors. This also means available benchmarks are susceptible to code rot if maintenance of the associated framework ceases.

Porting benchmark applications between frameworks is onerous when the structure and

syntax is completely different. Subtle errors in the ported implementation can lead to inaccurate comparisons between systems. Access to datasets or sufficient compute resources for certain applications can further hinder the creation of realistic benchmarking applications. To assuage these challenges within the workflows community, prior work [101] published a gallery of execution traces from real workloads using Pegasus, a synthetic workflow generator, and a simulator framework. WfCommons [69] introduces a standardized format for representing execution logs (WfFormat), an open source package for analyzing logs and generating synthetic logs (WfGen), and a workflow execution simulator (WfSim). WfCommons currently provides 180 execution instances from three workflow systems (Makeflow, Nextflow, and Pegasus). Similarly, WRENCH [55] provides a WMS simulation framework built on SimGrid [54]. In contrast, an Application Skeleton supports the design and development of systems by mimicking the performance of a real application [145].

FunctionBench [149], FaaSDom [169], PanOpticon [230], SeBS [71], and more [28, 264, 242] address a similar set of challenges as TaPS but in the context of cloud-hosted function-as-a-service (FaaS) platforms. In contrast, Das et al. [78] consider benchmarking serverless edge computing platforms. Other works investigate serverless workflow services [254], including SeBS-Flow [223] which extends SeBS to support serverless workflows (released after the publication of TaPS). SeBS provides a benchmark specification, a general model of FaaS platforms, and an implementation of the framework and benchmarks. This model is valuable because each benchmark is platform agnostic, relying only on the abstract FaaS model provided by SeBS. Implementing the concrete model for a new platform need only be performed once, and then any benchmark can be executed on that platform. Part of SeBS's platform model is support for persistent and ephemeral cloud storage systems. Supporting the evaluation of the compute and data aspects of task-based applications is crucial, but currently lacking outside of specific areas (i.e., SeBS for FaaS).

2.2 Design and Implementation

TaPS is a Python package that provides a common framework for writing task-based, distributed applications; a plugin system for running applications with arbitrary task executors and data management systems; and a benchmarking framework for running experiments in a reproducible manner. We choose Python for its pervasiveness in task-based, distributed applications, and we describe here the high level concepts that make the framework possible and the implementation details. Our goal is to create an easy-to-use framework for researchers to benchmark novel systems and an extensible framework so future applications and plugins can be incorporated into TaPS. The examples provided in this section are based on TaPS v0.2.2.

2.2.1 Application Model

TaPS provides a framework for the creation and execution of application benchmarks. As described in Section 2.1, applications are composed of tasks which are the remote execution of a function which takes in some data and produces some data. Tasks can have dependencies such that the result of one task is consumed by one or more tasks.

Supporting applications written using the explicit and implicit workflow models described in Section 2.1 is challenging because the two philosophies are fundamentally at odds with each other and, within the scope of explicit systems, the different configuration formats and use of DSLs further complicates the design of a unified, abstract task executor interface.

TaPS supports writing applications as Python code using implicit dataflow dependencies. (Though, it is not a requirement that tasks have dataflow dependencies within an application.) We take this approach for two reasons. First, the scope of applications compatible with implicit models is a super-set of those compatible with explicit models. Specifically, WFMs which use a static graph for execution are not expressive enough for writing more dynamic and procedural applications, whereas the implicit model enables arbitrarily com-

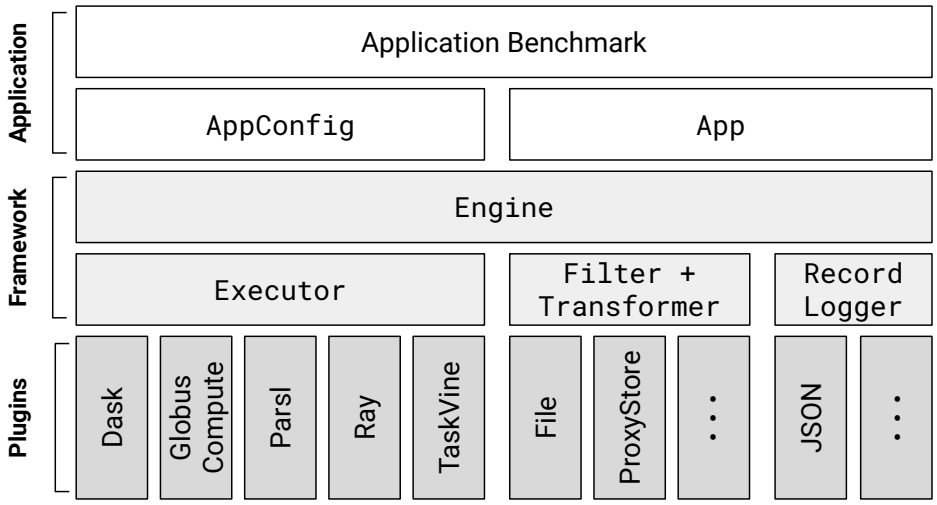


Figure 2.2: Overview of the TaPS stack. Applications are the benchmarking workloads and plugins are the systems being benchmarked. The framework layer enables any application to be run using any set of plugins.

plex applications composed through a procedural program. Second, WMFs which use DSLs require the application design to be tightly coupled to the WMF. This inherently makes it challenging to construct an application that is compatible with a multitude of frameworks.

2.2.2 Writing Benchmark Applications

In TaPS, an application is composed of two parts: an `AppConfig` and an `App` class (see Figure 2.2). The `AppConfig` contains all configuration options required to execute the corresponding applications (e.g., hyperparameters, paths to datasets, or flags). `AppConfig` exposes a `get_app()` method which initializes an `App` instance from the user-specified configuration. `App.run()` is the entry point to the application code and is invoked with two arguments: an `Engine` instance (discussed in detail in Section 2.2.4) and the path to a unique directory for the current application invocation. The `run()` method can contain arbitrary code, provided application tasks are executed via the provided `Engine` interface.

TaPS provides a CLI framework for executing application benchmarks (discussed further in Section 2.2.3). For example, the `foo` application in Figure 2.3 is started with: `python -m`

```python
@register('app')
class FooAppConfig(AppConfig):
    name: str = 'foo'
    sleep: float = Field(description='...')
    count: int = Field(1, description='...')

    def get_app(self) -> FooApp: ...

class FooApp:
    def __init__(self, ...) -> None: ...

    def run(self, engine: Engine, run_dir: Path) -> None: ...

    def close(self) -> None: ...
```

Figure 2.3: Applications in TAPS are defined via an `App` and `AppConfig` class. The configuration, provided by the user, is used to instantiate an instance of the app. Command line arguments are created using the `AppConfig`.

`taps.run --app foo {args}`. Applications are registered with this CLI by decorating the `AppConfig` with `@register('app')`. This will automatically add the application's name as one of the CLI choices and add CLI arguments based on the `AppConfig` attributes.

2.2.3 Configuring and Executing Benchmarks

Application benchmarks are typically invoked using the CLI. At minimum, this requires specifying the `--app` parameter and any required arguments specific to the chosen applications. The following example command executes the `cholesky` application, described in Section 2.3.

```
> python -m taps.run --app cholesky --app.matrix_size 100 --app.block_size 25
```

When invoked, the CLI (1) constructs an `AppConfig` instance from the user's arguments, validating that options can be parsed into the correct type and that all required arguments are present; (2) initializes the `App` using `get_app()`; (3) constructs an `Engine` according to user-supplied arguments; and (4) invokes `App.run()` to execute the application benchmark. The framework automatically writes a configuration file, log files, and task record files to the

```toml
[app]
name = "cholesky"
matrix_size = 100
block_size = 25

[engine]
task_record_file_name = "tasks.jsonl"

[engine.executor]
name = "process-pool"
max_processes = 10

[run]
dir_format = "runs/{name}_{executor}_{timestamp}"

[logging]
file_level = "WARNING"
file_name = "log.txt"
level = "INFO"
```

Figure 2.4: Configuration files in TaPS use the TOML format. This example file runs the `cholesky` application with a process pool task executor.

run directory.

The configuration file contains a record of all configuration options used to execute the application. An example of the TOML configuration file is presented in Figure 2.4. A configuration file path can be provided to the CLI as an alternative to CLI arguments:

```
> python -m taps.run --config config.toml
```

Thus, configuration files can be shared for reproducibility. If multiple configuration files are provided, keys will be merged with later files taking precedence. CLI arguments supersede all configuration file options.

More sophisticated benchmarking can be performed by writing custom Python scripts that use the TaPS API directly rather than via the TaPS CLI, as exemplified in Figure 2.5.

```python
import contextlib
import pathlib
from concurrent.futures import ProcessPoolExecutor
from datetime import datetime

from taps.apps.cholesky import CholeskyApp
from taps.engine import Engine
from taps.executor.utils import FutureDependencyExecutor
from taps.logging import init_logging

def main() -> int:
    init_logging()

    app = CholeskyApp(matrix_size=100, block_size=25)
    executor = FutureDependencyExecutor(
        ProcessPoolExecutor(max_workers=4)
    )
    timestamp = datetime.now().strftime('%Y-%m-%d-%H-%M-%S')
    run_dir = pathlib.Path.cwd() / 'runs' / timestamp

    with contextlib.closing(app), Engine(executor) as engine:
        app.run(engine, run_dir)

    return 0

if __name__ == '__main__':
    raise SystemExit(main())
```

Figure 2.5: Example usage of the TaPS API as an alternative to the CLI for running benchmarks. Here, the `cholesky` application is executed using a `ProcessPoolExecutor`. Use of the API can enable more sophisticated benchmarking, such as to create parameter matrices.

2.2.4 Application Benchmark Engine

The `Engine` is the unified interface used by applications to execute tasks and exposes an interface similar to Python's `concurrent.futures.Executor`. The `Engine` interface must be expressive enough to build arbitrary applications yet simple enough to incorporate third-party task executors and other plugins. We chose to adopt a model similar to Python's `Executor` because it is a *de facto* standard for managing asynchronous task execution across the Python ecosystem, and many third-party libraries provide `Executor`-like implementations, including Dask Distributed, Globus Compute, Loky, TaskVine, and Parsl. An additional benefit of this choice is that it is trivial to port applications already using an `Executor` interface into a TAPS application.

`Executor` is an abstract class with two primary methods, `submit()` and `map()`, designed to execute functions asynchronously. The `submit()` method takes a callable object and associated arguments, schedules the callable for execution, and returns back to the client a `Future` that will eventually contain the result of the callable. `Engine` implements both of these methods, but returns `TaskFuture` objects rather than `Future` instances. Functionally, `TaskFuture` behaves like `Future` but includes additional functionality for performance monitoring and task dependency management.

An `Engine` is created from four components: `Executor`, `Transformer`, `Filter`, and `RecordLogger`. This conceptual hierarchy of components in TAPS is illustrated in Figure 2.2. The dependency model approach used by the `Engine` means that component plugins can be easily created and/or swapped to compare, for example, different task executors or data management systems. Further, the `Engine` can be extended with additional components in the future to enhance benchmarking capabilities.

Table 2.1: Overview of the task executors supported natively within TAPS.

			Scheduler			Deployment	
Name	Reference	Languages	Distributed	Dataflow	Locality	Distributed	Batch Systems
`ThreadPoolExecutor`	[209]	Python					
`ProcessPoolExecutor`	[209]	Python					
Dask Distributed	[219]	Python		✓	✓	✓	✓
Globus Compute	[61]	Python				✓	✓
Parsl	[27]	Python		✓		✓	✓
Ray	[184]	Multiple	✓	✓	✓	✓	✓
TaskVine	[228]	C, Python		✓	✓	✓	✓

2.2.5 Task Executor Model

The fundamental component of the `Engine` is an `Executor`, an interface to the underlying task executor. We choose the `Executor` model again for the same reasons as with the `Engine`. In Section 2.2.6, we describe the details of each executor currently supported in TAPS. Similar to the `App` model, TAPS has a notion of a `ExecutorConfig` which can be registered with the framework to automatically add argument parser groups for the specific executor. `ExecutorConfig` has a method, `get_executor()`, which will initialize an instance of the executor from the user specified configuration.

A limitation of Python's `Executor` interface is the lack of support for dataflow dependencies between tasks. Some `Executor` implementations (Dask Distributed, Parsl, and TaskVine) do support implicit dataflow dependencies by passing the future of one task as input to one or more tasks, but many others (e.g., Python's `ProcessPoolExecutor` and Globus Compute) do not. The `Engine` requires it's `Executor` to support implicit dataflow patterns with futures, so TAPS provides a `FutureDependencyExecutor` wrapper to add this functionality if needed. This wrapper scans task inputs for futures and will delay submission of a task until the results of all input futures are available (in an asynchronous, non-blocking manner).

2.2.6 Supported Task Executors

Here, we briefly describe the task executors currently supported by TaPS (summarized in Table 2.1). As previously mentioned, the plugin system makes it easy to support more executors in the future, but our initial goal is to support a broad range. We support Python's `ProcessPoolExecutor` which provides a good baseline for low-overhead, single-node execution. We also support the `ThreadPoolExecutor`, but this is primarily intended to support development and quick testing because Python's Global Interpreter Lock prevents true parallelism with threading.

Dask Distributed [219] provides dynamic task scheduling and management across worker processes distributed across cores within a node or across several nodes. Tasks in Dask are Python functions which operate on Python objects; the scheduler tracks these task in a dynamic DAG. Globus Compute [61] is a cloud-managed function-as-a-service (FaaS) platform which can execute Python functions across federated compute systems. Globus Compute provides an `Executor` interface but does not manage dependencies between functions. Parsl [27] is a parallel programming library for Python with comprehensive dataflow management capabilities. Parsl supports many execution models including local compute, remote compute, and batch scheduling systems. Ray [184] is a general purpose framework for executing task-parallel and actor-based computations on distributed systems in a scalable and fault tolerant manner. TaskVine [228] executes dynamic DAG workflows with a focus on data management features including transformation, distribution, and task data locality.

2.2.7 Task Data Model

Optimizing the transfer of task data and placement of tasks according to where data reside is a core feature of many task executors. To support further research into data management, TaPS supports a plugin system for *data transformers*. A data transformer is an object that implements the `Transformer` protocol. This protocol defines two methods: `transform` which

```
[engine.filter]
name = "object-size"
min_size = 1000
max_size = 1000000

[engine.transformer]
name = "proxystore"

[engine.transformer.connector]
kind = "redis"
options = { hostname = "localhost", port = 6379}
```

Figure 2.6: Example TaPS configuration that enables the `ProxyTransformer` using ProxyStore's `RedisConnector` for objects between 1 kB and 1 MB.

takes an object and returns an identifier, and `resolve`, the inverse of `transform`, which takes an identifier and returns the corresponding object. Data transformer implementations can implement object identifiers in any manner, provided identifier instances are serializable. For example, an identifier could simply be a UUID corresponding to a database entry containing the serialized object.

A `Filter` is a callable object, e.g., a function, that takes an object as input and returns a boolean indicating if the object should be transformed by the data transformer. The `Engine` uses the `Transformer` and `Filter` to transform the positional arguments, keyword arguments, and results of tasks before being sent to the `Executor`. For example, every argument in the positional arguments tuple which passes the filter check is transformed into an identifier using the data transformer. Each task is encapsulated with a wrapper which will *resolve* any arguments that were replaced with identifiers when the task executes. The same occurs in reverse for a task's result.

`Filter` implementations based on object size, pickled object size, and object type are provided. We initially provide two `Transformer` implementations: `PickleFileTransformer` and `ProxyTransformer`. The `PickleFileTransformer` pickles objects and writes the pickled data to a file. The `ProxyTransformer` creates proxies of objects using the ProxyStore library, introduced in Chapter 3. ProxyStore provides a pass-by-reference like model for

distributed Python applications and supports a multitude of communication protocols including DAOS [131], Globus [106, 15], Margo [206], Redis [218], UCX [240], and ZeroMQ [135]. An example configuration for a filter and transformer are provided in Figure 2.6.

2.2.8 Logging and Metrics

Recording logs and metrics for post-execution analysis is core to any benchmarking framework. TaPS records the high-level application logs and low-level details of each executed task. The `RecordLogger` interface is used to log records of all tasks executed by the `Engine`. These records include metrics and metadata of the task, such as the unique task ID, the function name, task IDs of any parent tasks, submission time, completion time, data transformation and resolution times, and execution makespan. By default, TaPS uses the `JSONRecordLogger` which logs a JSON representation of the task information to a line-delimited file. In future work, we would also like to support WfCommon's WfFormat.

2.2.9 Task Life-cycle

An application creates a task by submitting a Python function with corresponding arguments to `Engine.submit()` which returns a corresponding `TaskFuture`. (Applications can also create many tasks by mapping a function onto an iterable of arguments via `Engine.map()`. For simplicity, we discuss single task submission here, but the same process applies with map.) The `Engine` generates a unique ID for the task and wraps the function in a task wrapper. The `Transformer` is then applied to the arguments according to the `Filter`. Then, the wrapped function and arguments (some or all of which may have been transformed) are passed to the `Executor` for scheduling and execution. The `Executor` returns a future specific to the executor type (e.g., a Globus Compute future for a `GlobusComputeExecutor`). This low-level future is then wrapped in a `TaskFuture`, and the `TaskFuture` is returned to the client. If a `TaskFuture` were passed as input to a task, the `Engine` will also replace the

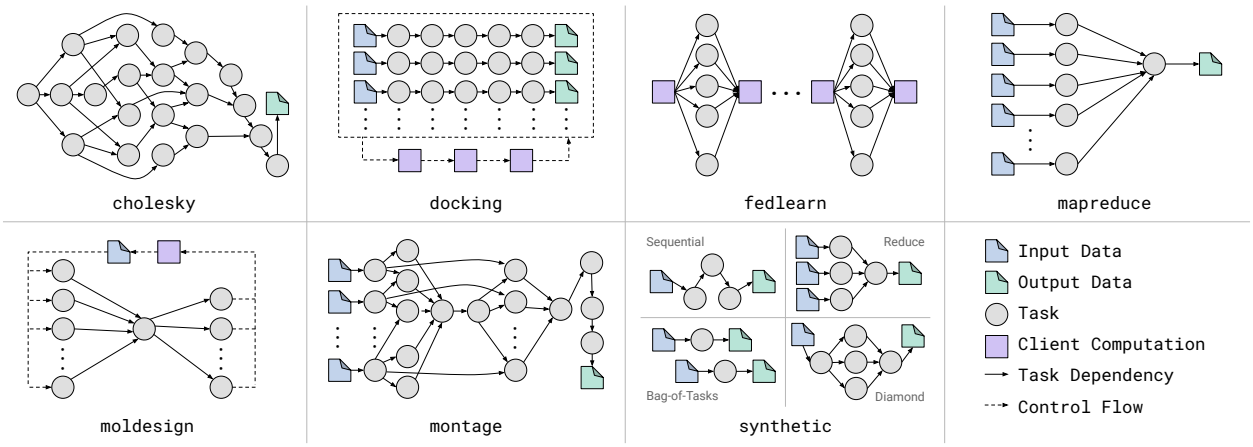


Figure 2.7: Task dependency diagrams for some of the provided applications within TaPS. The initial set of applications is designed to cover a wide range of patterns. Exact task graphs depend on the application configuration.

`TaskFuture` with the low-level future of the `Executor`. This is necessary to ensuring the `Executor` can schedule the tasks according to the implicit inter-task dependencies.

When a task begins execution, the task wrapper will record information about the execution to propagate back to the `Engine`. The task wrapper will also resolve any transformed arguments prior to invoking the original function provided by the client and possibly transform the function result. The completion of a task (i.e., when the result of the future is set) will trigger a callback which logs all of a task's information and metrics. If the function result was transformed, the `TaskFuture` will resolve the result inside of `TaskFuture.result()`.

2.3 Applications

We initially provide nine applications within TaPS, summarized in Table 2.2 and Figure 2.7. These distributed and parallel applications are diverse, spanning many domains, datasets, and structures to support comprehensive performance evaluation of existing and future systems.

Table 2.2: Overview of the applications implemented within TAPS.

Name	Reference	Domain	Dataset(s)
`cholesky`	[142]	Linear Algebra	Randomly Generated
`docking`	[213]	Drug Discovery	C-ABL Kinase Domain [25], Zinc Ord. Compounds [68]
`fedlearn`	[151]	Machine Learning	MNIST [87], FEMNIST [261], CIFAR-10/100 [152]
`mapreduce`	[83]	Text Analysis	Randomly Generated, Enron Corpus [98]
`moldesign`	[250]	Molecular Design	QM9 [214]
`montage`	[181]	Astronomy	Montage Images [181]
`physics`	—	Physics	Randomly Generated
`failures`	[266]	—	—
`synthetic`	[195]	—	Randomly Generated

Name	Task Types(s)	Data Format(s)
`cholesky`	Python Functions	In-memory
`docking`	Executable, Python Functions	File
`fedlearn`	Python Functions	In-memory
`mapreduce`	Python Functions	File, In-memory
`moldesign`	Python Functions	In-memory
`montage`	Executable	File
`physics`	Python Functions	In-memory
`failures`	Executable, Python Functions	File, In-memory
`synthetic`	Python Functions	In-memory

2.3.1 Cholesky Factorization

Cholesky factorization (also referred to as decomposition) is a fundamental linear algebra operation used in many domains. The tiled version of Cholesky factorization has been studied extensively, for example, in the context of NUMA machines [142] and from the perspective of communication overhead [33]. The tiled version produces an arbitrarily complex DAG depending on the number of tiles, which makes it a good candidate for evaluating task executors. The 4×4 tiled DAG is portrayed in Figure 2.7.

The `cholesky` application implements a tiled Cholesky factorization which, given an input matrix A that is positive-definite, computes L where $A = L \times L^T$ [142]. The algorithm comprises four task types: GEMM, a tiled matrix multiplication requiring three inputs; SYRK, a symmetric rank-k update requiring three inputs; TRSM, which solves a triangular matrix equation with two inputs; and POTRF, an untiled Cholesky factorization which

operates on a tile of A.

The `cholesky` application takes two user-supplied parameters: N, the side length of the input matrix to generate, and b, the side length of each square block in the tiled matrix. As b approaches N, the number of blocks in the tiled matrix, and thus the number of tasks required for the factorization, decreases. Given B, a randomly generated $N \times N$ matrix, the positive definite input matrix A is computed by using $A = (B + B^T) + \delta I$, where $\delta = N$ and I is the $N \times N$ identity matrix.

2.3.2 Protein Docking

Protein docking aims to predict the orientation and position of one molecule to another. It is commonly used in structure-based drug design as it helps predict the binding affinity of a ligand (the candidate drug) to the target receptor. Simulations required to compute docking score are computationally expensive, and the search-space of potential molecules can be expansive. To improve the time-to-solution, this implementation of protein docking is parallelized and includes ML-in-the-loop. A model is trained using the results of previous simulations to predict which molecules are most likely to have strong binding scores, thereby significantly reducing the search space.

The `docking` workflow is based on a reference implementation written in Parsl [213]. The workflow uses Autodock Vina [238] for the docking simulations and scikit-learn [199] to construct a KNN-based transformer for the ML model. It is composed of three task types: (1) data preparation, (2) simulation, and (3) ML training and inference. The workflow has two primary parameters: a CSV file containing the search space of candidate ligands and their associated SMILES strings and a PDBQT file containing the target receptor. One of the tasks launches a subprocess to execute a `set-element.tcl` script (provided in the reference implementation) that adds coordinates to the PDB file using VMD [139], a program used to display and analyse molecular assemblies.

2.3.3 Federated Learning

Federated Learning (FL) is a paradigm for deep learning across decentralized devices with their own private data. FL offloads the task of model training to the decentralized devices to avoid communicating their raw training data across the network, providing some level of privacy and reducing data transfer costs. FL is organized into multiple rounds. In each round, a central server is responsible for collecting locally-updated model parameters from each device and aggregating the parameters to produce/update a global model. The new global model is then redistributed to the decentralized devices for further training and the loop repeats for future rounds [174].

We implement a simple FL application, `fedlearn`, that simulates a decentralized system with varying number of simulated devices and data distributions. `Fedlearn` follows the flow of execution described above and consists of three tasks: local training, model aggregation, and global model testing. The first task emulates the local training that is performed on a simulated remote device. The second task takes the returned locally-trained models for a given round as input to perform a model aggregation step to update the global model. The third task takes the recently-updated global model and evaluates it using a test dataset that was not used during training. All tasks are implemented as pure Python functions with model training and evaluation performed using PyTorch [194].

The application can be tuned in several ways, including, but not limited to, the total number of aggregation rounds, the number of simulated devices, the distribution of data samples across the simulated devices via the Dirichlet distribution, training hyperparameters (e.g., epochs, learning rate, minibatch size), and fraction of devices randomly sampled to participate in each round. The application supports four standard deep learning datasets (MNIST [87], Fashion-MNIST [261], CIFAR-10, CIFAR-100 [152]), each of which is split into disjoint subsets across each simulated device for local training. A multi-layer perceptron network with three layers and ReLU activations is used with MNIST and Fashion-MNIST,

and a small convolutional neural network with ReLU activations is used with CIFAR-10 and CIFAR-100.

2.3.4 MapReduce

MapReduce [83] is a programming model for parallel big data processing comprised of two tasks types. Map tasks filter or sort input data, and a reduce task performs a summation operation on the map outputs. The canonical example for MapReduce is computing words counts in a text corpus. Here, the map tasks take a subset of documents in the corpus as input and count each word in the subset. The subset counts are then summed by the reduce task.

The `mapreduce` application implements this word frequency example. The goal of this application is to evaluate system responsiveness when processing large datasets. The implementation can operate in two modes, one in which a text corpus of arbitrary, user-defined size is generated, and another in which user-provided text files can be read. For a real dataset, we use the publicly available Enron email dataset [98]. Beyond specifying the input corpus or parameters of the randomly generated corpus, the number of mapping tasks and n, the number of most frequent words to save, are configurable.

The map task, implemented in Python, takes as input either a string of text or a list of files to read the text string from and returns a `collections.Counter` object containing the frequencies of each word. The reduce task takes a list of `Counter` objects and returns a single `Counter`. The application produces an output file containing the n most frequent words and their frequencies.

2.3.5 Molecular Design

Molecules with high ionization potentials (IP) are important for the design of next-generation redox-flow batteries [249, 251]. Active learning, a process where a surrogate ML model is used

to determine which simulations to perform based on previous computations, is commonly employed to efficiently discover high-performing molecules.

The `moldesign` application is based on a Parsl implementation of ML-guided molecular design [250]. The application has three task types. Simulation tasks compute a molecule's IP, training tasks retrain an ML model based on the results of simulation tasks, and inference tasks use the ML model to predict which molecules will have high IPs and should be simulated. This application is highly dynamic and does not have strong inter-task dependencies—the client processes task results to determine which new tasks should be submitted. Molecules are sampled from the open-source QM9 dataset [214]. The number of initial simulations to perform, simulation batch size, and number of molecules to evaluate in total are configurable. These parameters control the maximum parallelism of the application and the length of the campaign.

2.3.6 Montage

Montage is a toolkit for creating mosaics from astronomical images [36]. The Montage Mosaic workflow streamlines the creation of such mosaics by invoking a series of Montage tools on the provided input data. This workflow was adapted from Montage's "Getting Started" tutorial [182].

The `montage` application is executed using a directory of input images and parameters for the table and header file names. The 2MASS input images are made available by Montage [3]. The application consists of a series of image processing tasks that will (1) reproject the images, (2) update metadata, (3) remove overlaps, and (4) combine images into a mosaic. Parallelism within the workflow occurs during the reprojection of the images, removing overlaps between two images, and removing the background in each input image. Tasks read and write intermediate files so all workers require access to a shared file system.

2.3.7 Physics Simulation

The `physics` application simulates the trajectory of objects in a randomly generated environment (e.g., a ball hitting a surface, bouncing, and then rolling to a stop). Physics simulations are often performed in an embarrassingly parallel fashion, where each simulation is performed with different initial conditions, and the results aggregated. This application is a surrogate for this style of workflow, and `PyBullet` [207] is used as the simulation engine.

2.3.8 Failure Injection

The `failures` application can inject failures into another TaPS application. Injecting failures enables analyzing the failure recovery characteristics of executors. Task-level failure types include runtime exceptions (e.g, divide-by-zero, import error, out-of-memory, open file limit (ulimit) exceeded, and walltime exceeded) and dependency errors from a failed parent task. System-level failures include task worker, worker manager, and node failures. The failure type, failure rate, and base application to inject failures into are configurable.

2.3.9 Synthetic Workflow

The `synthetic` application is used to create synthetic computational workflows and is useful for stress testing systems. Tasks in this application are no-op sleep tasks which take in some random data and, optionally, produce some random data. One of four structures for the workflow DAG can be chosen: sequential, reduce, bag-of-tasks, and diamond, as described in Figure 2.7. The number of tasks, input and output data sizes, and sleep times are all configurable.

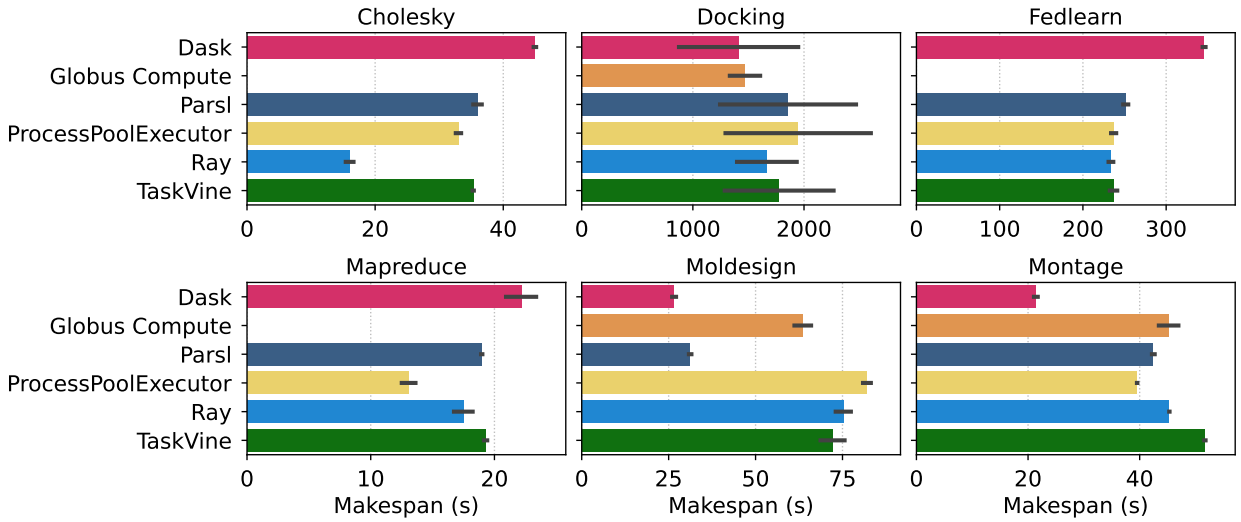


Figure 2.8: Average makespan across three runs for six of the TaPS applications. Error bars denote standard deviation.

2.4 Evaluation

We showcase the kinds of performance evaluations possible with TaPS using the provided applications. We draw some general conclusions but do not make an exhaustive comparison between executors. Rather, we aim to demonstrate the varied performance characteristics of our supported applications and plugins, highlight the kinds of investigations or analyses that can be performed with TaPS, and pose interesting questions for future investigations. We use a `compute-zen-3` node, with two 64-core CPUs and 256 GB memory, on Chameleon Cloud's CHI@TACC cluster for evaluation [146].

2.4.1 Application Makespan

We first compare application makespan, which includes executor and worker initialization, application execution, and shutdown, across each task executor. The space of possible configurations for each application and executor is combinatorially explosive. Thus, we choose application parameters, where possible, which result in high numbers of short tasks to accentuate the effects of overheads in the respective executors. Parameters are summarized

Table 2.3: Summary of application configurations used in Figure 2.8.

Application	Workers	Task Count	Max Serialized Object Size
`cholesky`	64	385	24 MB
`docking`	32	192	$O(1)$ kB
`fedlearn`	32	48	20 MB
`mapreduce`	32	33	114 MB
`moldesign`	32	346	$O(1)$ MB
`montage`	32	419	$O(1)$ kB

Application	Parameters
`cholesky`	Matrix Size: 10 000×10 000, Block Size: 1000×1000
`docking`	Initial Simulations: 3, Batch Size: 8, Rounds: 3
`fedlearn`	Dataset: MNIST, Clients: 16, Batch Size: 32, Rounds: 3, Epochs/Round: 1
`mapreduce`	Dataset: Enron Email Corpus, Map Task Count: 32
`moldesign`	Initial Simulations: 16, Batch Size: 16, Search Count: 64
`montage`	—

in Table 2.3. We also prefer configurations which reduce run-to-run variances, except for `docking` which is inherently stochastic (this was changed in a later release of TaPS). For each executor, we use the respective equivalent of a default local/single-node deployment, but we note that it is reasonable to expect performance improvements by tuning each executor deployment to the specific application and hardware.

The results, presented in Figure 2.8, indicate that no executor is optimal and lead us to ask further questions. Why are the following 2–3× faster than the others: Ray in `cholesky`, Dask and Parsl in `moldesign`, and Dask in `montage`? How does performance correlate to average task duration or data flow volume? How do different executors deal with nested parallelism (i.e., tasks which invoke multi-threaded code)?

We observe that Dask performs the best in applications with small maximum object sizes, such as `docking`, `moldesign`, and `montage` where, as shown in Table 2.3, the maximum serialized object sizes are less than ∼1 MB. However, Dask is slow with applications that embed large objects in the task graph, such as the 114 MB mapper outputs in `mapreduce`. Ray marks input arrays as immutable enabling optimizations which yield considerable speedups

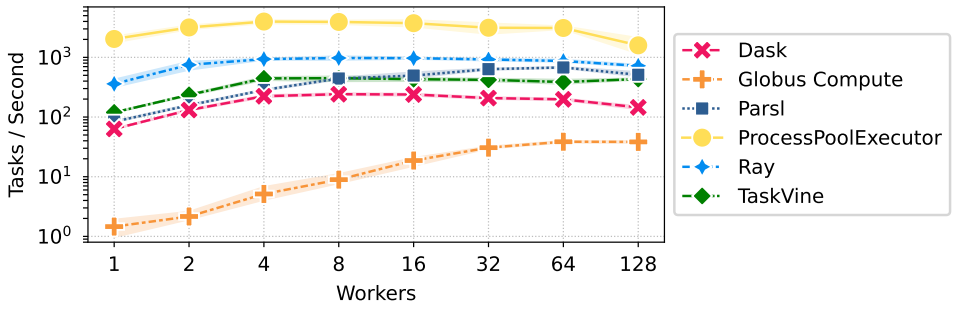


Figure 2.9: Executor scaling performance with no-op tasks. Each configuration is repeated three times and shaded regions represent the standard deviation.

in `cholesky`. Applications with nested parallelism (the simulation codes in `docking` and `moldesign` and tensor operations in `fedlearn`) lead to different outcomes. Globus Compute, Parsl, and `ProcessPoolExecutor` required setting `OMP_NUM_THREADS=1` to prevent resource contention leading to applications hanging, whereas Dask, Ray, and TaskVine worked immediately with all task types, albeit with varied performance. The Globus Compute service limits task payloads to 10 MB so the `cholesky`, `fedlearn`, and `mapreduce` applications are not natively supported and necessitate alternative data management systems (discussed further in Section 2.4.3).

2.4.2 Scaling Performance

We evaluate scaling performance of each executor using the `synthetic` app by executing 1000 no-op, no-data tasks and recording the task completion rate as a function of the number of workers on a single node. Here, the client submits n initial tasks where n is the number of workers and submits new tasks as running tasks complete. This configuration is intended to stress-test all aspects of the system including scheduler throughput, worker overheads, and client task result latency. We disable task result caching where applicable.

The results are presented in Figure 2.9. The `ProcessPoolExecutor` performs the best because, unlike the other executors, there is no scheduler. Thus, this serves as a good baseline for this single-node scaling setup; however, the lack of scheduler also means the

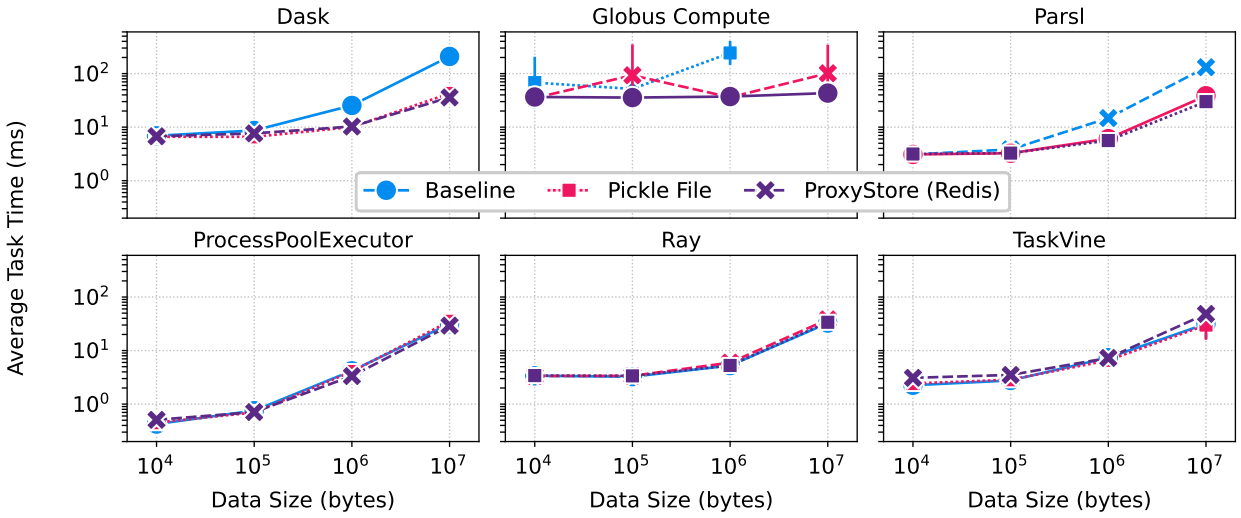


Figure 2.10: Average round-trip time for no-op tasks as a function of input/output data size. Error bars denote standard deviation from three runs of 320 tasks (10 × 32 workers). The Globus Compute baseline is not evaluated at 10 MB due to task payload limits of the Globus Compute service.

`ProcessPoolExecutor` lacks features useful for optimizing real applications such as multi-node support, data-aware task placement, and result caching. The general trend for Dask, Ray, and TaskVine is similar; task throughput increases up to four or eight workers and then degrades at high worker counts. However, Ray and TaskVine are both faster, with Ray being 5–10× faster than Dask. This can, in part, be attributed to Dask being pure Python while TaskVine's core is C and Ray's core is C++. Parsl, which is pure Python, exhibits superior scaling efficiency, closing the performance gap to Ray at larger scales. Globus Compute's task throughput is limited by its cloud service, but we do observe strong scaling performance with more workers as task requests and results can be more efficiently batched which amortizes cloud overheads.

2.4.3 Data Transfer

We examine the effects of data transfer on task latency and evaluate the `Transformer` plugins in Figure 2.10. We submit tasks to a pool of 32 workers and measure the average round-trip

task time using the `synthetic` application. The client generates b bytes of random data as input to the task and the task returns b bytes of random data. We compare the baseline performance of the executors to using two different transformers: `PickleFileTransformer`, which writes pickled task data to the local NVMe drive, and ProxyStore, which we configured to use a Redis server to store intermediate data.

Dask and Parsl exhibit similar behaviour with task payloads greater than 100 kB inducing considerable increases in task latency. Using an alternate mechanism for data transfer alleviates much of this overhead, leading to 5.8× and 4.4× speedups for Dask and Parsl, respectively, at the largest data sizes. Globus Compute benefits the most from alternative data transfer mechanisms such as ProxyStore because the baseline method relies on data transfer to/from the cloud which is considerably slower. Use of ProxyStore also avoids Globus Compute's 10 MB task payload limit. The `ProcessPoolExecutor`, due to its simplicity, does not benefit much from either alternative transfer mechanisms. Ray and TaskVine perform well in all scenarios because Ray uses a distributed object store for large task data and TaskVine communicates intermediate data by files. Thus, these systems already employ techniques similar to the data transformers we evaluated.

2.5 Summary

We have proposed TaPS, a performance evaluation platform for task-based execution frameworks. TaPS aims to provide a standard system for benchmarking frameworks. Benchmarking applications can be written in a framework agnostic manner then evaluated using TaPS' extensive plugin system. TaPS provides many reference applications, a diverse set of supported task executors and data management systems, and performance and metadata logging. We then showcased TaPS through a survey of evaluations to understand performance characteristics of the applications and executors, such as task overheads, data management, and scalability. Our hope is that TaPS will be a long-standing tool used to provide a

common ground for evaluation and to facilitate the advancement in the state-of-the-art for parallel application execution.

CHAPTER 3

A NOVEL WIDE-AREA DATA MANAGEMENT PARADIGM

The function-as-a-service (FaaS) and workflow programming paradigms facilitate the development of scalable distributed applications. Programmers specify *what* task (e.g., function or workflow stage) to perform without regard to *where* they are executed; the FaaS or workflow system then handles the mechanics of routing each task to a suitable processor. FaaS systems often assume that tasks are independent, while in workflow systems tasks may be linked in a dependency graph (e.g., a directed acyclic graph). In both cases, it is common for all data movement to pass via a central location such as a FaaS service, workflow engine, shared file system, or task database, where task inputs and outputs can be stored persistently on stable storage. Such centralized approaches may lead to unnecessary communication [27, 222] but facilitate the implementation of other useful features like re-execution of failed tasks or dynamic adjustments of task location.

The passing of data among tasks via a central location become increasingly problematic when tasks are located on distinct computers. Consider a program that makes a function call `x=f()` to produce a value `x` that is to be consumed by a second function call `g(x)`. If `f()` and `g()` are dispatched to different computers C_a and C_b, respectively, then `x` must be transferred from C_a to C_b. Requiring that this transfer pass via a central location (e.g., FaaS service, workflow controller, shared file system) is inefficient, particularly if `x` is an intermediary value of no use to the client. Instead, it would be preferable to communicate `x` directly from `f()` to `g()`. To do this, we need methods for: (1) representing `x` such that `f()` and `g()` can produce and globally reference `x` and (2) communicating `x` from `f()` to `g()`, despite `f()` and `g()` running in different processes, compute nodes, or systems.

To address these challenges, we present ProxyStore, an abstraction for managing the routing of data between processes in distributed and federated Python applications. ProxyStore allows developers to focus, when writing and composing distributed applications, on

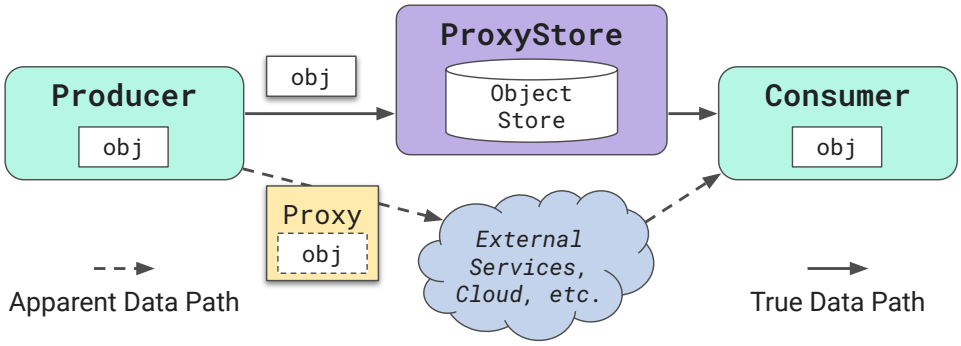


Figure 3.1: ProxyStore decouples the communication of object data from control flow transparently to the application. Data consumers receive lightweight proxies that act like the true object when used, while the heavy lifting of object communication is handled separately.

logical data flow rather than physical details of where data reside and how data are communicated. This decoupling enables the dynamic selection of different data movement methods, depending on *what* data are moved, *where* data are moved, or *when* data are moved—a long-standing challenge in distributed application design [110, 165, 62, 157]. The *proxy* programming model transparently provides pass-by-reference semantics and just-in-time object resolution to consumers. A proxy is lightweight and can be communicated efficiently via any means while its referenced object is communicated transparently via optimal routes. By thus abstracting the use of specialized communication methods, the proxy paradigm improves code compatibility, performance, and productivity.

ProxyStore provides interfaces to common mediated communication channels (e.g., shared file systems, Globus [106, 15, 59], Redis [218]) and custom implementations that leverage the powerful communication technologies of high-performance computing (HPC) environments and enable direct communication between remote systems.

The contributions of this chapter are:

- The design and implementation of the proxy model which is, to the best of our knowledge, the first system that transparently provides both pass-by-reference and pass-by-value semantics for distributed applications.

- Data transfer mechanisms that enable fast intra- and inter-site communication in various settings and an extensible framework for seamless integration of new technologies.
- Component level benchmarks of ProxyStore and comparisons to prior works.
- Experiments using ProxyStore to accelerate real-world federated science applications.
- Case studies demonstrating the integration of ProxyStore into existing distributed computing frameworks.

ProxyStore is an open-source Python package available on GitHub [204] and PyPI [205]. The rest of this chapter is as follows: Section 3.1 explores related work in federated and distributed application design; Section 3.2 outlines ProxyStore design goals and introduces the core components; Section 3.3 describes the communication channels provided; Section 3.4 presents synthetic evaluations and component-level benchmarks; Section 3.5 applies ProxyStore to real-world use cases; Section 3.6 discusses experiences integrating ProxyStore into distributed computing frameworks; and Section 3.7 summarizes our contributions and future plans.

3.1 Background and Related Work

Increasing hardware heterogeneity, faster and more reliable networks, and shifts in application requirements have motivated federated application design, i.e., applications that span several cloud, high-performance (HPC), edge, and personal systems. Here we discuss technologies that enable the management of computation across diverse systems.

Communication decoupling: The appropriate design for a distributed application's communication fabric depends on the decoupling needed among application processes. Eugster et al. [99] describes how decoupling can occur along space, time, and synchronization

dimensions. Processes decoupled in *space* interact indirectly via a shared service (e.g., message queue or object store). A producer and consumer are decoupled in *time* if they need not be active at the same time, entities can go online or offline independently and as needed. Decoupling in *synchronization* means that data production or consumption does not occur in the primary control flow, so that, for example, processes need not block on communication or can be notified asynchronously of events.

HPC applications built on the Message Passage Interface (MPI) often forgo space and time decoupling. This rigid design has allowed for the development of efficient MPI implementations [113, 191], including for heterogeneous systems [107]. Asynchronous operations allow for some overlapping of computation and communication, but in general MPI applications are tightly coupled.

Modern application design, such as those using serverless or workflow systems, favors decoupling across all dimensions for greater flexibility in deployment. Asynchronous systems and on-demand computing necessitate flexibility among the entities involved in distributed applications. Publish/subscribe, message queues, and object stores typically provide space and time decoupling at a minimum and most implementations provide some form of synchronization decoupling (generally producer-side). Subsequently, significant investment has been made into services meeting these requirements (e.g., AWS S3 [43], Azure Storage [50], and KeyDB [229]).

Prior work distinguishes *direct* communication channels from *mediated* channels where "the communication between participants is done over storage or other indirect means" [73]. Direct channels typically provide for rapid communication but prevent space and time decoupling among actors. Mediated channels necessarily provide space decoupling and can also provide time decoupling if the mediator (e.g., storage) persists for the entirety of the period over which any producers or consumers exist.

Data fabrics: Tuple spaces, such as in Linda [8], were early shared data fabrics. In

the tuple-space model, producers post data as tuples in a shared distributed memory space, from which consumers can retrieve data that match a specified pattern. Tuple spaces have since been implemented in many languages, including Python [32]. DataSpaces [90] provides a tuple-space-like interface to a virtual shared object space designed to support large-scale workflows composed of coupled applications. The shared space is implemented with remote procedure calls (RPC) and transfer provided by the Margo and Mercury RPC libraries [220, 225, 231]. WA-DataSpaces [11] extends the DataSpaces model to support data staging and predictive prefetching to improve data access times. The InterPlanetary File System (IPFS) is a decentralized, peer-to-peer file sharing network that provides content-addressing via a flat global namespace [34].

Network policies: Network access is a core problem in federated computing because policies vary between networks. Network address translation (NAT) and firewalls often prohibit outside access to local devices, thus preventing direct communication between hosts. These problems are particularly prevalent in scientific computing where experimental instruments are often in different locations from the data storage and analysis computers. At some sites, Science DMZs [77] permit bypassing firewalls under programmatic control [60]. Cross-site data transfer can be performed via cloud services (e.g., in Globus Compute [61]), but this adds latency and can be cost-prohibitive for data-intensive applications. SciStream [64] addresses these issues by using gateway nodes (e.g., data transfer nodes in a Science DMZ) to facilitate fast memory-to-memory data transfers between remote hosts.

NAT traversal: A general solution for communication between two hosts behind separate NATs is via User Datagram Protocol (UDP) hole punching. In this model, a UDP connection is established between hosts by using a third-party, publicly accessible relay server that facilitates the connection [105]. For example, Globus Transfer uses such a mechanism for transfers between two Globus Connect Personal endpoints [59]. The FaaS Messaging Interface (FMI), modeled after the Message Passing Interface (MPI) [175], provides point-to-point

and collective communication for serverless functions [73]. It supports both mediated channels, which use external storage accessible by all functions, and direct channels, which use Transmission Control Protocol (TCP) connections that, however, may not be accessible by all function invocations. When direct TCP communication is not possible, FMI uses a relay server and hole punching to establish a direct connection between function invocations. Libp2p defines a modular specification for developing peer-to-peer applications with support for NAT traversal [164, 224]; implementations of the protocol are provided or planned for many popular languages.

Workflows: Contemporary computational science applications executed at scale are increasingly written as workflows, collections of many distinct tasks interconnected through data dependencies. This trend has necessitated the development of advanced computational tools and frameworks that can glue software components together and provide a platform for scalable and flexible execution on arbitrary hardware. Workflow management systems, including Dask [219], Dragon [212], FireWorks [141], Parsl [27], Pegasus [86], Radical Pilot [16], Ray [184], Swift [256], and TaskVine [228], have emerged as powerful solutions to this challenge. Applications can be expressed as fine-grained tasks, often a function, with special constructs, such as futures, used to implicitly express inter-task data dependencies. The framework then abstracts the complexities of executing tasks in parallel and managing intermediate data across personal, cloud, or high-performance computing (HPC) systems. Many of these frameworks include both intra- and inter-site data transfer functionality as a core feature, for movement both of input and output data between clients and execution environments and of intermediate data between tasks [85, 12]. Parsl, Pegasus, and Swift all enable transparent intra-site communication via shared file systems, and provide some support for inter-site communication via files. For example, Parsl supports movement of Python objects via ZeroMQ [135] sockets in a hub-spoke architecture between the main Parsl process and workers; uni-directional file staging via Hypertext Transfer Protocol (HTTP) and

File Transfer Protocol (FTP) (developers must specify URLs for downloading or uploading artifacts); and Globus-based data movement between sites based on user-supplied configuration information specifying the Globus endpoint for each site, in which case, Parsl inserts data transfer operations in the workflow graph and executes movement before/after task execution.

Function-as-a-Service: Cloud-hosted serverless frameworks (e.g., Amazon Web Services (AWS) Lambda [17], Azure Functions [176], Google Cloud Functions [123]) serialize input and output data along with a function request or result. Functions can also read and write from cloud object stores (e.g., AWS S3 [43]) and pass object IDs as function inputs or outputs. Apache OpenWhisk [24], Fn [104], KNIX MicroFunctions [150], Abaco [116] are serverless frameworks which can be deployed on existing compute infrastructure. ChainFaaS [117] and DFaaS [65] extend the serverless model to enable execution on personal and edge devices. These systems use Docker for deployment or otherwise require root privileges, thus making them unsuitable for HPC.

Globus Compute [61], formerly *func*X, is a cloud-managed serverless framework that supports remote execution across federated endpoints such as cloud machines, HPC clusters, edge nodes, and workstations. The Globus Compute cloud service routes each client task to a specified target endpoint and stores results until retrieved by the client. The cloud service is essential to providing the compute-anywhere features of Globus Compute but requires that all inputs and results be sent to, and stored in, the cloud (Redis servers hosted in AWS and S3), even if the Globus Compute client and endpoints are located in the same site, which introduces additional latency and costs. Globus Compute enforces a 5 MB task payload size limit to manage storage and egress costs.

3.2 Design and Implementation

Here we describe the goals and design of the framework, and detail the implementation choices necessary to enable the proxy model. ProxyStore provides four primary components: the `Proxy`, `Factory`, `Connector`, and `Store`. The ProxyStore design enables more features and greater flexibility compared to the de facto approaches for mediated communication in federated applications.

3.2.1 Assumptions

We make the following assumptions about usage model and target applications. (1) The application requires some combination of space, time, and synchronization decoupling (i.e., ProxyStore is not intended for highly synchronous applications). (2) The application can be described as a composition of dependent tasks that consume and produce Python objects. We target Python for its pervasiveness in the scientific and workflow systems communities and for the language features that make the proxy model possible. (3) Intermediate objects are written only once but may be read many times. Most task-based workflows fit this paradigm, especially those with pure functional tasks. (4) Objects need not be moved to a centralized store, but can stay where they are produced or be moved to where they are to be consumed. (5) Users may have their own object storage and communication backends that meet their performance and persistence requirements. Federated applications that employ FaaS and workflow systems fit these assumptions well.

3.2.2 Requirements

ProxyStore must support applications with any of the following attributes: (1) data can be produced in many places and must be globally accessible (including across NATs); (2) computation can be performed in many places, and regardless of location must be able

to consume previously produced data and produce new objects that can then be accessed by others; (3) objects may be persistent (must be available for future unknown purposes) or ephemeral (e.g., an intermediate value that is produced by one function and consumed by another, and then never accessed again) and, thus, must exist as long as their associated proxies exist; (4) storage locations have varying reliability (e.g., persistent disk vs. in-memory) and performance; (5) multiple storage or communication methods may need to be employed within a single workload; and (6) data consumers need not know the communication method required to access data.

3.2.3 The Proxy and Factory Objects

We meet these design requirements via the use of lazy, transparent object proxies that act as wide-area object references. The term *proxy* in computer programming refers to an object that acts as the interface for another object. Proxies are commonly used to add additional functionality to their *target* object or enforce assertions prior to forwarding operations to the target. For example, a proxy can wrap sensitive objects with access control or provide caching for expensive operations.

Two valuable properties that a proxy can provide are *transparency* and *lazy resolution*. A *transparent* proxy behaves identically to its target object by forwarding all operations on itself to the target. For example, given a proxy `p` of an object `x`, the types of `p` and `x` will be equivalent: i.e., `isinstance(p, type(x))` and any operation on `p` will invoke the corresponding operation on `x`.

A *lazy* or *virtual* proxy provides just-in-time resolution of its target object. The proxy is initialized with a *factory* rather than the target object. A factory is an object that is callable like a function and returns the target object. The proxy is *lazy* in that it does not call the factory to retrieve the target until it is first accessed—a process that is referred to as *resolving* the proxy. Functionally, proxies have both pass-by-reference and pass-by-value

attributes. The eventual user of the proxied data gets a copy, but unnecessary copies are avoided when the proxy is passed among multiple functions.

This factory-proxy paradigm provides powerful capabilities. The proxy itself is a light-weight reference to the target that can be communicated cheaply between processes and systems. The proxy is self-contained because the proxy always contains its factory and the factory includes all logic for data retrieval and manipulation. That is, the proxy does not need any external information to function correctly. Proxies eliminate the need for shims or wrapper functions that convert objects into forms expected by downstream code. Rather, the proxy can be passed to any existing method or function and the conversion is handled internally by the factory. The consumer code is unaware that the resulting object is anything other than what it expected. Proxies also have other advantages. For example: lazy resolution can help amortize costs and avoids unnecessary computation/communication for objects that are never used; nested proxies can enable partial resolution of large objects; and proxies can be moved in place of confidential data (e.g., patient health information) while ensuring that the data can be resolved only where permitted.

ProxyStore implements lazy transparent object proxies. The `Proxy` class implementation is initialized with a factory and intercepts any access to a proxy instance attribute or method; calls the factory to resolve and cache the target object, if the target has not yet been resolved; and forwards the intercepted action to the cached target. The factory used to initialize a proxy can be any callable Python object (i.e., any object that implements `__call__`, such as lambdas, functions, and callable class instances), as shown in Figure 3.2. `Proxy` modifies its own pickling behavior to include only the factory, not the target, when serializing the proxy, so as to ensure that (1) proxies are small when communicated and (2) a proxy can still be resolved after being communicated to another process.

Many computing frameworks perform introspection on objects to enable optimizations. For example, Dask hashes input arguments to tasks to enable reuse of the results from

```python
import numpy as np
from proxystore.proxy import Proxy

def factory_function() -> list[int]:
    # Function that when called returns target object
    return [1, 2, 3]

p = Proxy(factory_function)
assert p == [1, 2, 3]

class ClassFactory[T]():
   def __init__(self, obj: T):
       # Class factory are stateful, enabling more
       # complex target object resolution
       self.obj = obj

   def __call__(self) -> T:
       # Called to resolve the target object
       return self.obj

x = np.array([1, 2, 3])
p = Proxy(ClassFactory(x))

assert isinstance(p, Proxy)
assert isinstance(p, np.ndarray)

# Using the proxy is equivalent to using the numpy array directly
assert np.array_equal(p, [1, 2, 3])
assert np.sum(p) == 6
y = x + p
assert np.array_equal(y, [2, 4, 6])
```

Figure 3.2: `Proxy` instances are created from a factory, a callable Python object such as a function or class that implements `__call__`. The proxy will invoke the factory just-in-time to retrieve the target object, after which the proxy will transparently act as the target.

previously computed pure functions (functions that always return the same result given the same inputs). Similarly, Dask inspects task object types to apply specialized serializers. These optimizations enhance performance but interact poorly with proxy types. For example, accessing the `__module__` property of a proxy to serialize the type by reference (as done by `pickle`) will cause an `AttributeError`, and hashing or checking the type of a proxy would resolve the proxy, incurring unexpected and hard to debug I/O operations. To resolve this, the `Proxy` uses a custom implementation of Python's `@property` decorator to cache common read-only attributes of a proxied object. These include the module path, the class type, and the hash of the target object to ensure that a proxy need not be resolved when common object metadata are accessed.

The `Proxy` relies strongly on Python's duck typing and, thus, code that uses ProxyStore cannot be statically analyzed and validated using static type checkers such as mypy [186], leading to often cryptic errors when a proxy type is used incorrectly at runtime. ProxyStore provides a mypy extension to support static inference with proxy types. For example, mypy can understand that any attributes or methods on a type `T` are also available on a `Proxy[T]` or that a function that accepts a `ProxyOr[T]` should work with a `T` or `Proxy[T]`. This extension ensures that developers can confidently use `Proxy` types within larger, type-annotated code-bases.

When a proxy is used, its factory must be able to resolve its target object efficiently. In a distributed application, this means a factory must be resolvable when the producer and consumer processes exist independently in space or time. Facilitating this property when processes can exist in the same network or across multiple requires careful consideration for the underlying mediated communication channels used. We discuss how we achieve this goal with the `Connector` in the following section.

3.2.4 The Connector Protocol

```python
KeyT = TypeVar('KeyT', bound=NamedTuple)

class Connector(Protocol[KeyT]):
    def close(self) -> None: ...
    def config(self) -> dict[str, Any]: ...
    def from_config(self, config: dict[str, Any]) -> Connector[KeyT]: ...
    def evict(self, key: KeyT) -> None: ...
    def exists(self, key: KeyT) -> bool: ...
    def get(self, key: KeyT) -> bytes | None: ...
    def get_batch(self, Sequence[KeyT]) -> list[bytes | None]: ...
    def put(self, obj: bytes) -> KeyT: ...
    def put_batch(self, objs: Sequence[bytes]) -> list[KeyT]: ...
```

Figure 3.3: Description of the `Connector` protocol.

The `Connector` is a low level interface to a mediated communication channel. In order to support a wide range of application requirements, we have designed ProxyStore to be extensible to support various mediated channels that can support different space and time decoupling patterns. The `Connector` protocol defines how a client can connect to or operate on a mediated channel, and a `Connector` implementation must provide four primary operations: `evict`, `exist`, `get`, and `put`. The complete protocol is described in Figure 3.3, and includes batched versions of operations and configuration mechanisms. The operations act on byte-string data and keys. E.g., `put` takes a byte-string to put in the mediated channel and returns a uniquely identifying key (a tuple of metadata); the byte-string is retrievable by calling `get` on the key. We chose this model so that third-party code can easily provide new `Connectors` that are plug-and-play with the rest of ProxyStore's features. A `Connector` implementation can be either an interface to an external mediated channel (e.g., a Redis server) or a mediated channel itself. ProxyStore provides many `Connector` implementations that fit both of these categories which we describe further in Section 3.3.

3.2.5 The Store Interface

The `Store` is the high-level interface used by applications to interact with ProxyStore as shown in Figure 3.4. A `Store` is initialized with a `Connector` instance (a dependency

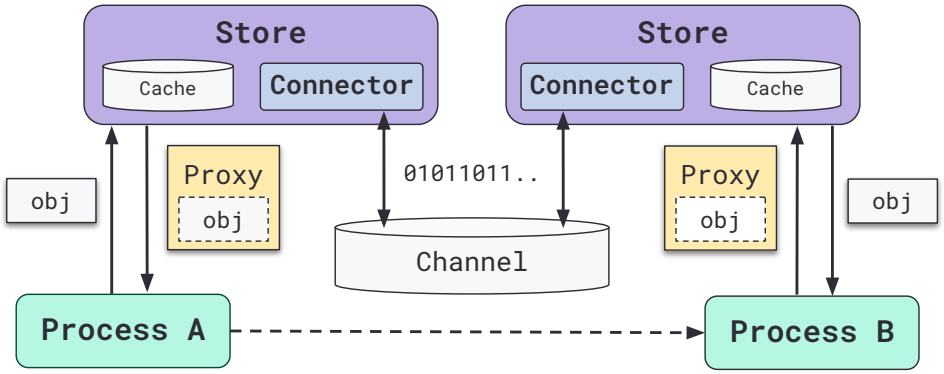


Figure 3.4: Processes interact with a `Store` to proxy objects, and proxy consuming processes will transparently interact with the local `Store` instance. The underlying communication is executed using the `Connector` interface.

```python
from proxystore.connectors.redis import RedisConnector
from proxystore.proxy import Proxy
from proxystore.store import Store

def compute(data, Data) -> ...:
    # Proxy data is resolved from 'demo-store' on first use
    assert isinstance(data, MyDataType)
    # More computation...

with Store('demo-store', RedisConnector(...)) as store:
    data = Data(...)

    # Store the object and get a proxy
    proxy = store.proxy(data)
    assert isinstance(proxy, Proxy)

    compute(proxy)  # Succeeds
```

Figure 3.5: Example of creating a proxy via the `Store`.

injection pattern) and provides additional functionality on top of the `Connector`. Similar to the `Connector`, the `Store` exposes `evict`, `exist`, `get`, and `put` operations; however, these operations act on Python objects rather than byte strings. The `Store` (de)serializes objects before invoking the corresponding operation on the `Connector`; custom (de)serialize functions can be registered with the `Store` if needed. The `Store` also provides caching of operations to reduce communication costs, with caching performed after deserialization to avoid duplicate deserializations.

However, rather than the application invoking the aforementioned operations directly, the `proxy` method, also provided by the `Store`, is used. Calling `Store.proxy` puts an object in the mediated channel via the `Connector` instance and returns a proxy (Figure 3.5). The object is serialized before being put in the mediated channel; a factory is generated, containing the key returned by the `Connector` and additional information necessary to retrieve the object from the mediated channel; and then a new proxy, internalized with the factory, is returned.

An evict flag can be passed when creating a proxy. If set, the proxy will evict the object from the mediated channel when first resolved. Subsequently, the `proxy` operation, alone, is a complete interface to an object store because the `proxy` method handles the `put` operation and the proxy resolution process handles `get`/`evict`.

The `Proxy` and `Factory` instances created by a `Store` provide functionality for asynchronously resolving the target object in a background thread using the `resolve_async` function. This is useful in code which expects a proxy and wants to overlap the communication of the proxy resolution with other computations.

`Store` instances are registered globally within a process by name so that initialization is performed only once, caches are shared, and stateful connection objects in the `Connector` are reused. Consider a `Connector` instance C and corresponding `Store` S. S has been registered in process P_a with name "my-store" and is used to create a proxy p. If p is resolved on a remote process P_b where a `Store` with name "my-store" has not yet been registered, p

```python
import io
from typing import Any

import torch
from proxystore.serialize import serialize
from proxystore.store import Store

def serialize_torch_model(obj: Any) -> bytes:
    if isinstance(obj, torch.nn.Module):
        buf = io.BytesIO()
        torch.save(model, buf)
        return buf.read()
    else:
        # Fallback for unsupported types
        return serialize(obj)

model = torch.nn.Module()

store = Store(serializer=serialize_torch_model)
proxy = store.proxy(model)
```

Figure 3.6: Example of a `Store` configured to use a custom serializer for PyTorch models.

will initialize and register a new `Store` instance named "my-store" with the appropriate `Connector` when p is resolved. This is possible because p's factory, created in process P_a, includes the appropriate metadata necessary to recreate C and S in process P_b. Subsequent proxies created by any `Store` with the same name and resolved in P_b will then use the registered `Store` rather than initializing a new one.

Providing `metrics=True` when instantiating a `Store` enables performance tracking on operations, such as to inspect the average communication time or number of cache hits. Metrics will also be recorded on proxies created by that `Store`. Metrics are local to the process.

Serialization of large or complex Python objects within the `Store` can be expensive. The default ProxyStore serializer aims to be fast and compatible with any Python object. To achieve this, we minimize memory copies by using specialized serializers for common data formats in scientific computing, including NumPy arrays, Pandas DataFrames, and Polars DataFrames. Serialization of these types is 2–3× faster compared to `pickle`. For unknown

Table 3.1: Summary of provided `Connector` implementations.

Connector	Storage	Intra-Site	Inter-Site	Persistence
File	Disk	✓		✓
Redis	Hybrid	✓		✓
Margo	Memory	✓		
UCX	Memory	✓		
ZMQ	Memory	✓		
DAOS	Hybrid	✓		✓
Globus	Disk		✓	✓
Endpoint	Hybrid	✓	✓	✓

data types, `pickle` is used with a fallback to `cloudpickle` if `pickle` fails. Alternatively, users may override the default serializer for a `Store` upon initialization, or override the serializer when creating individual proxies. An example, demonstrating a custom PyTorch model serializer, is shown in Figure 3.6.

3.3 Mediated Communication Methods

All `Connector` implementations are built on mediated, byte-level data storage. Data storage methods are broadly classified as in-memory or on-disk. Mediated channels use one or both methods, depending on performance and persistence aims. The proxy abstraction provided by the `Store` enables a producer to unilaterally (i.e., without the agreement of the receiver) choose the best mediated channel for object communication.

Data storage may be local to the process or machine, within the same network, or at a remote site. Here, we describe the various `Connector` implementations provided out-of-the-box that can be used with the `Store` that support in-memory and on-disk data storage within and between sites (summarized in Table 3.1). We also describe an implementation provided `MultiConnector` abstraction which enables intelligent routing of objects across connectors.

3.3.1 Intra-Site Communication

Various technologies, such as shared file systems, TCP/UDP sockets, and remote distributed memory access (RDMA), enable data transfers between nodes on the same local area network: i.e., not located behind different NATs.

On-disk Storage: For large objects or data that needs to be persisted, ProxyStore provides the `FileConnector` for mediated communication via a shared file system. The `FileConnector` is initialized with a path to a data directory in which proxied objects can be serialized and written (and then read) as files.

Hybrid Storage: The `RedisConnector` uses an existing Redis [218] or KeyDB [229] server as the mediator. Redis provides a hybrid between in-memory and on-disk data storage with low-latency, easy configuration, persistence, and optional resilience via replication across nodes. The `RedisConnector` implementation is only 31 lines of Python code, exemplifying the ease with which the proxy model can be extended to other mediated communication methods via the `Connector` protocol.

The `DAOSConnector` enables use of DAOS, a distributed object store designed for high-speed non-volatile memory like Intel Optane [163] and NVMe. DAOS is available on next-generation compute clusters like Aurora at the Argonne Leadership Computing Facility and is typically deployed across a machine in a similar fashion to a shared file system like Lustre. Thus, using the DAOS within ProxyStore is easy—minimal configuration is required—and performance is superior to shared file systems. The user need only provide the name of their DAOS pool and container to use.

Distributed In-memory Storage: Distributed memory backends for intra-site communication permit applications to benefit from increased memory capacity and scalability. Two implementations are provided, `MargoConnector` and `UCXConnector`, to leverage rapid communication on high-speed networks by using the Py-Mochi-Margo [206] and UCX-Py [240] libraries, respectively. A third implementation, `ZMQConnector`, uses ZeroMQ for commu-

nication and is provided as a fallback for compatibility. When one of these connectors is initialized for the first time in a process, it spawns a process that acts as the storage server for that node. Thus, these connectors act as interfaces to these spawned servers which make up the actual distributed in-memory store. These distributed storage methods are elastic—expanding as proxies are propagated to new nodes—and enable the use of state-of-the-art direct communication methods in a mediated fashion.

3.3.2 Inter-Site Communication

ProxyStore enables data transfer between computers at different sites (and also between computers at the same site that are located behind different NATs) by using disk-to-disk solutions for bulk data and memory-to-memory solutions for low latency.

On-disk Storage: Bulk file transfers between sites are ubiquitous in scientific applications. To support such transfers, the `FileConnector` is extended as the `GlobusConnector` to use Globus to move object files between sites. Globus transfer supports efficient, secure, and reliable file movement and is widely adopted across computing centers with more than 20 000 active endpoints. Globus Connect software is easily deployed on computers without an existing endpoint.

The `GlobusConnector` is initialized with a mapping of hostname regular expressions to a tuple of `(Globus Endpoint UUID, Endpoint path)`. A proxy, while resolving itself, will match the hostname of the current system to the provided hostname regular expressions to determine the directory on the local endpoint with the transferred files. `GlobusConnector` keys are the tuple `(object_id, task_id)` where the `task_id` is the Globus transfer task ID. A proxy will wait for the transfer task to succeed before resolving itself or raise an error if there is a Globus Transfer failure. For efficient movement of many objects, the `Store` provides a `proxy_batch` method that will invoke a batch transfer of proxied objects as a single Globus transfer.

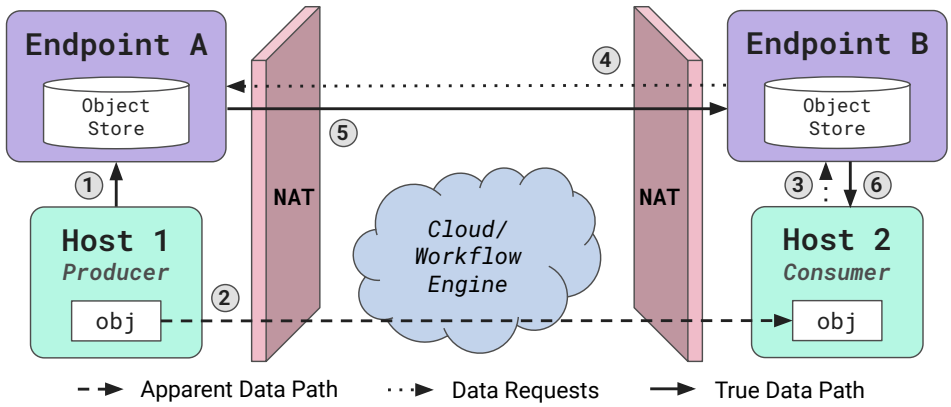


Figure 3.7: Data flow when transferring objects via proxies and PS-endpoints between sites. The proxy gives the appearance that data flows through the entire application, but the actual data transfer is performed via a peer connection between the PS-endpoints at the producing and consuming sites.

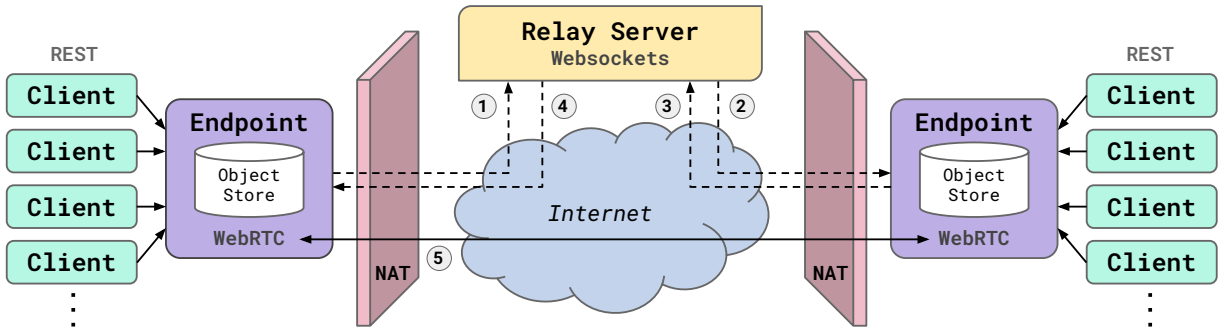


Figure 3.8: Client requests directed to any PS-endpoint are forwarded to the correct PS-endpoint via a peer connection. The peer connections are opened by using UDP hole-punching and a publicly accessible relay server. When PS-endpoint A wants to connect to PS-endpoint B, A asks the relay server R to forward a session description protocol (SDP) [127] to B (1 and 2). This description contains information about how the two peers can connect, such as what protocols they support. B receives A's session description from R and replies with B's session description (3 and 4). A and B then generate interactive connectivity establishment (ICE) candidates [147] (i.e., public IPs and ports to try for the connection) which they exchange via R. Once A and B have exchanged ICE candidates, they can connect by completing the hole punching process (5).

In-memory Storage: A common pattern in inter-site applications is the use of a centralized orchestrator that can communicate with all sites and mediates the control flow between actors across the sites. A simple example is a cloud-hosted queue of tasks, which actors at each site poll to obtain tasks to execute or to place new tasks on the queue. In this model, data producers may not always know where data are eventually needed, but it can also be prohibitively expensive (monetary or overhead) to store data in the cloud or in some other central service. The proxy model allows applications to pass data by reference across sites and perform the underlying communication more directly, avoiding additional overheads of unnecessary data movements. ProxyStore includes a ProxyStore endpoint (PS-endpoints) model that facilitates direct data transfer between sites as shown in Figure 3.7.

PS-endpoints are in-memory object stores, with optional on-disk storage if host memory is insufficient or data persistence is required. PS-endpoints are managed with the `proxystore-endpoint` command-line interface. Clients use the `EndpointConnector` to interact with PS-endpoints, and object keys are the tuple `(object_id, endpoint_id)`. If a PS-endpoint receives an operation request on a key with an `endpoint_id` that is not its own, the PS-endpoint establishes a peer connection to the target PS-endpoint and forwards the request.

Peer-to-peer communication between PS-endpoints is achieved via the Web Real-Time Communication (WebRTC) standard [253, 41]—specifically, by using the *RTCPeerConnection* and *RTCDataChannel* components of the `aiortc` open-source WebRTC implementation [9]. The RTCPeerConnection handles the establishment of peer connections across firewalls using NAT traversal and hole punching, as described in Section 3.1; security; and connection management. The RTCDataChannels are associated with an RTCPeerConnection and enable bidirectional transfer between peers; data are transported over the channel via SCTP (Stream Control Transmission Protocol) over DTLS (Datagram Transport Layer

Security).

PS-endpoints use a publicly accessible relay server or signaling server to facilitate the creation of RTCPeerConnections. The process of establishing the connection via the relay server is illustrated in Figure 3.8. Once a peer connection is established, the PS-endpoints maintain the connection until one of the PS-endpoints is stopped; the connection is re-established if lost for any reason, e.g., due to a PS-endpoint going offline temporarily. The hosting requirements for the relay server are minimal because establishing a peer connection only requires the relay server to exchange a few small (O(KB)) messages between the peers. We provide a WebSocket-based [102] relay server implementation that can be self hosted, and we host a public relay server with authentication provided via Globus Auth [239] (widely adopted in academic communities), which allows users to authenticate with their institutional credentials.

Continuing with the environment described in Figure 3.7, the flow of data and their associated proxies is as follows:

1. Host A creates a proxy of the target object. The serialized target is placed in Host A's local endpoint (Endpoint 1). The proxy contains the key referencing the target, the endpoint UUID with the target data (Endpoint 1's UUID), and the list of all endpoint UUIDs configured with the `EndpointConnector` (the UUIDs of Endpoints 1 and 2).

2. Host A communicates the proxy object to Host B. This communication is cheap because the proxy is a lightweight reference to the object.

3. Host B receives the proxy and attempts to use the proxy initiating the proxy resolution process. The proxy requests the data from Host B's local endpoint (Endpoint 2).

4. Endpoint 2 sees that the proxy is requesting data from a different endpoint (Endpoint 1) so Endpoint 2 initiates a peer connection to Endpoint 1 and requests the data.

5. Endpoint 1 sends the data to Endpoint 2.

6. Endpoint 2 replies to Host B's request for the data with the data received from Endpoint 2. Host B deserializes the target object and the proxy is resolved.

PS-endpoints are single-threaded, asyncio applications. When started, they connect and register with the relay server, and the relay server assigns a unique UUID if not already assigned. An asyncio task is created which listens on the WebSocket connection with the relay server for incoming peering requests and responds appropriately. Once a peer connection is established, the PS-endpoint also listens for and responds to incoming requests from its peers.

Over the last couple of years, we have hosted a public relay server that is the default when using the CLI. To configure and start and endpoint:

```
$ proxystore-endpoint configure demo
INFO: Configured endpoint: demo <a6c7f036-3e29-4a7a-bf90-5a5f21056e39>
INFO: Config and log file directory: ~/.local/share/proxystore/demo
INFO: Start the endpoint with:
INFO:   $ proxystore-endpoint start demo
```

Users will be prompted to authenticate with Globus Auth, ensuring that endpoints can only communicate if owned by the same user. Additional CLI tools are provided beyond those for configuring, starting, and stopping user endpoints: `proxystore-endpoint check-nat` runs heuristics to determine if NAT hole-punching is likely to work on the current network, and `proxystore-endpoint test` can be used to query endpoint and perform test transfers.

Similarly, a relay server can be started via the CLI.

```
$ proxystore-relay --config relay.toml
```

Here, `relay.toml` defines the serving parameters, including keys for TLS and the authentication mechansims.

3.3.3 The MultiConnector Abstraction

Sophisticated applications that employ multiple data communication patterns can benefit from using multiple types of mediated communication (i.e., `Connector` implementations). Rather than creating multiple `Store` instances and a policy directing when to use each instance, ProxyStore provides the `MultiConnector` abstraction, which is initialized with a mapping of `Connector` instances to policies, to indicate how each `Connector` should be used. Thus, an application can use a single `Store` instance, and operations will be routed transparently and automatically to the appropriate `Connector`. Policy definitions are flexible and can be extended by developers. An example policy may include minimum and maximum object sizes, representing the ideal operating range for that `Connector`; tags denoting the sites at which the `Connector` is accessible (e.g., a `MargoConnector` is available within a single cluster, while an `EndpointConnector` is available at multiple sites); and a prioritization function for breaking ties when multiple `Connector` instances are otherwise suitable for a given object.

`Store` operations accept additional keyword arguments that are passed to the corresponding `Connector` method. The `put` and `proxy` methods of `MultiConnector` take a set of optional constraints on the data being stored. These constraints, as well as other metadata (object type or size, location, etc.), are matched against each policy of each `Connector` managed by the `MultiConnector`. If no match is found then an error is raised, although deployments may often prefer to provide a low priority fallback with no constraints.

3.4 Synthetic Evaluations

We evaluate the component-level performance of ProxyStore, quantify overhead reductions in compute frameworks when using ProxyStore, and demonstrate the use of ProxyStore in three real-world scientific applications. For brevity, we use the term `XStore` to mean we are using a ProxyStore `Store` initialized with an `XConnector` for communication.

E.g., `RedisStore` is a `Store` initialized with a `RedisConnector`.

We performed experiments using six machines: Theta, Polaris, Perlmutter, Frontera, Midway2, and Chameleon Cloud. Theta and Polaris are at Argonne National Laboratory. Theta is a 4392-node Intel Knights Landing (KNL) cluster. The 560-node Polaris has four NVIDIA A100 GPUs per node. NERSC's Perlmutter cluster has 1536 NVIDIA A100 GPU nodes and 3072 AMD EPYC CPU nodes. We use the login nodes of Midway2 at the University of Chicago and the Texas Advanced Computing Center's (TACC) Frontera cluster as clients to distributed applications running on the aforementioned systems. Chameleon Cloud [146] provides bare-metal compute nodes.

3.4.1 ProxyStore with FaaS

We first evaluate ProxyStore with the federated FaaS platform Globus Compute [61], with the goal of quantifying the performance gains that may be achieved with minimal code changes to the producer and no changes to the compute framework.

To quantify the extent to which passing task inputs with proxies can reduce data transfer overheads, we perform experiments with Globus Compute where we execute no-op and 1 s sleep tasks with payload sizes from 10 bytes to 100 MB (Figure 3.9). We use four different configurations of Globus Compute clients and endpoints. (1) Theta → Theta: Client and tasks all run on the same Theta node. (2) Perlmutter Login → Perlmutter Compute: Client runs on a Perlmutter login node and tasks on a Perlmutter compute node. (3) Midway2 → Theta: Client runs on a Midway2 login node and tasks on a Theta compute node. (4) Frontera → Theta: Client runs on a Frontera login node and tasks on a Theta compute node. In the first two scenarios, the client and task execute in the same site and thus we compare the round-trip time when data are moved via the Globus Compute cloud service to data movement via ProxyStore's `FileStore`, `RedisStore`, and `EndpointStore`. In the latter two scenarios, the client and task execute in different sites, so we compare the baseline

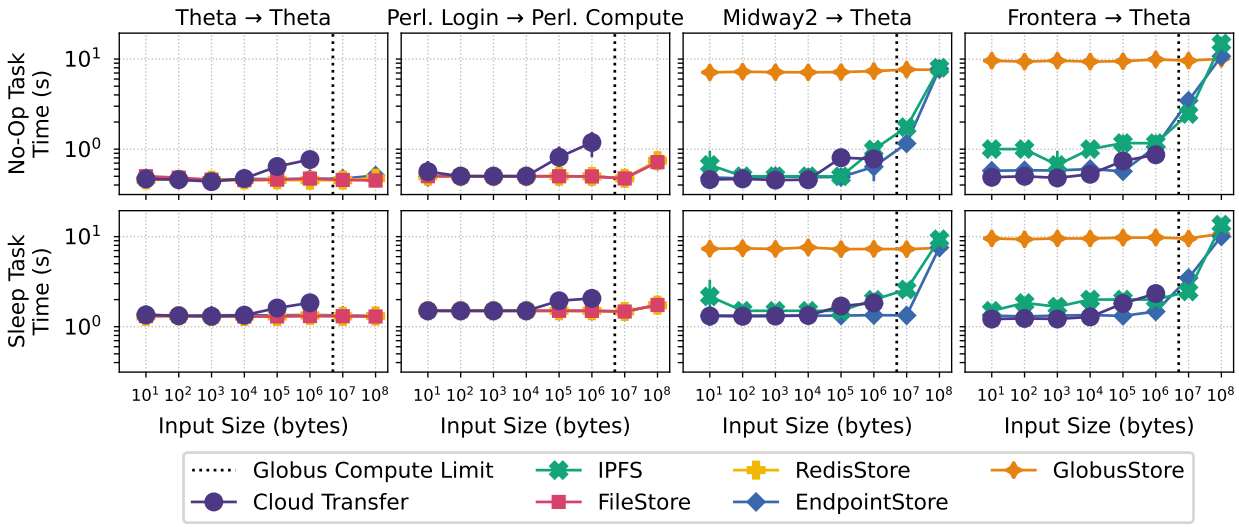


Figure 3.9: Average performance for round-trip Globus Compute no-op (top) and 1 s sleep tasks (bottom), for intra-site (two left columns) and inter-site (two right columns) configurations. In intra-site configurations, we compare baseline input data transfer via cloud to ProxyStore's `FileStore`, `RedisStore`, and `EndpointStore`. For inter-site, we compare to IPFS and ProxyStore's `EndpointStore` and `GlobusStore`. Dashed lines denote the 5 MB Globus Compute payload size limit for transfer via the cloud; ProxyStore can handle >5 MB task payloads without modifying task code to communicate via alternate means. Error bars denote standard deviation but are often smaller than data point markers.

```
1  from globus_compute_sdk import Executor
2  from proxystore.connectors.redis import RedisConnector
3  from proxystore.store import Store
4
5  with Store('example', RedisConnector(...)) as store:
6      data = store.proxy([1, 2, 3])
7
8      with Executor('<UUID>') as gce:
9          future = gce.submit(sum, data)
10         assert future.result() == 6
```

Figure 3.10: Example ProxyStore usage with Globus Compute.

to ProxyStore's `EndpointStore` and `GlobusStore`. We also compare with a configuration in which data are moved to the Globus Compute endpoint by using the InterPlanetary File System (IPFS) [34]. IPFS is a peer-to-peer distributed file system, so we treat the Globus Compute client and Globus Compute endpoint as two nodes of the distributed file system. In no-op tasks, we ensure that the proxy is resolved even though no computation is performed, and in the sleep tasks, we begin asynchronously resolving the proxy before sleeping and then wait on the asynchronous resolve after the sleep to simulate overlapping proxy resolution with compute.

The baseline round-trip time, where data are transferred along with the task request to the Globus Compute cloud service, increases with data size up to the Globus Compute limit. In the first two scenarios where the client and task execution occur at the same site, all three ProxyStore options eliminate the Globus Compute data transfer overhead. This was achieved with only two client-side lines of code: one to initialize the `Store` and one to proxy task inputs before submitting to Globus Compute (Figure 3.10 lines 5–6). The asynchronous resolve in the sleep task requires one additional line of code in the task itself, but the overlap of communication and compute can yield benefits.

In the inter-site cases where the clients run on Midway2 or Frontera login nodes and execute tasks on Theta, we use the `GlobusStore` and `EndpointStore`. `GlobusStore` performance is not competitive with the Globus Compute baseline up to Globus Compute's

payload limit. The performance is a consequence of Globus transfer's hybrid software-as-a-service model, which results in high bandwidth for larger transfers but not low latency for small transfers. However, the benefits of Globus transfer become substantial as data sizes grow beyond those used in this experiment. The `EndpointStore` outperforms the baseline, except for no-op tasks between Frontera and Theta where the performance is comparable. For the largest (100 MB) payloads, `EndpointStore` performance is less than the theoretical peak of the connection. We investigate this discrepancy further in Figure 3.4.3.

We also compare to IPFS for inter-site data transfer. Task data are written to disk, the file is added to IPFS, and the content ID of the IPFS file is passed as input to the Globus Compute task. When the Globus Compute task is invoked, IPFS is used to retrieve the file, and the data are read back into memory. Whereas ProxyStore required two extra client side lines of code, IPFS support required 13 extra lines of code on both the client and task. The performance of PS-endpoints and IPFS for no-op tasks between Midway2 and Theta are within run-to-run variances of each other. PS-endpoints are faster with the one second sleep tasks because of the asynchronous resolution of proxies. PS-endpoints outperform IPFS for Frontera to Theta transfers due to Frontera having a slower file system and slower transfers between the IPFS peers compared to the Midway2 $\rightarrow$ Theta scenario. IPFS and PS-endpoints address a different set of problems—IPFS is designed for decentralized and persistent sharing of content-addressable files; however, IPFS has a mature peer-to-peer transfer protocol which we can use as a point of comparison to show that PS-endpoints can outperform IPFS.

We repeat these experiments with the distributed in-memory connectors described in Section 3.3.1 and compare performance to DataSpaces, a shared-space abstraction designed for large-scale scientific applications. The experiments were executed on Polaris, which has a high-performance HPE Slingshot 11 network, and on two Chameleon Cloud nodes with a Mellanox Connect-X3 40GbE InfiniBand interconnect. Figure 3.11 shows that the baseline

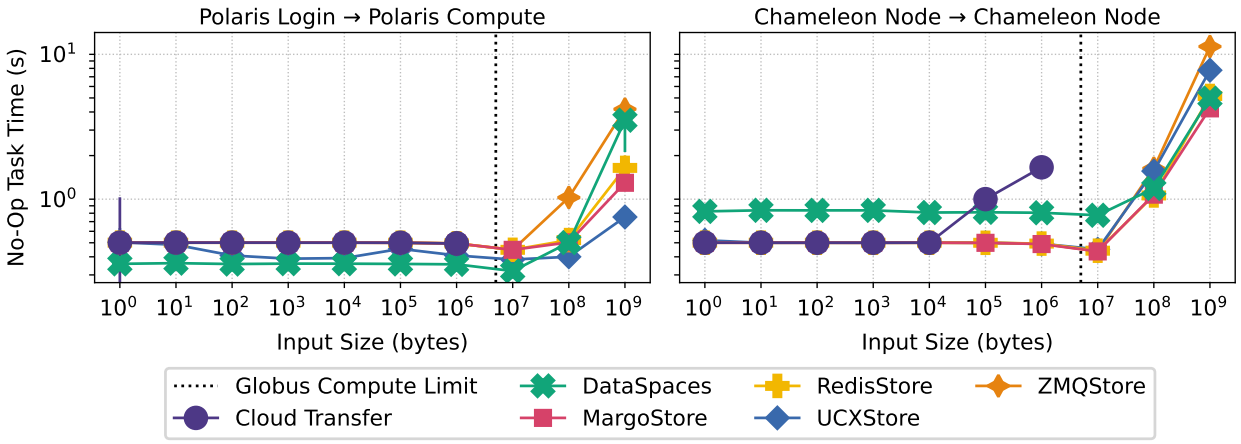


Figure 3.11: Average round-trip performance of no-op Globus Compute tasks on Polaris and Chameleon Cloud for the baseline cloud transfer via the Globus Compute service, ProxyStore centralized stores (`RedisStore`), ProxyStore distributed in-memory stores (`MargoStore`, `UCXStore`, `ZMQStore`) and DataSpaces. Error bars denote standard deviation.

cloud transfer and ProxyStore alternatives all exhibit similar performance at data sizes <1 GB, after which bandwidth dominates performance.

`MargoStore` and `UCXStore`, which both leverage RDMA, achieve the best overall performance on Polaris. However, `UCXStore` performs measurably worse than `MargoStore` and `RedisStore` for larger data sizes on Chameleon. We suspect the disparity is a result of the network differences between the two systems. While we expect DataSpaces and `MargoStore` to perform similarly because both use Margo for the transport layer, `MargoStore` outperforms DataSpaces on both systems. We observed prominent startup overheads, particularly for smaller transfers, with DataSpaces on Chameleon.

We focus on FaaS for HPC and choose Globus Compute because it is designed to coordinate computation across federated resources (e.g., cloud, HPC, and edge devices). However, ProxyStore is agnostic to the compute framework and will work with other FaaS systems. We expect comparable performance characteristics since Globus Compute's data storage and communication mechanisms are similar to cloud-specific FaaS systems.

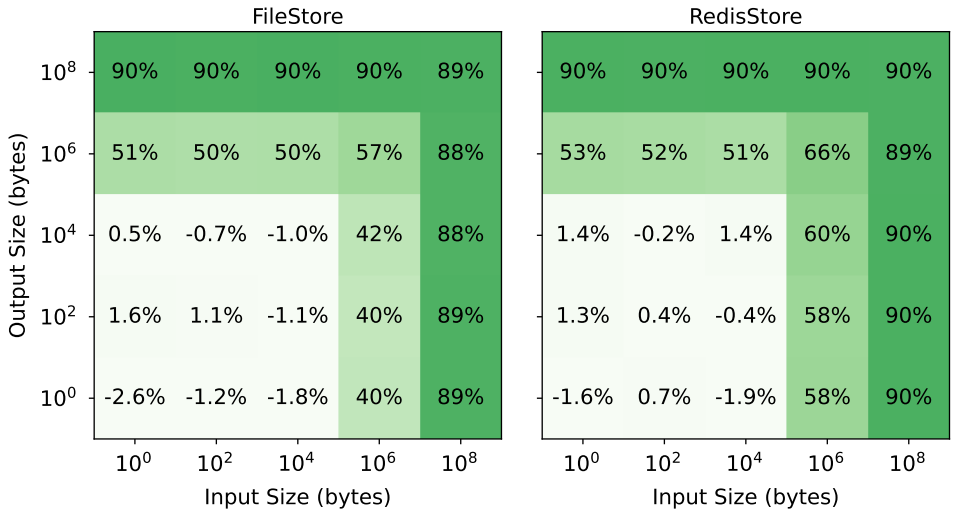


Figure 3.12: Percent improvements in task round-trip time when using ProxyStore to move data vs. Colmena's default method with Parsl. Each task configuration is repeated 100 times, and the median time is used to compute the improvement.

3.4.2 ProxyStore with Workflow Systems

Colmena is a Python library for steering large ensembles of simulations [249]. Colmena applications contain three components (described in more detail in Section 3.6.1: (1) a thinker, one or more agents that create tasks and consume results; (2) a task server, which coordinates tasks to be executed by using a workflow engine (here, Parsl); and (3) workers which execute the tasks and return results to the task server. We integrate ProxyStore into Colmena at the library level. Users can register a `Store` and associated threshold for each task type. Task inputs or results greater than the threshold will be proxied before the task is sent to the task server. Passing proxies with the task can alleviate overheads in the Task Server and underlying workflow system.

We investigate overhead improvements in Figure 3.12, where we report the percent improvement over the baseline of median round-trip task times. We execute a series of no-op tasks using Colmena and Parsl with varied input and output sizes. The thinker, task server, and worker are co-located on a single Theta node to isolate effects of the network. Neither ProxyStore's caching capabilities nor asynchronous resolving of proxies are used. For

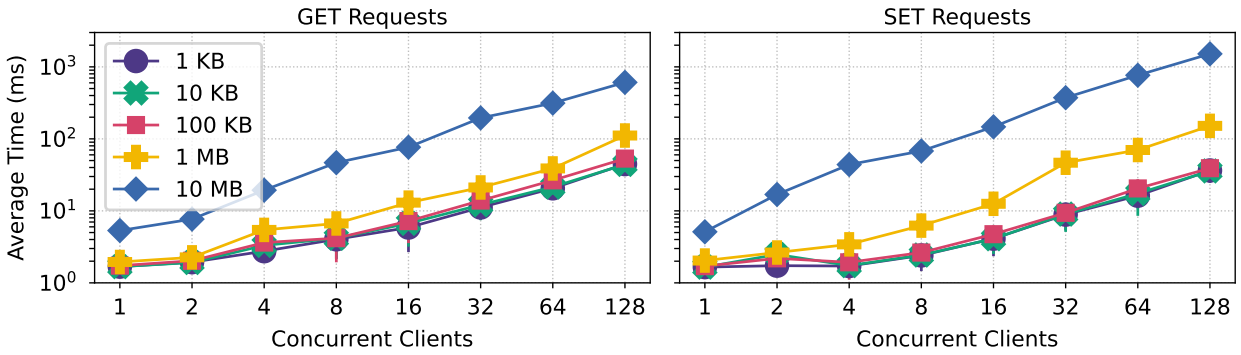


Figure 3.13: Average client `get` and `set` request times to a single PS-endpoint with respect to payload size and concurrent clients issuing the same request. Error bars denote standard deviation.

small data sizes (<100 KB), any improvements in overhead in Colmena are largely negated by the additional overhead of proxying the data (i.e., I/O with storage). However, ProxyStore yields 40–60% improvements in overhead for 1 MB data sizes and 88–89% for 100 MB data sizes. This exemplifies why passing by proxy can be invaluable in distributed systems with many interconnected components. Proxies can be passed around cheaply while ensuring that data are only communicated between producer and consumer.

3.4.3 ProxyStore Endpoint Performance

To better understand the characteristics of PS-endpoints, we next study the times taken for client-to-PS-endpoint requests and PS-endpoint-to-PS-endpoint requests.

Client Access: In Figure 3.13, we show average per-request times for `get` and `set` operations versus the number of concurrent clients making the same request and for varied payload sizes. Each client makes 1000 requests, and the experiment was performed with Python 3.11 on a Perlmutter CPU node. Response times scale linearly with number of clients for more than two concurrent clients, and also scales with payload size. This is reasonable given that the proof-of-concept PS-endpoint implementation is single-threaded. Handling many concurrent workers with low latency is better suited for another mediated

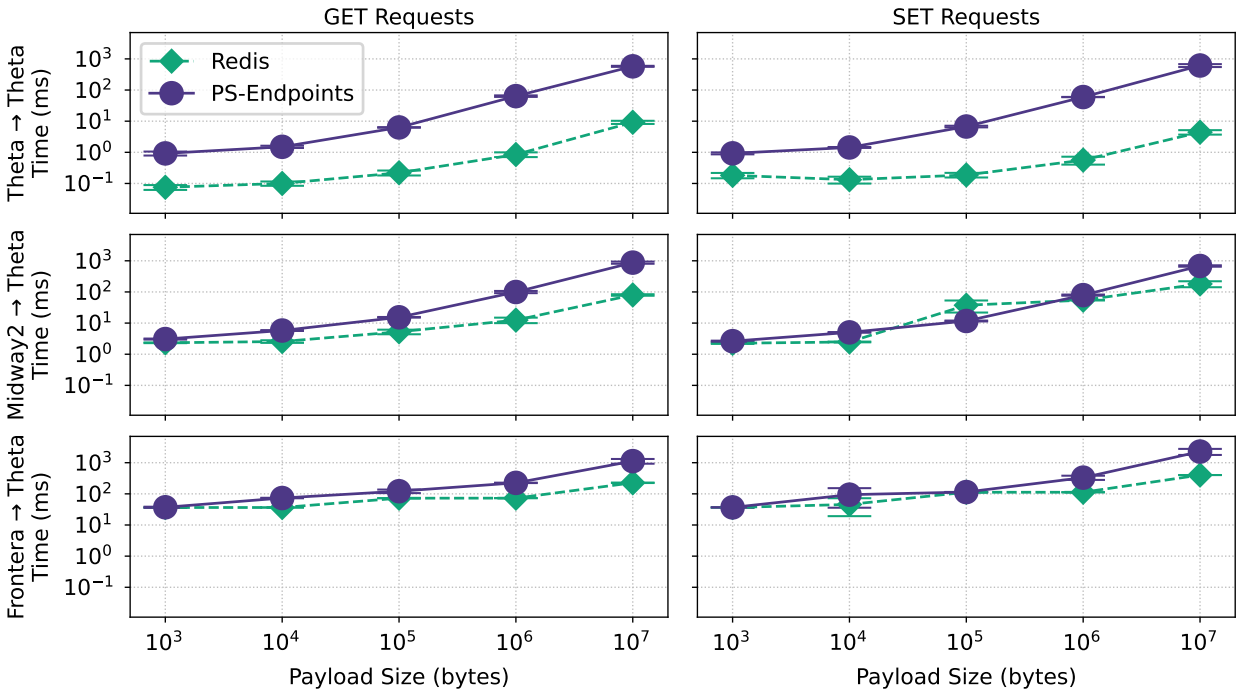


Figure 3.14: Average `get` and `set` times, over 1000 requests, between two PS-endpoints, with error bars showing the standard deviation. Comparisons are made to hosting a Redis server on the target site and opening an SSH tunnel when the two sites are different. The PS-endpoint configuration has one more hop (client—local endpoint—remote endpoint) than Redis (client—remote Redis).

communication channel such as Redis.

Endpoint Peering: The primary use case for PS-endpoints is transfers between different sites. Thus, we measure request times between PS-endpoints as a function of payload size (see Figure 3.14). We consider three scenarios: requests between two PS-endpoints on different Theta nodes, which serves as a baseline; requests between PS-endpoints on Midway2 and Theta; and requests between PS-endpoints on Frontera and Theta. These scenarios differ in latency—packets need only travel tens of meters in the first scenario but 1500 kilometers in the third—and bandwidth—the first scenario can utilize the high bandwidth Aries Dragonfly network of Theta while the latter must cross multiple network boundaries. While no system provides equivalent features to PS-endpoints, we compare its performance to that of a Redis server hosted on the target site with a (manually created) secure shell (SSH) tunnel between

the two sites. While in practice SSH tunnels can be fragile and difficult to configure (e.g., to authenticate automatically), they are commonly used by workflow systems [27, 256] and the comparison can help highlight strengths and weaknesses of the PS-endpoint implementation.

We observe that Redis with SSH is generally faster than PS-endpoints, a result for which we identify two primary reasons. First, the PS-endpoint configuration has one more hop than the Redis configuration because two endpoints must be used in contrast to a single Redis server and SSH tunnel. This factor is most prevalent in the Theta-to-Theta scenario where network latency is minimal so the overhead of the extra hop dominates. Second, we discovered that the `aiortc` RTCDataChannel cannot fully utilize the available bandwidth between sites. This is why the difference in performance between PS-endpoints and Redis increases at larger data sizes. A simple test where we established an RTCDataChannel between a process on Frontera and another on Theta achieved a maximum bandwidth of 80 Mbps, a fraction of the full bandwidth available. This is because computing centers throttle UDP connections to avoid congestion, and `aiortc` congestion control is slower than other congestion control algorithms like Google's BBR [51]. We support multiplexing data transfer over multiple RTCDataChannels; however, the single-threaded asyncio model is unable to benefit from multiplexing over more than a couple RTCDataChannels. Despite these networking limitations, the performance of PS-endpoints is still competitive with Redis for long distance transfers while not requiring SSH tunnels or open ports.

3.5 Application Evaluations

In this section, we demonstrate the performance benefits of ProxyStore using three real-world scientific applications.

Table 3.2: Round-trip task times for the real-time defect analysis application. The Globus Compute endpoint is hosted on a Polaris login node and the tasks are executed on a Polaris compute node. In the Globus Compute baseline and `FileStore` configurations, the client (simulating an experimental setup) is hosted on Theta, and the client is hosted on Midway2 in the `EndpointStore` configuration. Transferring task inputs and outputs via ProxyStore yields >30% performance improvements in intra- and inter-site task execution.

Configuration	Proxied	Time (ms)	Improvement
Globus Compute baseline	—	3411 ± 389	—
`FileStore`	Inputs	2318 ± 130	32.1%
	Inputs/Outputs	2160 ± 46	36.6%
`EndpointStore`	Inputs	2375 ± 98	30.4%
	Inputs/Outputs	2280 ± 107	33.2%

3.5.1 Real Time Defect Analysis

A common pattern in scientific applications is to transfer data produced by an experiment to a compute facility for analysis. For example, Argonne National Laboratory's transmission electron microscopy facility uses Globus Compute to invoke a machine-learned segmentation model to quantify radiation damage in acquired images, dispatching this computation to an HPC facility for fast GPU inference. We modify an open-source real time defect analysis application [247] to create and send proxies of images, rather than the actual images. We create a test deployment to mirror the production environment with remotely located instruments and compute.

We measure the baseline round-trip task time for inference on 1 MB images and compare to `FileStore` and `EndpointStore` (Table 3.2). In all cases, we use a Globus Compute endpoint on a Polaris login node that executes tasks on a Polaris compute node. In the Globus Compute baseline and `FileStore` cases, our client (i.e., simulated beam facility) is hosted on a Theta login node, and in the `EndpointStore` case, when the client is on Midway2, with PS-endpoints on both Midway2 and a Polaris login node. We test with only the input

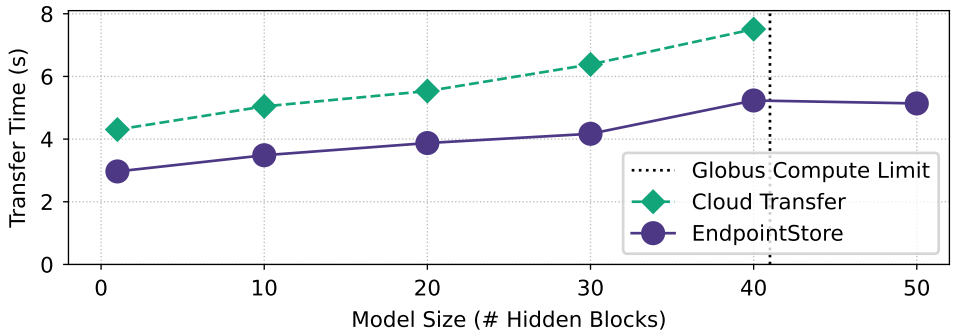


Figure 3.15: Average transfer times for the federated learning use case. PS-endpoints greatly reduce transfer times between nodes compared to cloud transfer. In addition, without ProxyStore, we are unable to transfer models larger than ∼40 hidden blocks due to cloud transfer limits.

images being proxied and with both the input images and inference outputs being proxied. Note that in the former, the code executed on the Globus Compute endpoint is unchanged, while the latter required two additional lines of task code to proxy the output by using the same `Store` that was used to resolve the input proxy.

We see in Table 3.2 that ProxyStore improves round-trip task times by 32.1% and 30.4% with `FileStore` and `EndpointStore`, respectively, when only the inputs are proxied. Further improvements of a few percentage points can be gained if the downstream code also returns proxies. We note that ProxyStore enabled greater flexibility in terms of how clients interact with tasks executed on the Globus Compute endpoint. Each client can choose its preferred communication method, depending on the mediated communication channels available from itself to the Globus Compute endpoint.

3.5.2 Federated Learning

Federated learning (FL) [174] is an increasingly popular approach to distribute machine learning (ML) training across, often edge [227, 168], devices. In FL, an aggregator node initializes an ML model and shares it with edge devices to train the model on their own private data in small batches. Once the edge training is complete, the locally-trained models

are returned to the aggregator node to "average" the model to create a new global model. This new global model is then shared again with the edge devices for further training. In FL only the model is transferred across the network; the distributed edge devices' data are never shared.

Here, we demonstrate the applicability and benefit of ProxyStore not only for FL use cases but edge computing workflows in general. Due to constrained capacity of edge devices, limited connectivity, and application requirements, making effective use of networks and providing low latency is often crucially important [171]. ProxyStore allows for FL control to be separated from data movement, enabling aggregation to occur *anywhere*, and for models to be transferred directly between edge and aggregation nodes when needed.

Our application is implemented using FLoX [151], a FL framework which uses Globus Compute to orchestrate training of TensorFlow [4] models. Our application trains a convolutional neural network for image classification with the Fashion-MNIST dataset [262]. We increase the number of hidden layers of the neural network to show ProxyStore's ability to support larger models compared to a purely FaaS-based approach. We use the same test bed as used in [151] to deploy our application across four edge devices. Figure 3.15 shows the transfer time as we increase the number of model parameters when using Globus Compute or using Globus Compute and ProxyStore. We see that ProxyStore both reduces transfer time and also enables use of larger models. In the cases where Globus Compute is able to complete the model transfer, ProxyStore is able to reduce transfer time by ∼68% on average. Further, ProxyStore can be used to implement hierarchical model aggregation, where sets of edge-trained models are aggregated in a distributed fashion.

3.5.3 Molecular Design

We adapt an open source molecular design workflow to use the `MultiConnector` for communication between tasks. The workflow uses a mix of quantum chemistry simulations and

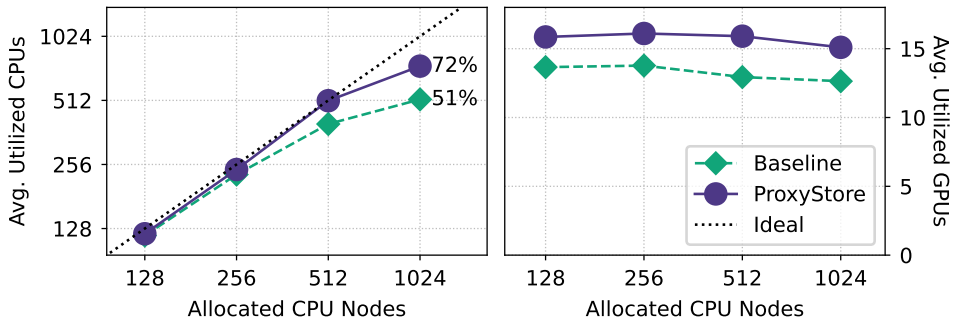


Figure 3.16: Average node utilization of the molecular design application, with and without ProxyStore. The number of GPUs used for training and inference tasks is constant while the number of CPU nodes for simulations is increased. Without ProxyStore, the application struggles to keep all CPU nodes and GPUs fed with tasks. ProxyStore reduces the amount of data flowing through the workflow system, thus reducing the latency between task results being received and new tasks dispatched.

surrogate machine learning models [251] to identify electrolytes with high ionization potentials (IP) in a candidate set.

The workflow comprises: (1) *simulation* tasks that compute IPs on CPUs, (2) *training* tasks that train surrogate models to predict IPs, and (3) *inference* tasks that use trained surrogate models to predict IPs, which are then ranked by confidence and used to guide future simulation tasks. The simulation tasks run on Theta compute nodes, and the training and inference tasks run on a remote GPU node (located behind a different NAT and using a different authentication procedure than Theta). Tasks are orchestrated with a Colmena Thinker running on a Theta login node and task execution is managed with Parsl.

To optimize communication of task data, we use the `MultiConnector` configured to use `RedisConnector` for simulation tasks and `EndpointConnector` for training and inference tasks. `RedisConnector` is suitable for low-latency communication between Theta login and compute nodes and provides persistence when an application spans multiple batch jobs; PS-endpoints enable peer-to-peer transfer of model weights (10 MB in this case) to and from a remote GPU node. Inputs to inference tasks also require peer-to-peer data transfer to remote GPU nodes. The inference dataset is static, so while the first round of proxies result in data

being moved to the GPU node; proxies for later inference rounds benefit from cached data. We also investigated using `GlobusConnector` for data movement. While in this case the dataset was not large enough to benefit from Globus transfers, this would be a good option if a larger dataset were used. This workflow exemplifies how ProxyStore can coordinate optimal communication in complex workflows. We note that no task code needed to be modified to work with the diverse communication methods employed.

In this application, we want to use proxies to reduce overheads in the workflow system. We evaluate their effectiveness for this purpose by measuring average node utilization during application execution as a function of the number of Theta KNL nodes used for simulations. We see in Figure 3.16 that the workflow system struggles to keep nodes fed with new tasks as scale increases. However, use of ProxyStore removes data movement burdens from the workflow system and improves scaling, improving utilization by 29% and 43% at 512 and 1024 nodes, respectively. We also observe ProxyStore improves utilization of the remote GPUs by speeding up data transfer. At 1024 nodes with ProxyStore, computation, rather than communication, becomes the bottleneck because simulation results must be processed serially prior to dispatching new simulations. Processing a simulation result takes 267 ± 518 ms on average in the baseline 1024 node run, but ProxyStore improves this time by 25% to 201 ± 140 ms.

3.6 Framework Integrations

ProxyStore has been integrated into a variety of distributed computing frameworks. This section highlights the technical motivations and performance results for three of these integrations: Colmena, Dask, and Flight.

```python
from colmena.queue import RedisQueues
from proxystore.connectors.redis import RedisConnector
from proxystore.store import Store

store = Store('redis', RedisConnector(...), register=True)

queue = RedisQueues(
    ...,
    proxystore_name='redis',
    proxystore_threshold=100000,
)
```

Figure 3.17: Example of ProxyStore usage within Colmena queues.

3.6.1 Colmena

Colmena [249, 251, 248] is a Python library designed to facilitate intelligent task orchestration for scientific computing workflows. It provides a model for integrating machine learning (ML) models, simulation codes, and data-driven decision-making into high-performance computing (HPC) and distributed computing environments.

Applications built using Colmena are formed of three types of independent processes: a thinker, a task server, and one or more workers. The user-written thinker implements the decision-making policy used to steer the workflow and generate tasks. The thinker communicates task requests to the task server; the results of those tasks, once available, are returned from the task server to the thinker. The thinker makes decisions in response to results being communicated or other events (e.g., availability of compute resources). The task server matches each task request to the corresponding task definition (e.g., function) and dispatches the resulting task to an appropriate worker. Task requests are received from the thinker and can be executed in any order. Each worker receives tasks from the task server, executes each task, and provides results back to the task server. Key innovations within Colmena are focused on maximizing the concurrent and performance of these different components.

For tasks with large input or result values, Colmena uses ProxyStore to pass values

directly from the thinker to the worker—bypassing the task server. (ProxyStore was originally referred to as the "Colmena value server" before being split into a separate project.) The thinker and task server communicate via message queues, and these queues can be configured to automatically proxy input and result values larger than a user-defined threshold. Configuration is simple, requiring only a few additional lines of code to initialize a `Store` and set queue parameters (see Figure 3.17). Alternatively, users can manually proxy large objects in their thinker or task implementations. Colmena starts asynchronously resolving all proxies in a task's input prior to the task being executed on a worker; thus, ProxyStore communication is overlapped with the task's execution. The start of a task often involves some initialization or importing of libraries, such that by the time a value is needed by the task, the corresponding proxy has already been resolved in the background.

We built a synthetic application Colmena application to permit the systematic evaluation of communication overheads in the system. This application uses a thinker plus N workers, one per node; the Thinker generates T identical tasks, each with duration D, unique (and thus non-cacheable) input of size I, and producing a result of size O. This thinker first submits one task per worker and then continues to submit a new task each time that it receives a result, until T tasks have been submitted. We use this application to measure costs for different $\{T, D, I, O, N\}$ combinations.

To evaluate the impact of ProxyStore, we first run the synthetic application for 200 zero-length tasks with 1 MB inputs on eight nodes (i.e., $\{T{=}200, D{=}0, I{=}1$ MB, $O{=}0, N{=}8\}$), both with and without ProxyStore (referred to in figure as the "Value Server"), while measuring task overheads. We see in Figure 3.18 (top left) that the use of ProxyStore reduces, in particular, thinker to task server communication and serialization times. The cost of transferring input data to the worker is reduced by the use of the asynchronous proxy resolution. (Note that if input values were all identical, this cost would be largely eliminated due to caching.)

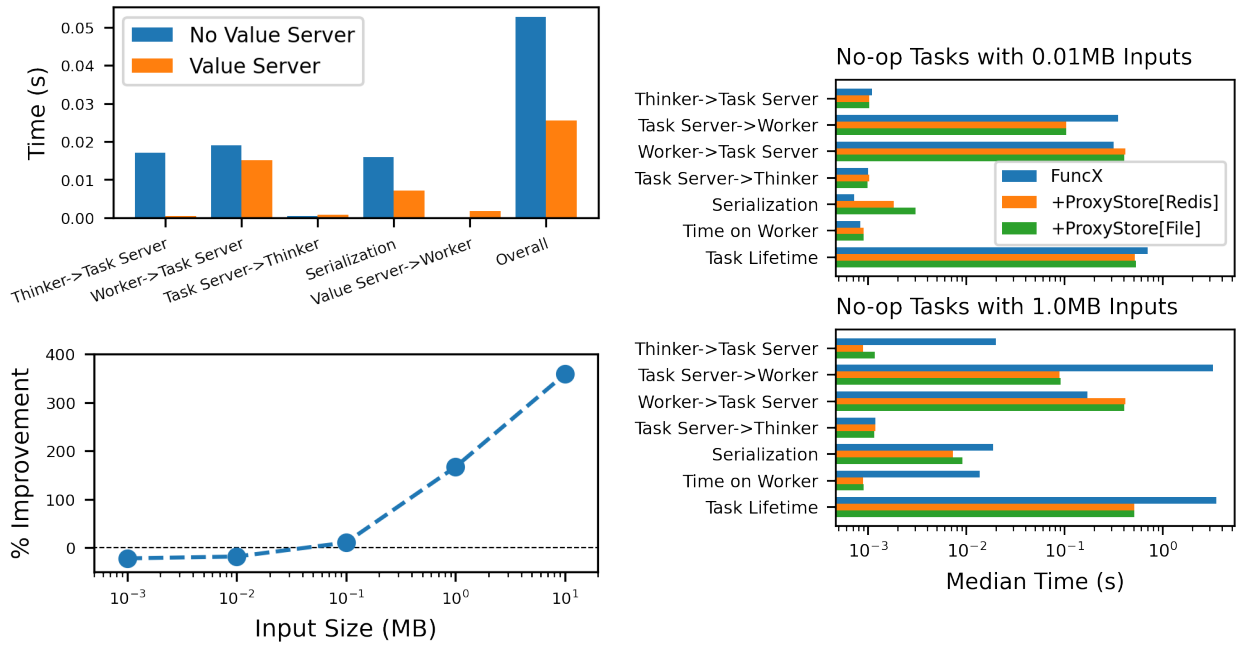


Figure 3.18: (Top Left) Median per-task durations for components in the Colmena task life cycle on the Theta cluster at ALCF, with and without the ProxyStore (i.e., the "Value Server"), as measured for a synthetic application with eight workers, zero-length tasks, 1 MB inputs, and 0 B outputs. ProxyStore reduces time spent serializing, communicating, and deserializing task data. (Bottom Left) Percent reduction in synthetic application overheads for a similar configuration on Theta, with and without ProxyStore, as a function of input data size. ProxyStore yields performance benefits when task inputs are larger than around $O(10)$ kB. (Right) Break-down of median times for different components of the end-to-end execution of a no-op task with Colmena using the Globus Compute (previously called "funcX") task server. ProxyStore reduces communication costs for both small (10 kB) and large (1 MB) task inputs by avoiding repeated serializations and deserializations of object transmitted through the task server and Globus Compute cloud service. Figures from Ward et al. 2021 and 2023.

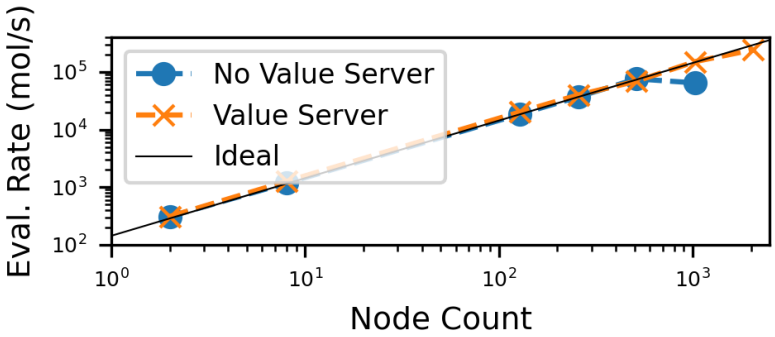


Figure 3.19: Machine learning inference task performance (molecule evaluations per second) on Theta versus number of nodes (one worker per node). The inference rate is measured starting from the time the first worker begins computation (i.e., after loading libraries) to when all inference tasks have completed. ProxyStore reduces communication overheads in the Colmena task server which enables linear scaling at 2048 nodes. Without ProxyStore, performance degrades after 512 nodes. Figure from Ward et al. 2021.

To further study how the benefits of ProxyStore vary with input size I, we repeat the experiments of Figure 3.18 (top) but while varying I from 1 KB to 10 MB. The results, shown in Figure 3.18 (bottom left) as percentage improvement in communication overhead time *with* ProxyStore relative to the time *without*, show that for small inputs (<10 KB), the additional cost of communicating data via ProxyStore is larger than the cost of passing the input data through the task server—but that as the input size increases, the cost of passing input data through the task server increases rapidly and ProxyStore yields large improvements.

Next we compare the performance achieved when communicating task inputs with Globus Compute (referred to as "funcX" in the figure, the former name for Globus Compute) and two ProxyStore backends: shared file system and Redis. We uses the synthetic app to execute no-op tasks that return no output to measure task overheads. We perform the experiment with 10 kB and 1 MB inputs. We chose the input sizes based on the characteristics of Globus Compute which stores function arguments and results smaller than 20 kB in an Amazon ElastiCache Redis store and objects greater than 20 kB in Amazon S3. The thinker and task server are located on a Theta login node, and we use a Globus Compute endpoint

on Theta which executes tasks on a single Theta KNL node. We execute 50 tasks and record the time spent in different stages of the task's life cycle.

We show in Figure 3.18 (right) communication times between the thinker, task server, and worker, as well as the serialization time, time spent on the worker, and overall task lifetime. *Serialization* time is that spent serializing and deserializing tasks on the thinker, task server, and worker. When serializing a task, Colmena scans for task inputs or outputs with sizes exceeding the ProxyStore threshold (set to zero for this experiment). If such large objects are found, the object is proxied and the lightweight proxy is serialized along with the task instead. Therefore, the serialization time reflects proxying time, which includes time spent communicating objects to Redis or writing objects to disk. *Time on worker* is the time between the task starting on the worker and the worker returning the completed task; it includes deserialization of the task, possible resolving of proxies, the execution of the task itself (which in this case is a no-op), and the serialization of the results. *Task lifetime* is the time between a task being created by the Thinker and the result being received by the Thinker.

Task server-to-worker communication dominates the overall task lifetime because inputs must go through the Globus Compute cloud service. Passing objects via proxies can reduce this cost by 2–3× for 10 kB inputs and up to 10× for 1 MB inputs. Similar magnitude speedups are found for the communication between the thinker and task server with larger objects. The thinker and task server communicate via Redis queues so sending small objects (e.g., 10 kB) via ProxyStore's Redis connector performs similar to without ProxyStore, but larger objects see significant gains. The task server, upon receiving a task from the thinker, deserializes the task to determine the endpoint the task needs to be executed on and then serializes the task again to send to Globus Compute. ProxyStore avoids additional deserialization and serialization of the input data because the data are replaced by a lightweight proxy.

Last, in Figure 3.19, we validate the results of these synthetic evaluations through scaling ML inference within a molecular design workflow on Theta. At 1024 nodes, the task server fails to keep workers fed with data for the relatively short ML inference tasks; ProxyStore, on the other hand, maintains scaling performance at 2048 nodes.

3.6.2 Dask

Dask [219] is a parallel computing library in Python that enables efficient parallel computations on large datasets by breaking them down into smaller, manageable tasks. With 12.3k stars on GitHub [81] and 4.9M downloads per month in September 2024 [82], Dask is a *de facto* standard for numerical workflows. Dask Distributed extends the Dask and Python `concurrent.futures` APIs to provide a lightweight and easy-to-use library for distributed computing. A centralized scheduler manages the dynamic execution of tasks across local cores or multiple nodes in a cluster and is optimized for low-latency task dispatching, spending between a 100 μs and 1 ms on each task. However, this overhead can drastically increase when the graph of a task is large, such as when task parameters are large or complex. Large task graphs can incur significant I/O overheads in the scheduler for serialization, communication, and deserialization of messages.

When the client submits a task to the scheduler, the function code and arguments that make up the task are serialized and then encoded into a request with MessagePack, a self-describing semi-structured format. Once the scheduler receives the request message, a worker is selected based on heuristics and user-defined requirements and the task is forwarded to the worker. The worker executes the task and returns to the scheduler a message containing the size of the task result; the actual result stays on the worker. The scheduler communicates a message to clients that the task has completed and any futures to the result of the task are ready.

Dask provides mechanisms to optimize data transfer: (1) array-like data can be scat-

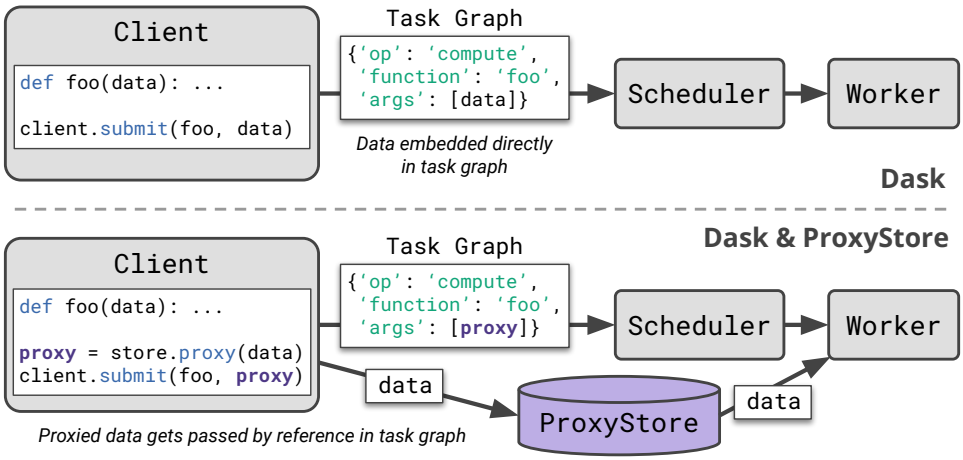


Figure 3.20: Pass-by-proxy semantics reduce data flow through the Dask scheduler without altering application behavior.

tered and gathered directly across workers; (2) native interfaces optimize common data operations [80] (e.g., through Dask Arrays, Bags, DataFrames, and Delayed); and (3) objects already located on workers, such as the results of tasks, will be communicated directly between workers rather than through the scheduler. The goal of these solutions is to prefer passing task data by reference rather than embedding data directly in the graph; however, these solutions do not cover all data types or application structures. For example, frequently moving large objects between the client and workers is considered an anti-pattern; Dask prefers that data remain on the worker cluster. Yet, this is a common pattern in scientific applications (e.g., active learning [249, 251]) that is not supported as well by Dask. Applications that create large Python objects not explicitly optimized for by Dask will see severely degraded performance due to serialization and scheduler overheads. Notably, these large Python objects are pickled as a byte-string then combined into a message with MessagePack, but MessagePack is slow with large byte-strings and can only support task graphs up to 4 GB in size.

ProxyStore can alleviate data transfer overheads in Dask by proxying large task objects instead of embedding them directly in the task graph (Figure 3.20). Importantly, use of ProxyStore does not require modification to task code and is not mutually exclusive with

Dask optimization options. Here, we discuss three methods for integrating ProxyStore into Dask applications. Our experiences integrating Dask and ProxyStore required overcoming many technical challenges overcome to ensure compatibility and performance; these challenges motivated the addition of many features aforementioned in ProxyStore, including property caching on `Proxy` instances, custom serializers for common data types, and the mypy plugin. The result is a robust and easy-to-use solution for building sophisticated computational science workflows, which we demonstrate through synthetic performance evaluations and real-world applications using TaPS.

We outline three methods, exemplified in Figure 3.21, to integrate ProxyStore into a Dask application: (1) manually proxy objects using ProxyStore's existing tooling, (2) use our custom Dask client to automatically proxy objects; or (3) use our custom executor interface to intelligently proxy objects and manage memory. For simple applications, the manual approach works well, but it can require significant code changes in more sophisticated applications. The custom client provides a drop-in replacement for existing Dask applications. The `StoreExecutor`, which extends Python's `concurrent.futures` interface, is the most powerful approach: it is compatible with many other executor client types, such as those provided by Parsl and TaskVine; custom policies can be defined to determine what objects to automatically proxy and, when combined with ProxyStore's `MultiConnector`, what mediated storage option to use; and it uses ProxyStore's ownership model (introduced later in Section 4.3.3), inspired by Rust's ownership and borrowing semantics, to perform safe and automatic memory management of proxies.

We used TaPS to evaluate the performance benefits of using ProxyStore within Dask applications. Experiments were performed using the Sunspot system at the Argonne Leadership Computing Facility, a pre-production supercomputer with early versions of the Aurora software development kit. Sunspot has 128 nodes interconnected by an HPE Slingshot 11 network and a high-performance DAOS storage system. Each node contains two Intel Xeon

```python
from dask.distributed import Client
from proxystore.ex.connectors.daos import DAOSConnector
from proxystore.store import Store

client = Client()
connector = DAOSConnector(pool=..., container=...)

with Store('example', connector) as store:
    proxy = store.proxy([1, 2, 3])
    future = client.submit(sum, proxy)
    assert future.result() == 6
```

(a) A proxy can be manually created via the `Store` interface and passed directly to tasks in place of the actual object.

```python
from proxystore.ex.plugins.distributed import Client
from proxystore.ex.connectors.daos import DAOSConnector
from proxystore.store import Store

connector = DAOSConnector(pool=..., container=...)

with Store('example', connector) as store:
    client = Client(ps_store=store, ps_threshold=1000)
    future = client.submit(sum, [1, 2, 3])
    assert future.result() == 6
```

(b) The custom Dask Distributed `Client` will automatically proxy task input and output objects larger than a user-defined threshold (e.g., 1 kB).

```python
import sys
from dask.distributed import Client
from proxystore.ex.connectors.daos import DAOSConnector
from proxystore.store import Store
from proxystore.store.executor import StoreExecutor

client = Client()
connector = DAOSConnector(pool=..., container=...)

with StoreExecutor(
    client,
    store=Store('example', connector),
    should_proxy=lambda x: sys.getsizeof(x) >= 1000,
) as executor:
    future = executor.submit(sum, [1, 2, 3])
    assert future.result() == 6
```

(c) The `StoreExecutor` can combine a `Store` and Dask `Client` and supports custom policies for what objects should be automatically proxied (here, objects larger than 1 kB) and automatically manages proxy lifetimes.

Figure 3.21: ProxyStore is easily compatible with existing applications. Here we demonstrate the three integration patterns. The `DAOSConnector` is used, but this specific connector can be exchanged depending on the application requirements and execution environment.

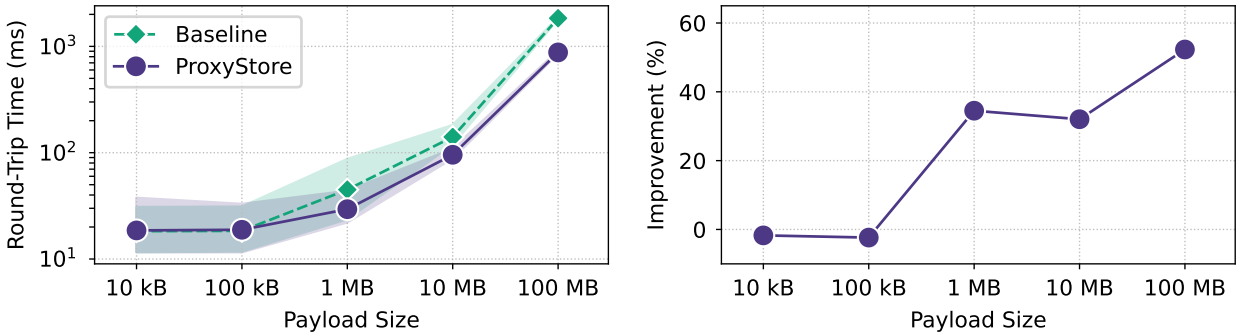


Figure 3.22: (Left) No-op task round-trip time with various payload sizes. (Right) Relative improvement in round-trip time compared to the baseline when using ProxyStore.

Max CPUs with 52 physical cores, 64 GB of high-bandwidth memory, 128 GB of DDR5 memory per CPU, and six Intel Data Center Max GPUs. We used Python 3.11, Dask Distributed 2024.7.1, ProxyStore 0.7.1, ProxyStore Extensions v0.1.4, and TaPS 0.2.1. We performed ProxyStore experiments using Redis due to DAOS outages on Sunspot at the time of writing; a Redis server was started on the rank 0 node of each batch job. Configuring ProxyStore to use DAOS would be even easier, as described in Section 3.3.1, and we expect comparable performance outcomes due to DAOS leveraging NVMe storage distributed throughout the racks of the cluster.

Overheads: We measure the round-trip time of no-op tasks with payloads of varying sizes in Figure 3.22. This experiment represents a worst-case scenario for the Dask scheduler: all data is sent between the client and workers and no data is reused across multiple tasks. Using ProxyStore's pass-by-proxy model improves round-trip time for larger task payloads ($>$ 100 kB) by up to 50%. This improvement is attributed to (1) smaller messages to be serialized and communicated, (2) less data transferred through the scheduler, and (3) improvements to ProxyStore's serialization that reduce memory copies.

Scaling: We measure task throughput with and without ProxyStore as a function of the number of Dask workers n. Each node hosts up to 104 workers, the number of physical cores per node. We execute 10 000 tasks that consume and produce 1 MB of random data

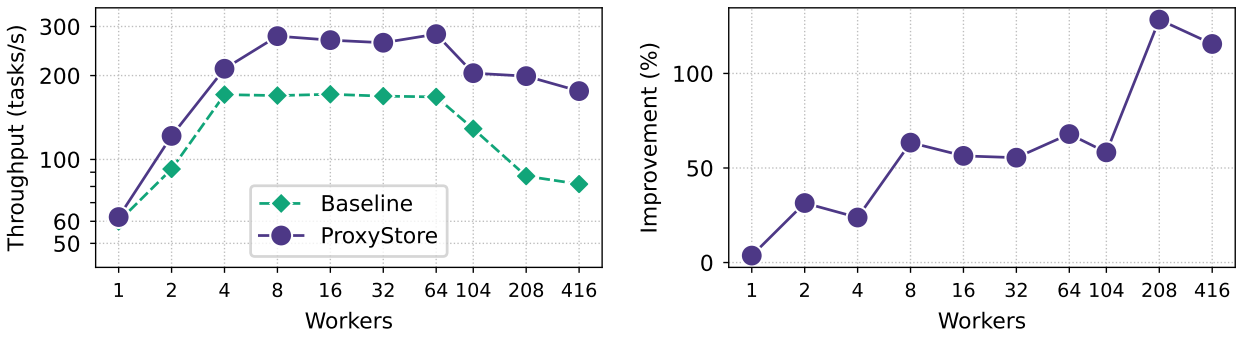


Figure 3.23: (Left) No-op task throughput with various worker counts. Tasks consume and produce 1 MB of random data. (Right) Relative improvement in throughput compared to the baseline when using ProxyStore. ProxyStore alleviates data flow burdens from the Dask scheduler, enabling the scheduler to dispatch tasks faster.

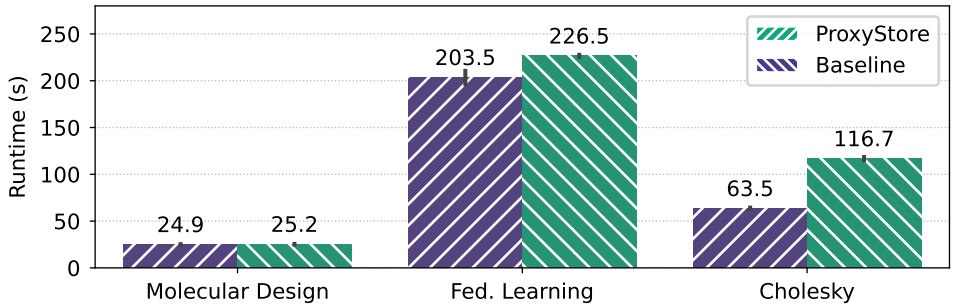


Figure 3.24: ProxyStore can reduce Dask overheads applications that embed large objects in the task graph, such as the Cholesky decomposition example and federated learning simulation provided by TaPS.

(chosen based on the results in Figure 3.22). An initial batch of n tasks are submitted; as current tasks complete, new tasks are submitted until all tasks are finished. Tasks are essentially no-ops besides the result data generation which takes only $O(1)$ ms; thus, the goal of this experiment is to stress the Dask scheduler and understand its limits. As depicted in Figure 3.23, task throughput with Dask quickly plateaus around 170 tasks per second and degrades when utilizing 104 workers. Use of ProxyStore alleviates data transfer burdens from the scheduler, enabling higher sustained throughput; however we still observe the same drop in performance at 104 workers which may indicate the presence of performance limitations in the Dask scheduler that are independent of data volume.

Applications: We use three reference applications from TaPS representing an array

of data patterns. Cholesky decomposition has short tasks that consume and produce large arrays, federated learning has long tasks that consume and produce large models, and molecular design has short tasks that consume and produce small datasets and models. We chose these three applications because they are implemented in a manner which accentuates data transfer between the client and workers. As demonstrated in Figure 3.24, ProxyStore yields the greatest benefits to Dask applications with larger tasks payloads and shorter running tasks—applications where task overheads represent a larger proportion of overall runtime.

3.6.3 Flight

Flight (Federated Learning In General Hierarchical Topologies) is a hierarchical federated learning framework for distributed environments [138]. Typical federated learning frameworks assume simple two-tier network topologies where edge devices are directly connected to a centralized aggregation server. While this is a practical mental model, it does not exploit the inherent topology of real-world distributed systems like the Internet-of-Things. Flight can construct arbitrary hierarchies of aggregators and workers across arbitrary compute resources using function-as-a-service via Globus Compute and peer-to-peer data transfer via PS-endpoints, as shown in Figure 3.25.

The purpose of hierarchical federated learning is to improve performance by distributing model aggregation to different locations. Thus, it would be both inefficient and costly if all data had to pass through the top-level coordinator (the root of the device hierarchy) rather than be passed directly between the participating entities. In addition, Globus Compute, which Flight builds upon, imposes a 5 MB payload limit which prohibits the transfer of even modestly-sized model parameters.

When defining a hierarchy in Flight, a PS-endpoint can be associated with each Globus Compute endpoint. The framework will then automatically proxy model parameters dis-

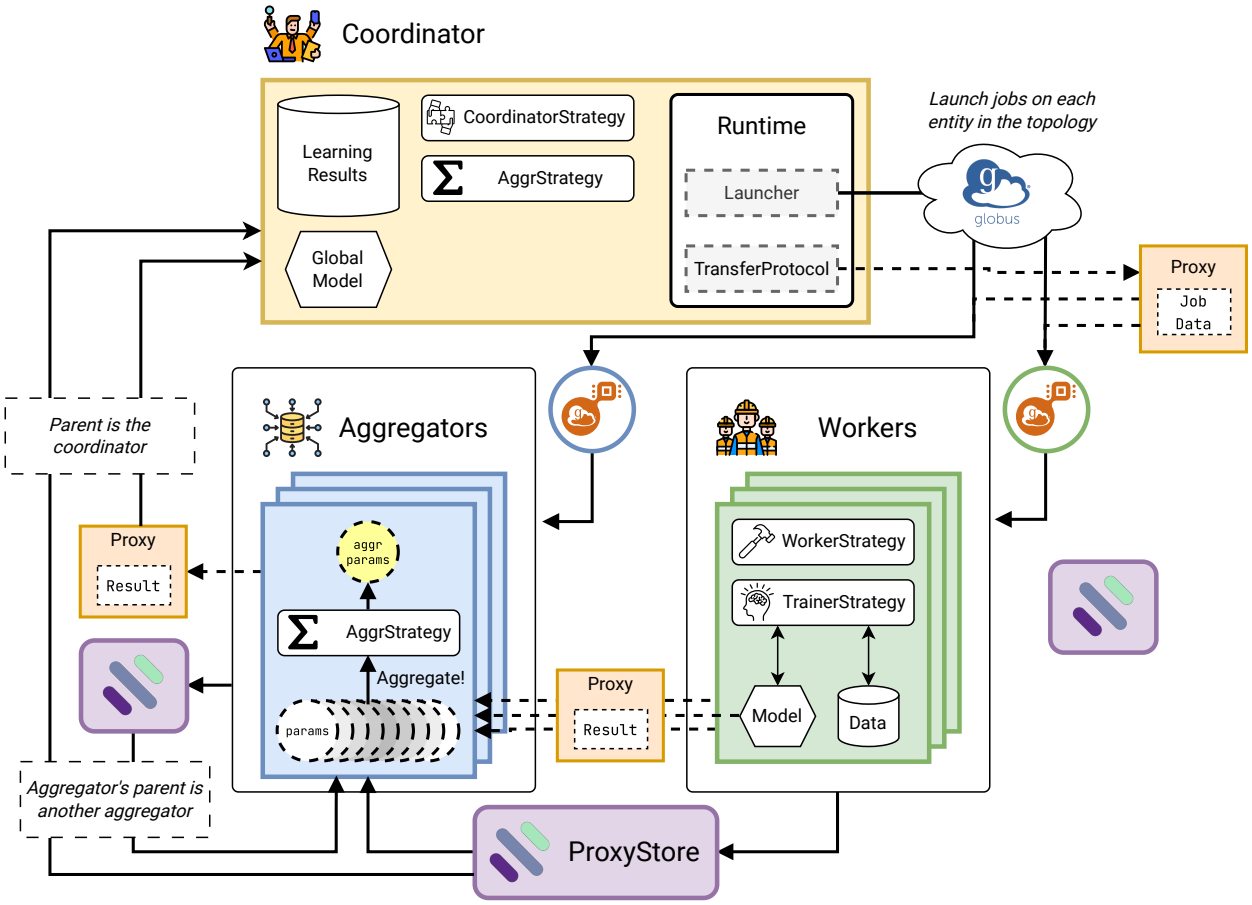


Figure 3.25: High-level view of the Flight architecture. A Coordinator launches jobs to be run on Aggregators and Workers via Globus Compute, while data (e.g., model parameters) are transferred through ProxyStore. Each Worker trains its local copy of the model and sends back its locally-updated model to its parent (either the Coordinator or an Aggregator). Each Aggregator aggregates the responses of its children (Workers and other Aggregators alike). The Coordinator facilitates the entire process.

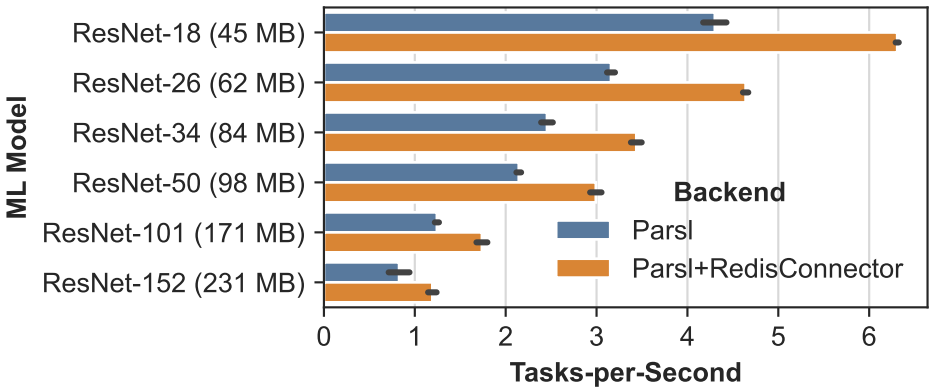


Figure 3.26: No-op inference tasks per second in Flight using Parsl to launch tasks on workers. Use of ProxyStore's Redis connector reduces transfer overheads and improves task throughput.

tributed to workers or accumulated by aggregators. While the intended use case for the ProxyStore integration is to enable peer-to-peer transfer between workers on Globus Compute endpoints, any ProxyStore connector can be used.

Figure 3.26 measures no-op training tasks per second where tasks are submitted using Parsl. We compare data transfer with Parsl to Parsl and ProxyStore (configured with the Redis connector). Out-of-band data transfer of model weights via ProxyStore reduces overheads in Flight, enabling higher task throughput.

In Hudson et al. [138], we demonstrate that ProxyStore is necessary to achieve competitive performance with state-of-the-art federated learning frameworks (namely, Flower [37]). Without ProxyStore, as demonstrated in Figure 3.27, Parsl does not scale as well as Flower for larger model sizes (i.e., ResNet-18, ResNet-50, and ResNet-152), whereas Flight performs comparably to, if not better than, Flower when using ProxyStore's Redis connector.

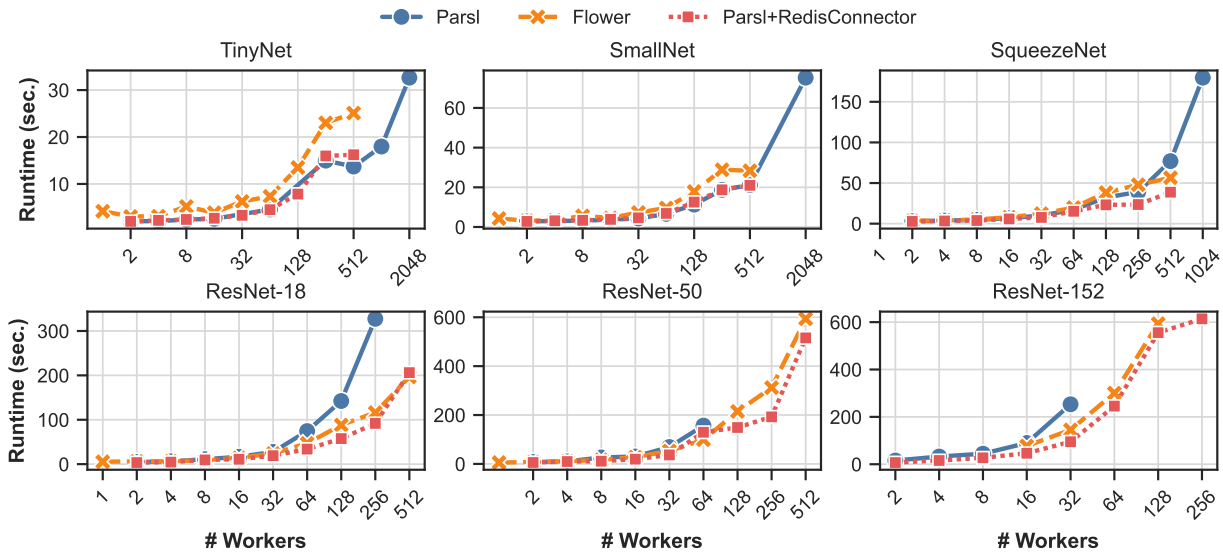


Figure 3.27: Weak scaling results comparing the runtime of Flower and Flight using Parsl and Parsl + ProxyStore's Redis connector for a series of increasingly complex models. Flight provides better performance and, in some cases, also scales to more workers when leveraging ProxyStore. Figure from Hudson et al. 2024.

3.7 Summary

ProxyStore is a novel framework for facilitating wide-area data management in distributed applications. The proxy model provides a pass-by-reference-like interface that can work across processes, machines, and sites, and enables data producers to change communication methods dynamically without altering application behavior. ProxyStore provides a suite of communication channel implementations intended to meet most requirements and can be extended to other communication methods. We demonstrated the use of ProxyStore with FaaS and workflow systems, synthetic benchmarks, and real-world scientific applications. We showed that ProxyStore can accelerate a diverse range of distributed applications and enables comparable performance to alternative approaches while avoiding the cumbersome code changes and/or manual deployment and configurations required by alternatives.

In Chapter 4, we investigate support for data flow semantics on proxies, so that readers of an object block until the object is written, as in Id [187]; high-throughput streaming

with proxies; and alternatives to wide-area reference counting for automatic object eviction. Other unexplored avenues for investigation include support for more communication methods, advanced data management policies for persistence and replication, extension to other programming languages, and optimizations such as intelligent prefetching and faster peer-to-peer networking protocols. Kamatar et al. [144] applied the proxy model to enable lazy library loading in HPC environments to address other common problems in distributed computing. Our work here aims to encourage further research in data fabrics for federated applications, and to enable scientists and engineers to more easily design sophisticated distributed applications.

CHAPTER 4

ADVANCED DATA FLOW PATTERNS FOR DISTRIBUTED APPLICATIONS

Task-based programming paradigms, such as function-as-a-service (FaaS) and workflows, have emerged as vital methods for achieving computational flexibility and scalability. Applications are written as compositions of many distinct components, referred to as *tasks*, and FaaS platforms and workflow systems, collectively referred to as *execution engines*, abstract the complexities of executing tasks in parallel, whether across personal, cloud, edge, and/or high-performance computing (HPC) systems [219, 176, 123, 17, 27, 61]. Such execution engines have enabled a wide variety of innovative applications.

Yet as the scale and ambition of task-parallel applications grows, they increasingly encounter difficulties due to the use of shared storage for the exchange of intermediate data among tasks—an approach commonly employed both by workflow systems (e.g., Parsl [27], Pegasus [86], Swift [256]) and cloud-hosted FaaS systems (e.g., AWS Lambda [17], Azure Functions [50], Google Cloud Function [123]). Such uses of shared storage can fail or become prohibitively expensive as the number of tasks, the geographic distribution of tasks, the quantities of data exchanged, and the required speed of data exchange grow.

Many researchers have investigated alternative mechanisms for distributed and wide-area data management that circumvent these limitations of shared storage. For example, Linda's tuple space model provides unified access to a shared distributed memory space [8], DataSpaces provides a similar model for large-scale applications [90, 11], and peer-to-peer systems like the InterPlanetary File System provide decentralized content-addressed file sharing [34]. Another approach to simplifying data sharing is the *object proxy paradigm*, which provides transparent access and management for shared objects in distributed settings. This mechanism, long used with Java's Remote Method Invocation (RMI) [40], is also supported in Python via the ProxyStore system [195]. ProxyStore's transparent object proxies pro-

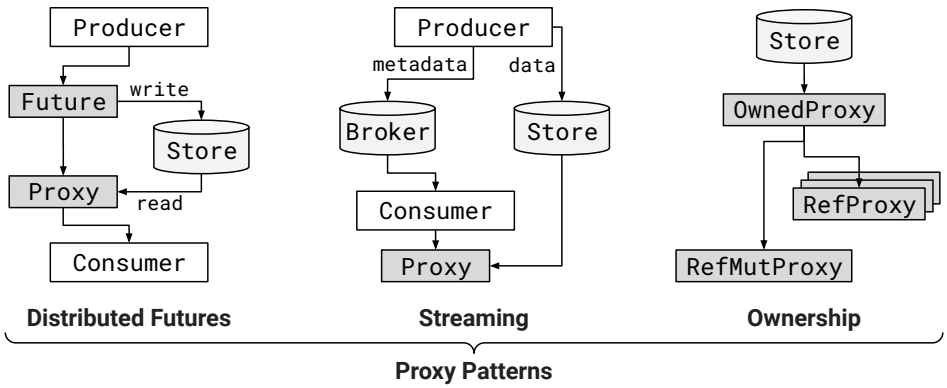


Figure 4.1: Overview of the three proxy-based data flow patterns we design.

vide lightweight, wide-area references to objects in arbitrary data stores—references that can be communicated cheaply and resolved just-in-time via performant bulk transfer methods in a manner that is transparent to the consumer code. Recent work has shown how by decoupling data flow complexities from control flow-optimized execution engines [249, 129, 70, 88, 196], the object proxy paradigm can simplify implementations of dynamic application structures such as ML model training and inference.

Yet the object proxy paradigm remains a *low-level* abstraction that can be hard to use in practice due to the complexities inherent in managing many references to remote objects. Thus, we ask: Can we identify common *high-level patterns* that build on the proxy model to accelerate and simplify development of advanced applications? To this end, we review in this chapter three computational science applications previously developed for workflow execution engines (*1000 Genomes*, *DeepDriveMD*, *MOF Generation*), identify limitations in the data flow patterns supported by those execution engines, and propose three new programming patterns that extend the proxy model to overcome these limitations (Figure 4.1):

- A *distributed futures* system for seamless injection of data flow dependencies into arbitrary compute tasks to overlap computation and communication;
- An *object streaming* interface that decouples event notifications from bulk data transfer such that data producers can unilaterally determine optimal transfer methods; and

- An *ownership* model that provides client-side mechanisms for managing object lifetimes and preventing data races in distributed task-based workflows.

Each pattern simplifies building sophisticated task-based applications that are to execute across distributed or remote compute resources (e.g., using FaaS or workflow systems). For each, we discuss its requirements and the protocols used to support it. Our reference implementations extend ProxyStore to leverage the existing low-level proxy model within Python, a popular and pervasive language for task-based distributed applications. The implementations are available within ProxyStore v0.6.5 and later, available on GitHub [204] and PyPI [205]. We evaluate our reference implementation for each pattern using (1) synthetic benchmarks across various FaaS and workflow systems and (2) our motivating applications, for which we reduce workflow makespan by 36% in 1000 Genomes, improve inference latency by 32% in DeepDriveMD, and optimize proxy lifetimes in MOF Generation.

The rest of this chapter is as follows: Section 4.1 introduces our motivating scientific applications; Section 4.2 summarizes ProxyStore, presented in Chapter 3; Section 4.3 outlines the design and implementation of each pattern; Section 4.4 demonstrates synthetic evaluations; Section 4.5 presents our experiences applying these patterns to our motivating applications; Section 4.6 provides context about related work; and Section 4.7 summarizes our contributions and future directions.

4.1 Motivating Applications

1000 Genomes: This bioinformatics pipeline [2] identifies mutational overlaps within the 2504 human genomes sequenced by the 1000 Genomes Project [1]. It comprises five stages: (1) fetch files, each containing all Single Nucleotide Polymorphisms (SNPs) in a chromosome, chunk, and process them in parallel to extract SNP variants by individual; (2) merge individuals' results of the prior stage; (3) score and select SNP variants based on their phenotypic effect; (4) compute overlap of selected SNP variants among pairs of individuals and by

chromosome; and (5) compute frequency of overlapping variants. Executing scientific workflows in a FaaS setting may be preferred when access to specialized hardware, such as AI or quantum accelerators, or the ability to rapidly scale up or down is required, but workflow execution on a FaaS system poses challenges because FaaS systems rely on control flow to determine when to submit tasks. From the application perspective, however, the availability of data—the data flow—is the condition upon which tasks can be submitted. We use the 1000 Genomes workflow as an example of the challenges that arise when executing data flow oriented applications on control flow-optimized systems.

DeepDriveMD: Molecular dynamics (MD) simulation acts as a computational microscope [95] to enable the study of complex biomolecular systems [53, 91, 237, 202]. However, many important phenomena are difficult to sample using conventional MD, even with powerful supercomputers [137]. DeepDriveMD [159, 46] implements an emerging HPC paradigm in which machine learning (ML) methods are used to track a simulated state space and guide simulations toward a sampling objective. The DeepDriveMD client submits discrete training, inference, and simulation tasks and receives their results. This pattern causes two challenges. First, all data must flow through the client which limits performance at scale (e.g., data volume or task frequency), so a mechanism is needed to alleviate data flow burdens from the client when possible. Second, repeated tasks perform redundant work. For example, each inference tasks loads the latest ML model from disk, infers using the input batch, and compiles the results which will later become the input to a simulation task. This is inefficient because the same model is loaded multiple times across tasks, tasks may execute on different workers negating cache benefits, and every task incurs non-trivial overheads for scheduling and execution.

Metal-Organic Framework (MOF) Generation: This workflow [180] uses molecular diffusion models [193] to generate organic ligands, assemble MOF candidates, and employ physics models to identify candidates best suited for storing CO_2. The workflow uses

Colmena[249]; the Colmena thinker is a central process that determines which tasks to execute, and with what parameters. A core computational challenge is ensuring that the thinker has timely data, such as the latest diffusion model results, when deciding the next task. Object proxies have been used to improve thinker response time in similar applications [249, 251] (as described in Section 3.6.1), but knowing the lifetime of proxied data is challenging in sophisticated workflows where the types of tasks to be executed are not know ahead of time.

4.2 Background

This section summarizes Chapter 3.

In software design, a *proxy* is an object that functions as an interface to another object [114]. A simple proxy will forward operations on itself to the real or *target* object, but often a proxy is used to provide extra functionality such as caching or access control, in addition to forwarding operations [195]. For example, distributed applications can use a proxy to invoke methods on a remote object, and data-intensive applications can use a virtual or *lazy* proxy which will perform just-in-time resolution of large objects (i.e., load the object from a remote location into local memory when first needed).

Lazy transparent object proxies can be used to communicate objects efficiently in distributed applications [195]. Here, a proxy refers to a target object stored in an arbitrary mediated communication medium (e.g., an object store, database, file system). The proxy forwards all operations on itself to the target, but importantly is totally transparent in that the proxy is an instance of the same type as the target. In Python, this means that `isinstance(p, type(t))` is true for a proxy p and its target t. The proxy is lazy in that it performs just-in-time resolution of the target. The target is not copied from the mediated storage into local memory until an operation is invoked on the proxy. This proxy paradigm has both pass-by-reference and pass-by-value semantics; unused copies of the target object are not made when the proxy is passed between processes but the actual consumer of the

proxy is given a copy.

The benefits of moving data via proxies are numerous: pass-by-reference reduces transfer overheads, no external information is required to resolve a proxy, shims or wrapper functions are eliminated, just-in-time resolution amortizes communication costs and avoids costs associated with unused objects, and proxies enable automatic access control. As such, this paradigm has been used to build a diverse suite of robust and scalable scientific applications [249, 267, 251, 70, 129, 144, 195, 88].

ProxyStore [195] implements this proxy paradigm which we use as the basis for our patterns' reference implementations. ProxyStore defines the *factory*, *connector*, and *store* constructs. The *factory* is a callable object that returns the target object when invoked. ProxyStore creates a unique factory for each target object containing the metadata and logic necessary to retrieve the target from a remote location. This factory is used to initialize a proxy, and a proxy is *resolved* once it has invoked its factory to retrieve and cache the target locally.

The *connector* is a protocol that defines the low-level interface to a mediated communication channel. A mediated channel is one where the communication between a producer and consumer is indirect, such as via a storage system [73]. This indirection is important because the process that creates a proxy and the process that resolves a proxy may not be active at the same time, in which case they could not communicate via direct mechanisms. ProxyStore provides many connectors, including interfaces to external mediated channels such as shared file systems, object stores (Redis [218] and KeyDB [229]), and peer-to-peer transfer systems (Globus Transfer [106, 59] and ProxyStore Endpoints [195]) and bespoke mediated channels that can leverage high-performance networks through the UCX [240] and Margo [206] libraries.

The high-level *store* interface, initialized with a connector, is used to create proxies of objects. A proxy p can be created from a target t by calling `Store.proxy(t)`. This

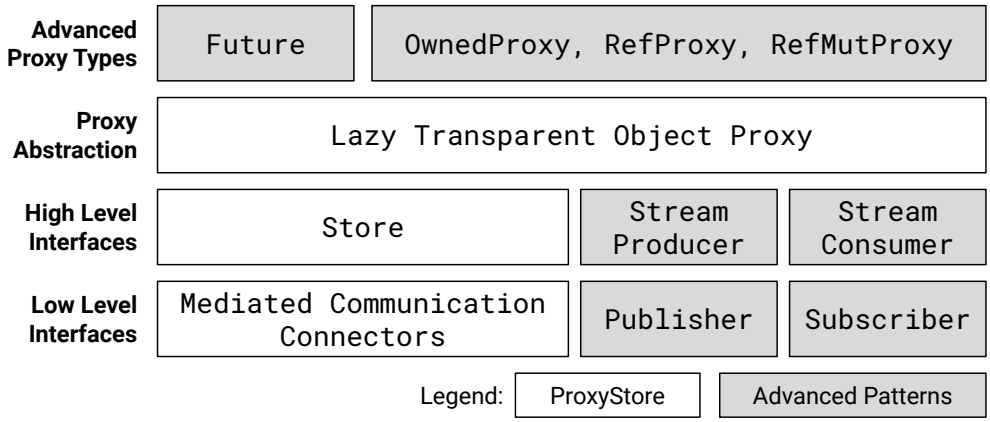


Figure 4.2: Overview of the ProxyStore interface and abstraction stack with the contributions of this chapter included in the shaded boxes.

method (1) serializes t using the default ProxyStore or user-provided serializer, (2) puts the serialized t in the mediated channel via the connector, (3) creates a factory with the appropriate metadata about t and the store/connector used, (4) initializes a proxy with the factory, and (5) returns the proxy. This process incurs some overhead but is trivial for larger objects. Prior work [195, 249] found the performance benefits of proxies to outweigh proxy creation and resolution overhead for objects larger than ∼10 kB; the exact threshold depends on many factors (e.g., connector choice, execution engine).

4.3 Proxy Patterns

We describe the design of each of the three advanced programming patterns that build on the aforementioned distributed object proxy base. We discuss the details of our reference implementations that extend ProxyStore, and Figure 4.2 describes how these patterns fit into the existing ProxyStore stack. These patterns are not mutually exclusive, but we discuss each in isolation for clarity.

4.3.1 Distributed Futures

A future represents a value that will eventually be available; the holder of a future

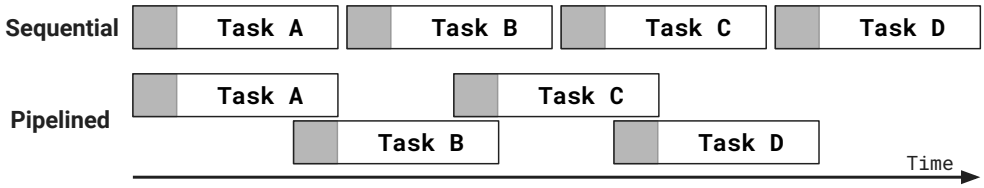


Figure 4.3: Four tasks executed in a sequential (above) or pipelined (below) fashion. Each task produces data needed by the following task. The grey region at the start of each task represents startup overhead before the input data can be used. By enabling a successor task to start before its predecessor has finished, futures enable overlapping of startup overhead with computation, a form of pipelining.

can block on it until the value is resolved. Futures simplify writing non-blocking compute (e.g., remote procedure calls, database queries, or FaaS invocations) and I/O (e.g., network requests or file system reads) operations. Execution engines use futures to represent eventual task results, and this is valuable for representing long running remote execution or assembling applications with asynchronous callbacks. However, the distributed futures provided by execution engines have three key limitations: (1) these futures perform control and data synchronization so data flow cannot be optimized independent of control flow, such as to pipeline task execution as in Figure 4.3; (2) the transfer mechanisms used by the future cannot be optimized based on the type or location of data; and (3) futures produced by execution engines are only usable within the context of that execution engine (e.g., a future from one engine cannot be sent as input to another).

We design a distributed futures system called *ProxyFutures* that (1) supports explicit and implicit usage, arbitrary execution engines, arbitrary distributed memory backends, and seamless injection of data flow dependencies, and (2) addresses a limitation of ProxyStore that a proxy cannot be created before its target object exists. In ProxyFutures, a future f is created for an eventual value x, and f can be used to create any number of proxies p_f.

Consider an application with a main process M, a data producing process P, and a data consuming process C. M dispatches two tasks: T_P to P and T_C to C. T_P is to produce a value x to be consumed by T_C; thus, T_C has a data dependency on T_P. M can create a

```python
from dask.distributed import Client
from proxystore.connectors.foo import FooConnector
from proxystore.store import Store
from proxystore.store.future import Future

def producer(future: Future[str]) -> None:
    future.set_result('value')

def consumer(data: Proxy[str]) -> None:
    assert data is 'value'

with Store('example', FooConnector()) as store:
    client = Client(...)
    future: Future[str] = store.future()

    t1 = client.submit(producer, future)
    t2 = client.submit(consumer, future.proxy())

    t1.result(), t2.result()
```

Figure 4.4: Example usage of the ProxyFuture interface within tasks executed by Dask. A proxy created from a `Future` will block implicitly on the result of the future when needed. This interface abstracts the low-level communication away from the functions which set the result or consume the proxy.

future f and associated proxy p_f, and pass f and p_f to T_P and T_C, respectively. When T_C first resolves p_f, it blocks until T_P has set the result of f. Importantly, T_C can be started before T_P has finished or even started. M, when creating f, can choose the communication method to be used based on where P and C are located and what communication methods are available between them; thus, the detailed communication semantics are abstracted from T_P and T_C. The implicit nature of p_f also means that the code for T_C can be invoked either on a value directly or on a proxy of the value. This equivalence simplifies code and testing and means that M can inject data flow dependencies via a future into arbitrary third-party functions that expect to receive data directly.

We implement this behavior by extending ProxyStore's `Store` interface to expose a `future()` method that returns a `Future` object. The `Future` class exposes two main methods: `set_result(obj: T)`, which sets the result of the future to an object of type `T` and `proxy()`, which returns a `Proxy[T]`. When a proxy created via `Future.proxy()` is resolved, the proxy

blocks until the target value has been set via a call to `Future.set_result()`, as shown in Figure 4.4. Use of ProxyFutures does not affect when a successor task starts; scheduling is still managed by the execution engine and/or user application. ProxyFutures are best integrated at the application level so that developers can optimize task execution per their application requirements and to express more complex data dependencies than typically supported by execution engines.

Internally, communication between a `Future` and any child proxy(s) is handled via the `Store` used to create the `Future`. Thus, a future and associated proxies can be serialized and sent to arbitrary processes on arbitrary machines. In contrast, many standard-library future implementations use non-serializable async, thread, and inter-process synchronization mechanisms (e.g., `std::future` in C++ [49], concurrent and async futures in Python [209]), while RPC-based futures are only resolvable within the RPC framework (Dask futures [219] or Ray `ObjectRefs` [246]). The self-contained properties of the proxy mean that all logic for communication and resolution are embedded within the future and proxy; the future creator chooses communication methods on behalf of the process(es) which might set or consume the result of the future.

4.3.2 Object Streaming

High-performance stream processing applications dispatch remote compute tasks on objects consumed from a stream, but task dispatch can quickly become a bottleneck with high throughput streams [159, 46, 267]. Consider the application in Figure 4.5, where process A is a data generator that streams chunks of data (i.e., arbitrary Python objects) to a dispatcher process B, which for each data chunk dispatches a compute task on a remote process C. Note that while the dispatcher consumes from the stream, it does not need the actual chunk of data; rather, it only needs to know that a chunk is ready (and potentially have access to user-provided metadata) in order to dispatch the task that will actually consume the chunk.

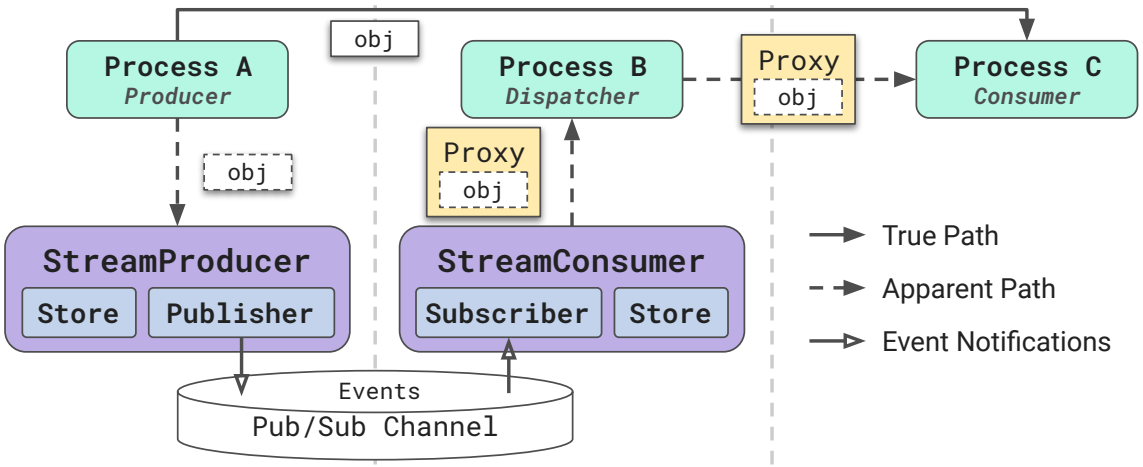


Figure 4.5: The `StreamProducer` abstracts low-level communication details from the `StreamConsumer` and transparently decouples metadata from bulk data transfer. Yielding proxies, rather than objects directly, in the `StreamConsumer` enables just-in-time resolution and pass-by-reference optimizations.

```python
from globus_compute_sdk import Executor
from proxystore.connector.foo import FooConnector
from proxystore.store import Store
from proxystore.stream import StreamProducer, StreamConsumer
from proxystore.stream.shims.kafka import KafkaPublisher, KafkaSubscriber

def producer() -> None:
    store = {'topic': Store(..., FooConnector(...))}
    publisher = KafkaPublisher(...)

    with StreamProducer(publisher, store) as producer:
        for item in ...:
            producer.send('topic', item)

def consumer() -> None:
    subscriber = KafkaSubscriber('topic', ...)

    with StreamConsumer(subscriber) as consumer:
        for item in consumer:
            assert isinstance(item, Proxy)

with Executor('<Endpoint UUID>') as client:
    t1 = client.submit(producer)
    t2 = client.submit(consumer)

    t1.result(), t2.result()
```

Figure 4.6: Example using the ProxyStream interfaces to stream data between two tasks executed remotely using Globus Compute. A Kafka broker is used for metadata and an arbitrary `FooConnector` for bulk data transfer.

We design a system called *ProxyStream* to enable scalable applications of this pattern. At its core, ProxyStream uses a stream of proxies, rather than data chunks. Bulk data are only transmitted between the data generator and the process/node computing on the proxy of the chunk, bypassing the intermediate dispatching process. ProxyStream optimizes for both metadata and bulk data transfer, has broad execution engine compatibility, provides a self-describing data format, and supports various communication modules to take advantage of high-performance networking stacks.

ProxyStream provides two high-level constructs, the `StreamProducer` and `StreamConsumer`, that combine a message stream broker for low-latency event metadata propagation and a mediated communication channel for efficient bulk data transfer. A `StreamProducer` is initialized with a `Publisher` and a ProxyStore `Store`. The `Publisher` defines a protocol for sending event messages to a stream. We provide shims to many popular event streaming systems (Kafka [23], Redis Pub/Sub and Queues [218], ZeroMQ [135]) which implement the `Publisher` protocol. When a new object and optional metadata are sent to the `StreamProducer`, (1) the object is put in the store, (2) a new event containing the user provided metadata and information about where the object is stored is created, and (3) the event is published via the `Publisher`.

A `StreamConsumer` is initialized with a `Subscriber`, which, like the `Publisher`, defines a protocol for receiving event messages from a stream (Figure 4.6). The `StreamConsumer` is an iterable object, yielding proxies of objects in the stream until the stream is closed. Calling `next()` on the `StreamConsumer` waits for a new event metadata message via the `Subscriber`, creates a proxy of the object using the event metadata, and returns the proxy to the calling code. This process is efficient because the bulk object data has not been read at this point; rather, this will be delayed until the resolution of the proxy.

This model has many benefits: (1) communication mechanisms are abstracted from the stream consumer, (2) stream objects are resolved only when actually needed (wherever the

proxy is resolved), (3) event message and bulk data transfer are decoupled, allowing the application to better optimize both forms of communication for the given application deployment environment and object characteristics, and (4) it provides a mechanism for implementing stateful actors in a workflow.

The ProxyStream interfaces support any combination of single/multi producer/consumer that is supported by the associated `Publisher` and `Subscriber` implementations. The `StreamProducer` supports mapping different stream topics to `Store` instances, enabling further optimization of communication mechanisms; batching; and plugins for filtering, sampling, and aggregation. The `StreamConsumer` supports plugins for filtering and sampling. ProxyStream is fault-tolerant provided that the broker and communication channel are fault-tolerant.

ProxyStream can be integrated at the application or framework level. Figure 4.6 depicts use of ProxyStream within a Globus Compute application; we integrate ProxyStream within the DeepDriveMD framework for the evaluation in Section 4.5.

4.3.3 Ownership

A limitation of the proxy model is the need to manage explicitly the lifetime of the associated target object. When a proxy is shared with more than one process, it is challenging to know when it is safe to free the target object. A ProxyStore proxy acts like a C/C++ pointer or raw pointer in Rust; thus, one process could prematurely free the target object, causing what is equivalent to a null pointer exception in the other process(es); delay freeing the object causing increased memory usage; or forget to free the object causing a memory leak. ProxyStore provides some guidance on using proxies safely, but ultimately it is up to the programmer to use proxies safely—a situation similar to C pointers.

To address this difficulty, we extend the proxy model with two features not provided by ProxyStore: automatic deletion of objects that have gone out of scope and safe support

for mutating objects. Inspired by Rust's borrowing and ownership semantics, our design works in distributed contexts; provides different proxy types that can represent the owned, reference, and mutable reference types; enforces ownership and borrowing rules at runtime based on a proxy's type; and performs automatic dereferencing, coercion, and deletion.

Rust's borrowing and ownership semantics, designed to ensure memory safety without garbage collection, can inspire a way of thinking about shared objects in distributed environments. Rust defines three *ownership rules*: (1) each value has an owner, (2) there can only be one owner at a time, and (3) a value is deleted when its owner goes out of scope [235]. A reference allows a value to be borrowed without relinquishing ownership. The *reference rules* are (1) at any given time, a value can have either one mutable reference or any number of immutable references and (2) references must always be valid. The Rust compiler enforces these rules, and the language provides data structures for runtime enforcement for more complex scenarios that the compiler cannot reason about.

Applying these rules in a distributed application, such as a computational workflow, can make memory management significantly easier without the need to perform global reference counting. Computations represented as directed acyclic graphs (DAGs) are particularly well suited to this model. As objects move from a parent DAG node to a child node, ownership can either be transferred to the child or the child can be given a borrowed reference. Thus, a node has full information about what operations are safe on objects that it receives. Ownership transfer means that the recipient node has full control over that object; an immutable reference means that the node can only read the object. A mutable reference means that the node has sole access to modify the object, but the node cannot create and share additional references: i.e., it is not allowed to pass a reference to its own child node.

One challenge of this model is knowing when a reference to an object goes out of scope, because this requires communication between the process that owns the object and the process that has a reference. However, in a task-based workflow, it is easy to reason that a

reference passed to a task goes out of scope when the task completes (assuming that the task is well-behaved; an improperly behaved task would be one that, for example, creates and stores a memory-to-memory copy of the reference) and workflow systems already propagate information about task completion.

A second challenge is representing the ownership or borrowing of an object. The Rust compiler and dot operator abstracts much of the nuance of dealing either with objects directly or with their references [221]. In Python, for example, an object `T` could be wrapped in a `Owned[T]`, `Ref[T]`, and `RefMut[T]`, in a similar manner to some Rust constructs. However, use of these constructs would be cumbersome, as all referencing, dereferencing, or coercion would have to be done manually.

The transparent object proxy is well-suited to solve these object scope and reference representation problems. An object that is proxied by a process becomes a shared object that is stored on some global object store accessible by all processes in the distributed environment. The target object is serialized, put in the global store, and an `OwnedProxy` is returned. The `OwnedProxy` contains a reference to the global object and, if the proxy has been resolved, a local copy of the object upon which the proxy forwards operations to.

An `OwnedProxy` enforces the following rules [c.f. Rust's ownership rules]: (1) each object in the global store has an associated `OwnedProxy`, (2) there can only be one `OwnedProxy` for any object in the global store, and (3) when `OwnedProxy` goes out of scope, the object is removed from the global store.

When invoking a task on an `OwnedProxy` (i.e., calling a local or remote function), the caller can do one of four things:

- Yield ownership by passing the `OwnedProxy` to the task.
- Clone `OwnedProxy` and pass the cloned `OwnedProxy` to the task. Cloning an `OwnedProxy` will create a new copy of the object in the global store that will be owned by the callee task while the caller still owns the original object.

- Make a `RefProxy` and pass the `RefProxy` to the task. The caller still retains ownership, and the task can only read the object via the `RefProxy`. The callee task can only mutate its local copy, not the global copy. The caller's `OwnedProxy`, used to create the `RefProxy`, keeps track of the references that it has created. Any number of tasks can be invoked on a `RefProxy` at a time.

- Make a `RefMutProxy` and pass the `RefMutProxy` to the task. The caller still retains ownership (essentially the privilege to delete), but the callee task now has sole access to modify the object in the global store. The caller's `OwnedProxy` marks that it has created a `RefMutProxy` and thus cannot mutate itself until the callee task that has the `RefMutProxy` completes. Only one task can be invoked on a `RefMutProxy` at a time and a `RefMutProxy` and `RefProxy` cannot exist at the same time.

The lifetimes of a `RefProxy` and `RefMutProxy` are strongly coupled to those of the tasks they are passed to. Any violation of these rules, such as an `OwnedProxy` that goes out of scope or is deleted while a `RefProxy` or `RefMutProxy` exists, will raise a runtime error. It is also possible to extend a static code analysis tool to verify correctness prior to execution.

Execution engines typically use futures to encapsulate the asynchronous execution of a task. Thus, we use callbacks on the task result futures to indicate that the references associated with a task have gone out of scope. The primary limitation of this approach is that each execution engine has a different syntax for submitting a task and getting back a future. Rather than modify each engine, we provide a set of shims that appropriately parse task inputs and construct a callback on the task's future that will propagate the necessary information about references going out of scope. An example is provided in Figure 4.7.

The `StoreExecutor`, an interface provided by ProxyStore, wraps an execution engine client (e.g., a Globus Compute, Dask, or Parsl client) and automatically proxies task parameters and results based on user-defined policies and manages references associated with tasks [198]. The `StoreExecutor` is easy to use, as demonstrated in Figure 4.8, but

```python
from concurrent.futures import ProcessPoolExecutor
from proxystore.store.base import Store
from proxystore.store.ref import borrow
from proxystore.store.scopes import submit

with Store(...) as store, ProcessPoolExecutor() as pool:
    proxy = store.owned_proxy('value')
    borrowed = borrow(proxy)

    future = submit(pool.submit, args=(sum, borrowed))
    assert future.result() == 6
    # Task is completed so the owned proxy is no longer borrowed

    # Owned proxy is safe to delete or be garbage collected
    del proxy
```

Figure 4.7: Example of creating an `OwnedProxy` and borrowing to pass to a task executed within a `ProcessPoolExecutor`. Here, references must be manually managed. The `StoreExecutor`, shown in Figure 4.8, simplifies this process.

```python
from concurrent.futures import ProcessPoolExecutor
from proxystore.proxy import Proxy
from proxystore.store import Store
from proxystore.store.executor import StoreExecutor, ProxyType

with StoreExecutor(
    ProcessPoolExecutor(),
    store=Store(...),
    ownership=True,
    # Only proxy objects of type list
    should_proxy=ProxyType(list),
) as executor:
    future = executor.submit(sum, [1, 2, 3])
    result = future.result()

    assert isinstance(result, Proxy)
    assert result == 6
```

Figure 4.8: Example usage of the `StoreExecutor`, which automatically manages proxying task arguments and results and the management of proxy lifetimes.

```
class Store(Generic[Connector]):
    def owned_proxy(obj, ...) -> OwnedProxy: ...

def into_owned(Proxy) -> OwnedProxy: ...
def borrow(OwnedProxy) -> RefProxy: ...
def mut_borrow(OwnedProxy) -> RefMutProxy: ...
def clone(OwnedProxy) -> OwnedProxy: ...
def update(OwnedProxy | RefMutProxy) -> None: ...
```

Figure 4.9: Proxy ownership model interfaces and functions. Functions are preferred over methods on the associated proxy reference types to prevent unintentionally clobbering a method of the same name on the target object.

applications requiring more fine-grain control can use the lower-level API in Figure 4.9.

The ownership model is not fault-tolerant when the client crashes in a manner which prevents garbage collection, but the model is compatible with fault-tolerant execution engines such as those that automatically rerun tasks on failure. Since only a single `RefMutProxy` can exist, the ownership model is not optimal for applications with many concurrent writers to the same object; a database, for example, may be more suitable.

So far, we have constricted ourselves to tasks (i.e., function invocations) as the only region of code over which we can define a lifetime; thus, all references to an object are equal to the lifetime of the single task invoked on that reference. Yet a workflow application may employ more complex lifetimes. For example, a lifetime could be assigned to a set of tasks that are a subgraph of the global DAG, and a programmer might want to define references to global objects that are associated with this custom lifetime. Using proxy references is a valid solution but would require additional code to manage and map references to the scopes contextual to the application.

We provide the `Lifetime` construct, an alternative to proxy references, for managing object lifetimes in more complex scenarios. A lifetime, attached to one or more proxies upon proxy creation, will clean up associated objects once the lifetime has ended. We provide three `Lifetime` types and the API can be extended to implement new types. The context-manager lifetime enables mapping proxy lifetimes to discrete segments of code, the time-leased lifetime

```python
from proxystore.store.base import Store
from proxystore.store.lifetimes import ContextLifetime

with Store(...) as store:
    with ContextLifetime(store) as lifetime:
        key = store.put('value', lifetime=lifetime)
        proxy = store.proxy('value', lifetime=lifetime)

    assert not store.exists(key)
```

```python
from proxystore.store import Store
from proxystore.store.lifetimes import LeaseLifetime

with Store(...) as store:
    lease = LeaseLifetime(store, expiry=10)

    proxy = store.proxy('value', lifetime=lease)
    lease.extend(5)
    time.sleep(20)
    assert lease.done()

    # Object associated with the proxy has been removed
```

Figure 4.10: Example usage of lifetimes when creating a proxy. A `Lifetime` instance represents a physical or logical scope that will clean up all resources (i.e., objects) that were associated with the lifetime when closed. (Top) A `ContextLifetime` defines a block of code within which a proxy is valid. (Bottom) A `LeaseLifetime` defines a time-based lease during which a proxy is valid.

will clean up associated objects once the lease has expired and not been extended, and the static lifetime persists objects for the remainder of the program. Figure 4.10 provides a context-manager and time-leased lifetime example.

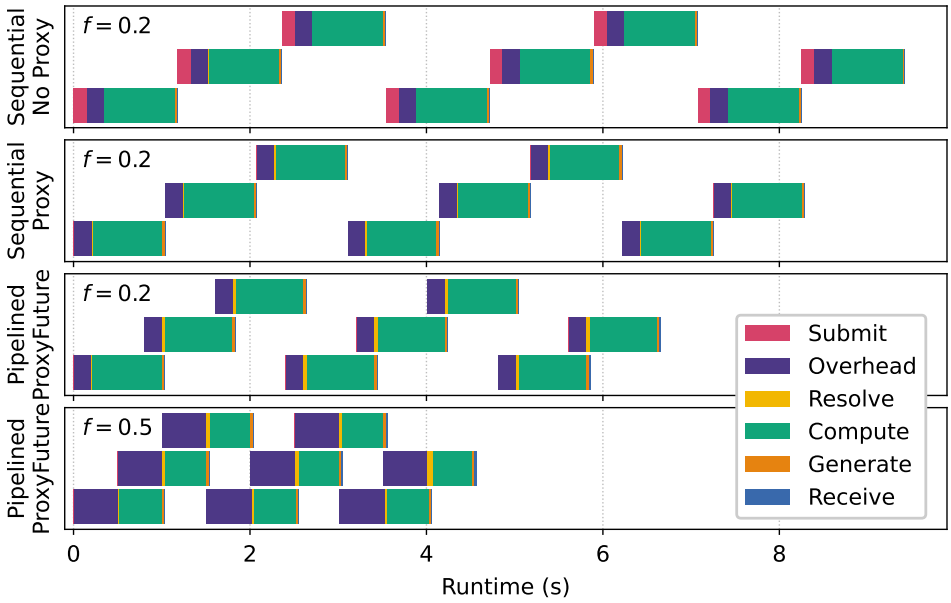
4.4 Synthetic Evaluations

We conducted experiments on Polaris at the Argonne Leadership Computing Facility. Polaris has 560 nodes interconnected by an HPE Slingshot 11 network and a 100 PB Lustre file system. Each node contains one AMD EPYC Milan processor with 32 physical cores, 512 GB of DDR4 memory, and four 40 GB NVIDIA A100 GPUs.

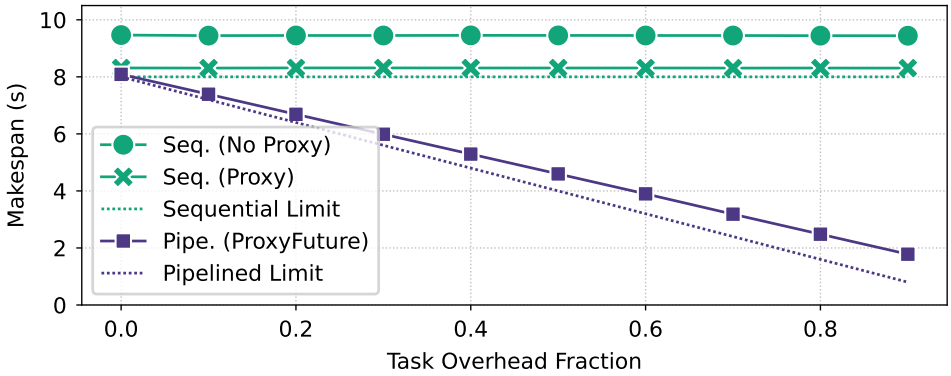
4.4.1 Task Pipelining with ProxyFutures

We first evaluate the effectiveness of ProxyFutures for reducing workflow makespan via pipelining. We define a synthetic benchmark that submits n tasks in sequence, each sleeping for s seconds and then producing d bytes to be consumed by the next task. As in Figure 4.3, a fraction f of each task is treated as startup overhead (e.g., library loading, model initialization, state synchronization). Thus, each task sleeps for $f \times s$ seconds, resolves its input data, and then sleeps for the remaining $(1 - f) \times s$ seconds to simulate computation. We compare three deployments: sequential without proxies (*No Proxy*), sequential with proxies (*Proxy*), and pipelined with ProxyFutures (*ProxyFuture*). In the first two, task t_i is submitted once the result of task t_{i-1} is available, with in *No Proxy*, the workflow engine handling data transfer, and in *Proxy*, data transfer being offloaded from the workflow engine. In *ProxyFuture*, tasks t_{i-1} and t_i share a proxy and future pair and t_i is submitted before t_{i-1} is complete.

Setup: We run a Dask cluster on a single Polaris compute node. In the *Proxy* and *ProxyFuture* deployments, a Redis server running on the compute node is used as the mediated communication channel for the proxies. We run $n = 8$ tasks with intermediate data of $d =$



(a) Task schedules for *no-proxy*; *proxy*; *ProxyFutures* $f = 0.2$, $f = 0.5$.



(b) Makespan vs. overhead fraction for *no-proxy*, *proxy*, *ProxyFutures*.

Figure 4.11: Results for synthetic benchmark with 8 tasks, each sleeping for 1 s and communicating 10 MB to its successor, and with overhead fraction f determining how much of the 1 s can be overlapped with its predecessor task. (Top) Task execution schedules in four scenarios: *sequential no proxy*, with delays due to workflow engine submission costs; *sequential proxy*, with proxies enabling immediate task start after proxy is resolved; and two *pipelined ProxyFuture* cases ($f = 0.2$ and $f = 0.5$), in which distributed futures relax strict inter-task dependencies and enable pipelining to overlap initial task overheads. The *overhead* and *compute* sleeps dominate in all cases, while times to *resolve* task input data and *receive* task results increase, with overhead fraction, while makespan decreases due to pipelining overlap. (Bottom) Synthetic benchmark makespan vs. overhead fraction, for *no proxy*, *proxy*, and *ProxyFuture* scenarios. Each value is averaged over five runs; standard deviations are all less than 20 ms.

10 MB and task time of $s = 1$ s; the short task time is to focus on the time spent producing and waiting on data. We vary overhead fraction f from 0 to 0.9.

Results: We plot in Figure 4.11a the start and end times of each stage in each task's lifecycle for each deployment, for $f = 0.2$, and for *ProxyFuture*, also for $f = 0.5$. Each task incurs fixed *overhead* and *compute* costs, of f and $(1 - f)$ s, respectively. Other costs include: *submit*, the time to submit and begin execution; *generate*, the time to produce output data; and *receive*, the time to receive the result by the client. *Proxy* and *ProxyFuture* also incur *resolve* costs associated with the use of proxies. Figure 4.11b shows the implications of these differences by presenting average makespan as a function of task overhead fraction for the three deployments. The use of proxies in *Proxy* improves task submission time relative to *No Proxy*, reducing makespan by 12%. The pipeline overlapping in *ProxyFuture* enables close to the theoretical limit (dashed line) as determined by inter-task data dependencies. For example, the ideal makespan reduction of a pipeline execution is 20% when $f = 0.2$; we observe 19.6% in *ProxyFuture*. The increased divergence from the ideal reduction at larger overhead fractions occurs because task submission and data transfer costs become more significant as overlapping increases. Thus, a subsequent task begins waiting on its future slightly before the prior task has set the result of the future.

Outcomes: DAG-based workflow execution models limit optimization of task execution because a child task cannot start until its parents have finished, even if the programmer knows it may be beneficial to start it sooner. For example, module loading can account for a significant portion of overall task runtime. Loading TensorFlow on NERSC's Perlmutter takes 5 s in the best case but nearly a minute when many workers read files concurrently [144]. This is particularly noticeable with smaller models where inference time can be measured in fractions of a second. On Polaris, the machine used here, we found that five common libraries (NumPy, Scikit-learn, SciPy, PyTorch, TensorFlow) [31] require from 100 ms to 2 s to import even under ideal conditions with a single worker. Tasks must also often perform other work,

such as file loading, initializing model weights, or state synchronization, before needing their input data. The ProxyFutures model provides for seamless encoding of data dependencies and optimistic task pipelining when tasks have nontrivial initial overheads. While we used Dask and Redis in this experiment, our approach will work with *any* task-based execution engine and mediated communication channel. This engine-agnostic approach will enable programmers to coordinate tasks across multiple execution engines concurrently.

4.4.2 Scalable Stream Processing

Here we evaluate scalable stream processing with ProxyStream. As in Figure 4.5, there is one data producer publishing data of size d to the stream with a rate r (items per second). A dispatch node consumes data from the stream and dispatches a compute task for each data item on to a cluster of n workers. Each compute task is simulated by a task which sleeps for s seconds. The dispatcher executes on a login node, and given n workers, one worker is allocated as the producer while the remaining $n-1$ workers are used to execute compute tasks.

Setup: We compare three streaming configurations. In *Redis Pub/Sub*, data are published directly to a Redis pub/sub topic that is consumed by the dispatcher before being sent to a worker to be computed on. In *ADIOS2*, data are written step-by-step to an ADIOS2 stream [120]. The dispatcher iterates on steps and launches worker tasks which will read the data from the ADIOS2 stream at a specified step. In *ProxyStream*, data are published to a `StreamProducer` which decouples metadata from bulk data, sending metadata to a Redis Pub/Sub topic and storing bulk data in a Redis Key/Value store. The dispatcher consumes proxies of stream data via the `StreamConsumer` and sends proxies to workers to be computed on. ADIOS2 and ProxyStream avoid data transfers through the dispatcher.

We use Parsl's `HighThroughputExecutor`, which can scale to thousands of tasks per second, to manage task execution. We set the producer's data publishing rate $r = (n-1)/s$

Figure 4.12: Compute tasks completed per second as a function of stream item data size and number of workers. One worker generates data consumed by a central dispatcher that launches simulated compute tasks (one second sleep tasks) for each item across the remaining $n-1$ workers. At small data sizes ($\leq$ 100 KB), data transfer overheads are negligible and the dispatcher can keep up with incoming stream data; however, at large data sizes and worker counts, the dispatcher becomes overwhelmed by the size of data transfers required for each task in the Redis Pub/Sub configuration. ProxyStream transparently decouples data flow from control flow improving overall system performance as stream data sizes and the number of workers is increased. Note in many cases the ProxyStream and ADIOS2 markers are overlapped.

items per second, where $s = 1$ s for all tasks. Assuming no overheads in the system, this rate would keep each of the $n-1$ compute workers constantly fed with new data. A range of data sizes d and workers n are evaluated to understand stream scaling throughput limitations. We assign one worker per core so there are at most 32 workers per node. We run each configuration for between five and thirty minutes, depending on the scale, which is long enough for the processing throughput (i.e., tasks completed per second) to stabilize.

Results: Figure 4.12 shows the average compute tasks completed per second. At the smallest data size, $d = 100$ kB, performance is comparable between the three methods because data are not large enough to stress the system. For larger worker counts n and data sizes d, the default Redis Pub/Sub deployment slows because the dispatcher becomes a bottleneck, processing stream data at ~100 MB/s. This rate is slower than the network connection between the Redis server and dispatcher because the dispatcher must, for each stream item, receive and deserialize the item from Redis; compose the task payload, serial-

izing the item again; and communicate the task payload to a worker. Thus the dispatcher cannot process the incoming stream data fast enough to keep workers fed with new tasks when the number of workers or data size is sufficiently high.

ADIOS2 performs better than Redis Pub/Sub because we configured workers to read items from the stream directly based on a step index provided by the dispatcher, improving the latency between the dispatcher receiving stream data and launching a new task. However, ADIOS2 requires changes to the worker task code not needed by the other two methods.

ProxyStream also alleviates data transfer and serialization burdens from the dispatcher enabling performance on par with or better than ADIOS2 but does so transparently without needing changes to the worker task code. The peak processing throughput of ProxyStream is 1.7× and 2.0× faster than ADIOS2 for 1 MB and 10 MB item sizes, respectively. Compared to the Redis Pub/Sub baseline, ProxyStream is 4.6× and 6.2× faster for 1 MB and 10 MB item sizes, respectively. At $d = 100$ MB, the largest data size evaluated, and $n = 256$, ProxyStream is 7.3× faster than Redis Pub/Sub. ProxyStream and ADIOS2 perform similarly at this scale because other aspects of the experimental configuration become bottlenecks. Namely, task execution overheads and storing the data produced by the generator limit peak throughput. A faster data storage system or multiple data generators would be needed to achieve scaling beyond this point, and ProxyStream does support modular data storage and multi-producer configurations.

Outcomes: Streaming proxies, rather than data directly, ensures that objects in the stream are only resolved once needed, thus avoiding overheads due to objects passing via intermediate processes. The `StreamProducer` and `StreamConsumer` interfaces provide a mechanism for composing arbitrary message brokers and mediated communication methods, permitting developers to optimize application deployments without altering task code. The resulting distributed applications are more portable and generalizable to new hardware systems.

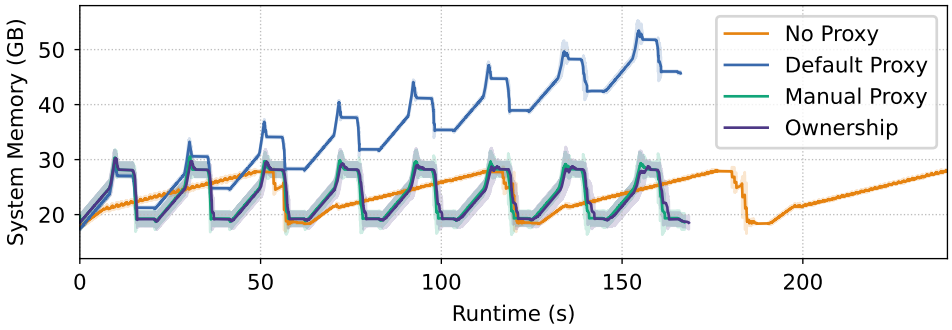


Figure 4.13: Average system memory usage over three runs of a simulated MapReduce workflow. Shaded regions denote standard deviation in memory usage. Memory management limitations in ProxyStore cause baseline memory utilization to increase over time. Manual management can alleviate this problem, but requires careful implementation and prior knowledge. In contrast, our ownership model provides automated memory management equal to a hand-tuned implementation and enforces a set of rules at runtime.

4.4.3 Memory Management

We evaluate the automatic memory management of the proxy ownership model by comparing system memory usage over a simulated workflow to ProxyStore's default memory management and a manual memory management approach which relies on the *a priori* knowledge of the programmer to free shared objects. We also compare to a baseline without any proxies where data are sent directly along with task requests.

Setup: We execute a simulated workflow that imitates a series of map-reduces across a local Dask cluster on a single compute node of Polaris. We run the workflow using each of the proxy memory management models, default, manual, and ownership, and a baseline without proxies using Dask for all data management. We record average memory usage across three workflow executions for each configuration. Eight consecutive map-reduces are performed where each of 32 mappers receives 100 MB and produces 10 MB. We choose 100 MB because the value is large enough to be observable in the memory trace (i.e., larger than the baseline memory usage fluctuations) but is also below the Redis default maximum value size of 512 MB. A single reducer consumes data produced by all mappers. In addition to consuming and producing data, each tasks sleeps for 5 s.

Results: Figure 4.13 presents the system memory usage traces for each memory management model. The limited default memory management of ProxyStore results in memory usage slowly increasing throughout execution as shared objects are created but never freed. The automated management of our ownership model performs identically to manual management and appropriately evicts objects as references go out of scope.

The "no proxy" baseline passes data directly to Dask and utilizes Dask's built-in distributed memory management. We observe that Dask appropriately frees all task data; however, the overall runtime is three times slower. The severe slow down is because Dask's graph serialization performs poorly with large (>1 MB in our experience) arbitrary Python objects, as investigated in Section 3.6.2. Dask is optimized for transferring arrays and dataframes, and we found Dask's performance to be similar to the proxy cases when data were formatted as NumPy arrays.

Outcomes: Our ownership model presents a marked improvement in using proxies in distributed workflows. Enforcing ownership rules at runtime makes it easy to reason about what operations on shared objects are safe and prevents programming mistakes which may lead to memory leaks. Our reference implementation is designed to be agnostic to the underlying task execution engine, but we believe that incorporating this model directly into execution engines can enable more powerful features.

4.5 Application Evaluations

1000 Genomes: We use the 1000 Genomes workflow to investigate ProxyFutures as a mechanism for reducing task overheads and extending data flow dependencies to FaaS systems. Tasks in the original 1000 Genomes workflow were implemented as Bash scripts. We use the Python implementation of 1000 Genomes, where tasks are implemented as functions, to execute the workflow using a FaaS execution engine such as Globus Compute (which we use in the experiments reported here, due to its integration with HPC systems).

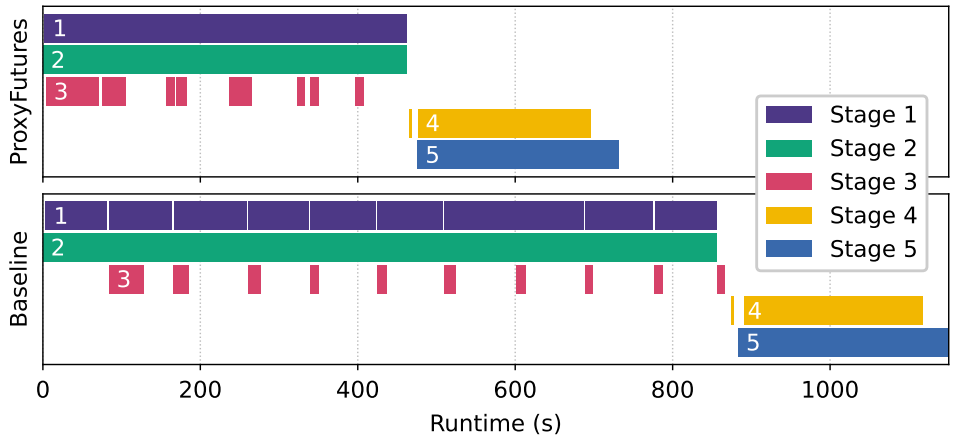


Figure 4.14: 1000 Genomes Workflow stage start and ends times. ProxyFutures reduces workflow makespan by starting computations when data are available rather than when prior tasks complete.

We evaluate the makespan of the resulting workflow, using 5% of the 1000 Genomes dataset, on a single `compute-zen-3` node, with two 64-core CPUs and 256 GB memory, on Chameleon Cloud's CHI@TACC cluster [146]. Figure 4.14 shows workflow stage start and end times for a baseline implementation, which uses Globus Compute's native futures for data synchronization between tasks, and a ProxyFutures implementation. As each stage can contain up to thousands of tasks, we consolidate the tasks within stages for clarity. ProxyFutures reduce workflow makespan by 36%, by better overlapping task execution and communication costs across stages. More specifically: (1) tasks within stages 1, 2, and 3 are better overlapped, reducing the stage makespans by 47–48%; (2) response time, the time between receiving a task result and submitting another task, is improved (for example, by 54% when starting stage 4); and (3) stages 4 and 5 are 5% faster due to reduced data transfer overheads. We also note there are no dependencies between tasks within stages 4 or 5, so these stages do not benefit to the same degree as the earlier stages.

DeepDriveMD: We modify the Parsl implementation of DeepDriveMD [47] to stream inference batches and results to and from a single, persistent inference task with ProxyStream. A persistent inference task eliminates task overheads and enables reuse of models

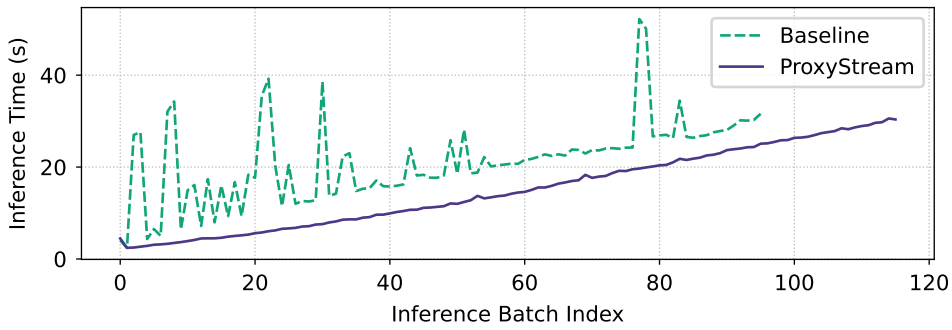


Figure 4.15: Comparison of inference round-trip time between two DeepDriveMD implementations: baseline and ProxyStream. The size of each batch increases over time as the application accumulates more data points.

and caches. Streaming with proxies reduces overheads in the DeepDriveMD client because received inference results are immediately added to a queue of simulation task inputs. In addition to ProxyStream, ProxyFutures are used to indicate availability of a new ML model to the inference task and proxy references for management of intermediate task data.

We compare the performance of DeepDriveMD to a version that uses proxy patterns. We run each version for three hours using 40 GPUs on Polaris, dedicating one GPU for inference, one for training, and the remainder for simulations. Round-trip inference time, shown in Figure 4.15, is reduced from an average of 21.9±8.8 s to 15.0±8.4 s, a 32% improvement, and 21% more inference batches were processed in the same wall time. Reducing inference time is key to enabling greater simulation throughput, such as when the number of simulation workers is increased or simulation time is reduced.

MOF Generation: We modify the MOF Generation application to communicate all task input and output data larger than 1 kB via proxies. (The overhead of proxying simple data types such as boolean flags or configuration strings is greater than sending those objects directly.) We deploy the application with default settings on ten Polaris nodes. We run the application twice: with the standard proxy implementation of ProxyStore and with our proxy ownership model. Here, ownership was sufficient; we did not use the lifetimes model. We record the number of actively proxied objects during the application's runtime. As shown

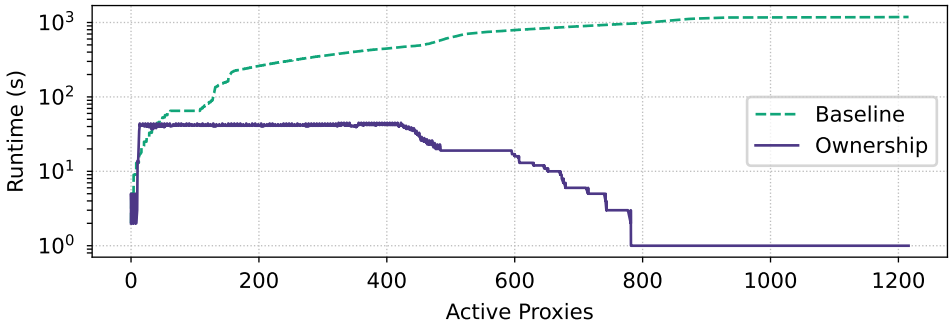


Figure 4.16: Number of active proxies (i.e., proxies that still have a stored target object) during the runtime of the MOF Generation application. Our ownership model for proxies appropriately cleans up proxies when no longer needed while maintaining the benefits of the pass-by-reference model.

in Figure 4.16, the ownership model appropriately evicts proxied data when the lifetime of the associated proxy ends without altering the runtime behaviour of the application. Manual memory management is possible, as discussed in Section 4.4.3, but automated management is safer and makes adoption of advanced programming practices, such as those we present here, easier and more appealing.

4.6 Related Work

Futures are a pervasive programming abstraction for asynchronous and concurrent programming [30, 111]. Implicit futures act as references; any dereference blocks automatically until the value is resolved [100]; thus, they typically require language-level support [67, 66, 108, 109, 58]. Explicit futures provide a public interface, such as a `get` method, that must be invoked to block and retrieve the value; consequently, they can be provided by languages and third-party libraries.

Explicit futures require control flow synchronization code, which reduces code flexibility and complicates functions that want to operate on a future or a value directly. Either two implementations or multiple execution paths must be present to support each case. Implicit futures are also inflexible because they require that the language's type system

handle the mechanics of lifting the value out of the future transparently. Thus few languages support implicit futures, and programmers have limited ability to modify the resolution and lifting processes. ProxyFutures address these key limitations by providing both an explicit mechanism, the `Future`, and an implicit mechanism, the `Proxy`, for Python applications.

Distributed futures represent values that, when available, may be located in remote process memory. Distributed futures are often underpinned by a remote procedure call (RPC) system, such as in Dask [219], PyTorch [210], and Ray [184, 246]. Because these futures are implemented by the RPC framework, rather than the language, all are necessarily explicit futures, and their use is limited to the confines of the framework. Thus, for example, one cannot create a distributed future in Dask or Ray and then invoke a serverless function with Globus Compute [61] on that future. In contrast, ProxyFutures works across frameworks and supports many mediated communication methods via a robust and extensible plugin system.

Streaming applications in which producers and consumers generate and process data continuously are commonly executed at scale on high-performance and cloud computing systems. Their persistence and resilience needs may be met by message queuing systems such as Apache Kafka [23], Redis [218], and RabbitMQ [211]. However, these systems typically optimize for high-throughput, low-latency transmission of small, structured events, in order that these events can be aggregated, filtered, or transformed, as in Kafka.

In contrast, high-performance science applications often produce large raw or unstructured data accompanied by structured metadata [39]. File-oriented distributed applications often use GridFTP [59, 15]. Dispel4py [103, 161] maps abstract definitions of streaming workflows onto concrete distributed execution frameworks, such as Python multiprocessing or MPI [175]. Streamflow [132] extends the DAG-based workflow model to integrate continuous event processing. ADIOS SST [97], a streaming engine for HPC workflows, and ADIOS WASP [63], a data staging platform for scientific stream processing, use self-describing file

formats and support advanced networking technologies such as RDMA. The SciStream middleware [64] enables fast, secure memory-to-memory streaming between nodes that lack direct network connectivity. CAPIO [172] provides a middleware layer for injecting I/O streaming capabilities into file-based workflows.

Consuming an entire stream item (data and metadata) is expensive when only metadata are needed for decision making or data is to be forwarded to another application component. ProxyStream decouples event metadata notification from bulk data transfer. Streaming proxies allows data transfers to occur when and where needed, with specifics of the message broker and data storage abstracted from the program.

Garbage collection in distributed environments is challenging. Automatic techniques such as reference counting and tracing garbage collectors exist, but often require *a priori* knowledge by the application programmer to add custom logic for shared object management, and can be inefficient in distributed environments [160, 38, 203, 183]. Maintaining global reference counts or traces adds network overheads, single sources of failure (if reference counting is centralized), or atomicity/consistency challenges (if reference counting is distributed).

Leases, a decentralized, time-based mechanism, can be used to avoid maintaining a shared state across processes [126]. Task-based execution engines can avoid shared state problems and the complexities of reference count message passing because the central client or scheduler can act as a single source of truth [219, 184]. The notion of ownership uses a program's inherent structure to decentralize state management. In PyTorch RPC, each object has a single owner that maintains the global reference count as remote processes need to access the data [210]. Related work extends this concept to implement distributed futures and task recovery in Ray [246].

Our proxy-based approach avoids the complexities of global reference counting by associating object lifetimes with tasks, and our framework-agnostic approach means that object

scopes can be appropriately managed across complex, distributed applications.

4.7 Summary

The lazy object proxy is a powerful construct for building distributed applications, providing benefits of both pass-by-reference and pass-by-value while abstracting low-level communication details from consumers. Here, we have applied this construct to realize three powerful parallel programming patterns: a compute framework agnostic distributed futures system, a composable streaming interface for data-intensive workloads, and an ownership model for object management in distributed, task-based applications. We evaluated these patterns through synthetic benchmarks and showcased three classes of scientific applications that can benefit from the proxy paradigm powered patterns. Specifically, we reduced the 1000 Genomes workflow makespan by 36%, reduced DeepDriveMD inference latency by 32%, and optimized memory usage during MOF generation.

These patterns enable the development of robust, scalable, and portable applications. For example, ProxyFutures empowers data flow dependencies between tasks executed across different execution engines, such as when one engine is used for local execution on a cluster and another for remote execution on cloud resources. ProxyStream can support long-running scientific campaigns by using cloud-hosted message brokers for reliable metadata streaming and Globus Transfer for federated, persistent bulk storage and efficient transfer. The proxy ownership model provides automated wide-area memory management for distributed and cross-site workflows. In the future, we will investigate further programming patterns that can be enhanced with the proxy paradigm. Our work here serves as a reference for integrating these design patterns into execution frameworks, such as Dask, Globus Compute, or Parsl, and other high-performance computing toolkits. By providing first-class support for these patterns directly within commonly used frameworks, we expect to enable speedups in many scientific applications.

CHAPTER 5

ENABLING AGENTIC WORKFLOWS ACROSS FEDERATED RESOURCES

The desire to automate scientific processes has led to advancements in many fields, from artificial intelligence (AI) [267] and computational workflows [85] to research data management [15] and self-driving laboratories (SDL) [5], but humans typically remain responsible for core aspects of the iterative research cycle, including hypothesis generation, experimental design, code development, and data analysis. Often, the *human-in-the-loop* is the rate-limiting step in discovery. This friction increases as the scale and ambition of computational science endeavors grow and leads to inefficient use of research cyberinfrastructure—the federated ecosystem of experimental and observational facilities, data repositories, and high-performance computing (HPC) systems [177].

Intelligent agents, either as an individual system or composing larger multi-agent systems (MAS), rather than humans, can be the driving entities of discovery. Agents are independent, persistent, stateful, and cooperative—working together to achieve a predefined goal with only intermittent human oversight. The contemporaneous explosion of interest in multi-agent systems is largely a consequence of advancements in reasoning capabilities of the large language models (LLMs) often used to back AI agents. Expressing components of scientific applications as agents—programs that can perform tasks independently or semi-autonomously on behalf of a client or another agent—is powerful. An agent manages its own local state and exposes a well-defined behavior. Agents can perform human roles in iterative scientific processes [245] or encapsulate research cyberinfrastructure (e.g., computational resources and procedures, experimental instruments, and data repositories) [115].

Significant progress has been made towards developing AI agents that can act on behalf of humans for such tasks as literature synthesis [153], hypothesis generation [124], and data analysis and publication [140]. However, existing agent frameworks (e.g., AutoGen [259])

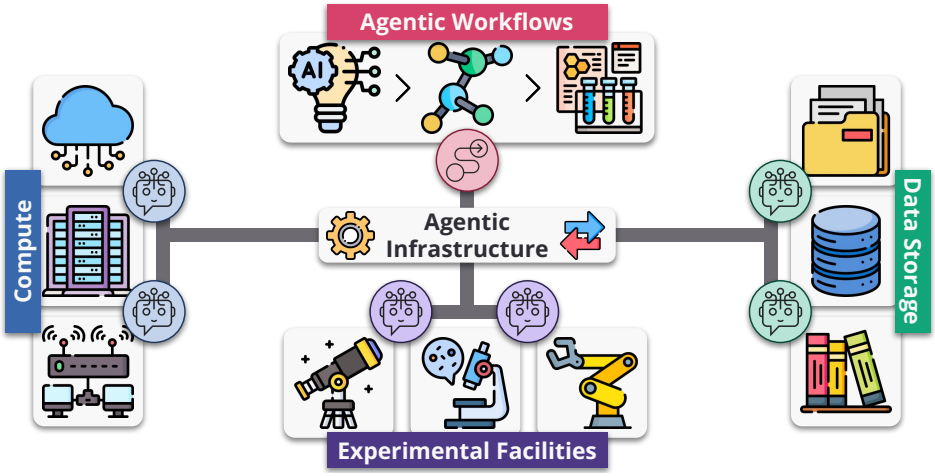


Figure 5.1: Cooperative agents, spanning federated research infrastructure (experimental facilities, computational systems, data storage), can enable agentic workflows that autonomously steer discovery.

are not ready to build and deploy agents that employ federated research cyberinfrastructure. New middleware is needed to enable *agentic workflows* that seamlessly integrate experiment, observation, theory, simulation, AI, analysis, and more, as in Figure 5.1.

Frameworks for building agentic workflows are limited in scope and generally target conversational, cloud-native applications (e.g., LLM-based AI chatbots) [155, 259, 189]. The federated nature of research infrastructure poses unique challenges: distributed resources have diverse access protocols, interactions between computational and experimental entities are asynchronous, and the dynamic availability of resources requires fault-tolerant and adaptive systems. Existing frameworks fail to address these intricacies. They lack abstractions and mechanisms tailored to support autonomous multi-agent workflows that integrate computation, data management, and experimental control, which leads to brittle, *ad hoc* integrations that are ill-suited for the demands of modern science. Moreover, the inherent complexity of such workflows is compounded by the need to balance efficiency with scientific rigor, especially in applications involving real-time decision-making, iterative exploration, and multi-agent coordination.

These challenges are often orthogonal and span many levels of abstraction, but achieving

this vision where intelligent agents serve as driving entities in scientific discovery requires a paradigm shift in how workflows are designed, orchestrated, and executed. We introduce a novel framework for building agentic workflows, emphasizing modularity, statefulness, and interoperability across the diverse research infrastructure. Specifically, this work contributes:

- A vision for an era of autonomous discovery steered by federated collections of agents deployed across research infrastructure (Section 5.1).
- A formalism for agentic workflows as a programming paradigm for autonomous discovery (Section 5.2).
- ACADEMY, a novel, modular, and extensible middleware for expressing agentic workflows and deploying multi-agent systems across federated resources. ACADEMY addresses unique challenges in scientific applications, such as high data volumes, variable resource availability, and the heterogeneous nature of experimental and computational systems (Section 5.3).
- Performance analysis of ACADEMY in diverse scenarios yielding insights into the scalability and practical considerations of deploying agentic workflows (Section 5.4).
- Case studies demonstrating the utility of agentic workflow design and highlighting improvements in automation, resource utilization, and discovery acceleration (Section 5.5).

These contributions advance the state of the art in multi-agent systems for scientific discovery and establish a foundation for future innovations in autonomous research workflows.

5.1 Autonomous Agentic Discovery

Scientific discovery has entered an era of unprecedented complexity. Modern research demands the seamless integration of experiments, observations, models, simulations, artificial

intelligence (AI), and machine learning (ML)—all while processing ever-growing volumes of data and leveraging large, diverse, and distributed computing infrastructure. The landscape of discovery has been reshaped across disciplines; researchers rely on long-running experimental observation, expansive data infrastructures, cutting-edge AI models, and high-performance computing to uncover insights. This observation is emblematic of broader transformations captured by the fourth and fifth paradigms of science, which describe the shift towards data-intensive methods and artificial intelligence, respectively, as integral aspects of scientific exploration [134, 170]. (The prior paradigms are classified as empirical, theoretical, and computational, respectively.) Fields ranging from astrophysics to social sciences now rely on vast datasets, AI models, and computational methods to drive innovation. The challenge lies not just in managing data or building models, but in building systems that enable researchers to integrate and utilize them at scale. Current approaches to integrating data-intensive workflows and AI methods have yielded considerable success, but use techniques that result in siloed solutions that fail to scale or generalize. This paradigm shift demands more than building increasingly sophisticated tools; it calls for a fundamental rethinking of how science is conducted.

Many aspects of the iterative research process can be automated (data acquisition and preparation, workflow orchestration, modeling and simulation, and data visualization); however, human experts are still required to propose hypotheses, design experiments, write programs, procure resources, and interpret results. These human-driven aspects require experts to keep up with state-of-the-art concepts, techniques, and results—a task that is increasingly impractical. The ability of scientists to remain up-to-date is precluded by the exponential growth in published data and papers [128]. Sustaining the ever-growing scale of scientific endeavors will require addressing these human limitations.

Agents—programs capable of independently or semi-autonomously performing tasks on behalf of a client or another agent—have been studied for decades, but contemporaneous

advancements in AI have reinvigorated interest. Namely, the reasoning capabilities of large language models (LLMs) have improved to the point where intelligent agents can steer more complex processes with the flexibility and autonomy that previously only humans could provide. A future in which agents play more anthropomorphic roles in iterative research cycles is tangible. Like human experts, agents are typically specialized and share the autonomous, persistent, stateful, and collaborative attributes necessary to work in unison to achieve broader goals. Yet, specialized agents can perform their roles more efficiently than humans—at least to an extent—enabling more autonomous, higher throughput, and arguably more reliable discovery.

As a programming paradigm, agents are a valuable conceptual model and practical framework for integrating loosely coupled systems that compose large-scale research infrastructure, and agents can be the missing piece that enable long-running, autonomous use of these resources. An agent-driven future of science requires continued advancements at many levels of abstraction and attention to both human and technical challenges, but we posit that the payoff will be immeasurable.

In this section, we review the history of agents from conceptualization with actors through to modern adoption in AI-based "agentic workflows." We describe our prediction for an era of agentic discovery in which federations of agents work cooperatively to augment or replace humans within scientific processes. We present a case study in materials discovery to orient our prediction and describe how agents will transform every phase of scientific discovery to create an autonomous discovery process. We distill key technical challenges that must be addressed to achieve such visions and outline several uncertainties and risks.

5.1.1 What is an Agent?

Agent-based frameworks have proliferated recently; yet, the reemergence of agents in LLM contexts obscures the breadth of research encompassed by the term. Contextualizing the

history, taxonomy, successes, and failures of agents is key to evaluating the impact of agentic applications for scientific discovery.

The origins of agentic systems are rooted in Carl Hewitt's actor model [133]. Actors are independent computational entities that enable concurrent computing (where the lifetimes of many distinct computations overlap) through asynchronous message passing. In response to a message, an actor can alter its local state, send messages to other actors, and create new actors. This conceptual model is simple—lack of global state obviates the need for locks and other synchronization primitives—and powerful—Hewitt describes that "all of the modes of behavior can be defined in terms of one kind of behavior: sending messages to actors." Laying a formal foundation for distributed systems, the actor model influenced many early concurrent programming languages (e.g., inter-process communication via mailboxes in Erlang and tuple-spaces in Linda) and methods for fault-tolerance, state management, and consistency (e.g., Paxos [154]).

A formal definition of "agent" has been long sought. Abstractly, an agent is an entity—something or someone—that acts on behalf of another entity; this is rather actor-like within the context of computer science. However, popular use of the term "agent" emerged throughout the 1980s within distributed artificial intelligence. Then, and continuing into the 1990s, AI was often regarded as "the subfield of computer science which aims to construct agents that exhibit aspects of intelligent behaviour" [257]. For example, reinforcement learning methods such as Q-learning [252] considered how to develop agents that maximize a reward through interacting with the environment. Like actors, agents were entities that could communicate and manage their own internal state, but exhibited intelligent behavior weakly defined as operating autonomously, perceiving and reacting to the environment, and taking goal-oriented actions.

While modern AI methods are dramatically different, multi-agent systems (MAS) were seen as critical to distributed artificial intelligence. Researchers posited that a MAS would

exhibit emergent behavior [244], enabling more complex problem solving through internal decision making where agents collaborate by goal setting, planning, negotiating, and reasoning, as in BDI (belief, desire, and intention) [215].

Despite the theoretical promise of MAS, practical challenges arose. Scalability, robustness, and the complexity of designing intelligent, cooperative agents hindered widespread adoption in the early 2000s. While some successes were notable, such as agent-based modeling, MAS struggled to deliver on the ambitious vision of emergence at scale. The actor model remained relevant within distributed computing, but the field of AI was no longer defined by the practice of constructing agents following the advancements in hardware and algorithms that lead to the rise of deep learning.

Agents found renewed relevance in the 2020s with the advent of large language models (LLMs). These systems, pre-trained on vast corpora of text, exhibit emergent capabilities in natural language understanding and reasoning. By embedding LLMs within agent architectures, researchers revisited earlier visions of autonomous, intelligent agents with newfound capabilities. Frameworks like AutoGen [259] and OpenAI Swarm [189] leverage LLMs to create agents that can perform tasks such as information retrieval, summarization, and interactive collaboration with humans and other agents. Tool calling, such as in Claude [21] or LangChain [156], enables LLMs to invoke user-provided external functions or APIs. Google's AI co-scientist [124], built on Gemini 2.0, is a multi-agent AI system designed to collaborate with scientists by generating novel hypotheses and research proposals. In these MASs, each agent plays a special role defined by its system prompt; agents then communicate, iteratively refining the response or mapping messages to actions, to satisfy their goal—the client's query.

Within computer science, an agent remains—and will likely remain—ill-defined. Rather than attempt to naively reconcile this history through an $(n + 1)$th definition of an agent (c.f. xkcd 927 [185] for a humorous take), we contextualize the remaining discussion by summarizing the kinds of high-level behaviors that an agent can exhibit. An agent is broadly

classified as *deliberative* (often called *intelligent*) or *reactive*. A deliberative agent contains a model of the environment (a state) that is used to reason about what long-term plans to make in order to achieve its goal [257]. In contrast, reactive agents lack a world model and only take actions in response to changes in their perceived environment [188].

We can further define more specialized behaviors. *Service* agents provide predefined services, including executing computational routines or provided resource access. *Embodied* agents interact with the physical world, often via an actuator. *Learning* agents refine their behavior over time, typically leveraging reinforcement learning to enhance performance. *AI* agents employ an AI model, typically, but not limited to, an LLM, for taking actions or making decisions. A MAS is composed of *cooperative* agents that work together toward high-level system goals by planning and coordinating smaller tasks across agents with the appropriate behaviors and resources. To a client, a MAS can manifest itself as a single agent, encapsulating the complexity of the system. We emphasize that nothing precludes an agent from exhibiting multiple behaviors—in fact, most do!

5.1.2 A Vision of Agentic Discovery

We predict that federations of cooperative agents—deliberative, reactive, embodied, and more—will augment, and often replace, the human-in-the-loop in scientific endeavors. This prediction originates from two observations. First, human decision making tasks limit the rate of discovery, and second, advancements in agentic systems are converging to a point where the complete scientific method can be carried out autonomously. We explore the first observation through a case study in the autonomous discovery of carbon capture materials. Then we describe a future in which specialized agents champion each phase of the scientific method.

A Case Study in Materials Discovery: The urgency of climate change demands innovative solutions across multiple fronts. Carbon capture is a crucial component of this

multi-targeted approach, as reducing atmospheric CO_2 levels can mitigate the greenhouse effect and slow global warming. Traditional methods of carbon capture, such as chemical scrubbing and geological storage, face challenges in efficiency, scalability, and cost [166]. Metal-organic frameworks (MOFs) offer a promising alternative; these polymers, composed of inorganic metal clusters and organic ligands, are porous with high surface area. MOFs can be tuned for selective gas adsorption properties making them highly effective for capturing and storing CO_2. The large surface area and pores of these structures makes them ideal for selectively adsorbing gases; thus, researchers are interested in designing MOFs for use in catalysis, drug delivery, gas storage, and—particularly—carbon capture to reduce atmospheric CO_2 levels and mitigate climate change. Discovering new MOFs with optimal properties is a daunting task due to the intractable combinatorial space of possible structures and the costs associated with synthesis and evaluation. Thus, scientists desire autonomous methods for screening vast quantities of MOF structures. One such example is MOFA [263], an online learning framework for generating, screening, and evaluating MOFs that couples generative AI methods with computation chemistry.

The high-level MOFA workflow is as follows: (1) a specialized AI model generates candidate ligands, (2) candidate MOFs are assembled using predefined metal clusters and generated ligands, (3) candidates are iteratively screened and validated using multiple molecular dynamics computations, (4) CO_2 adsorption of promising candidates is simulated and stored in a database, (5) the generative AI model is periodically retrained on these results to improve performance. MOFA is representative of a broad class of scientific workflows that have contributed to advancements in many fields. While these workflows aim to enable autonomous discovery, their rigid and tightly coupled design means that humans are still primarily responsible for stewarding their utility in broader scientific endeavors.

MOFA can screen MOFs many orders of magnitude faster than any human could synthesize and evaluate a MOF. Yet, MOFA represents only a small step within the search

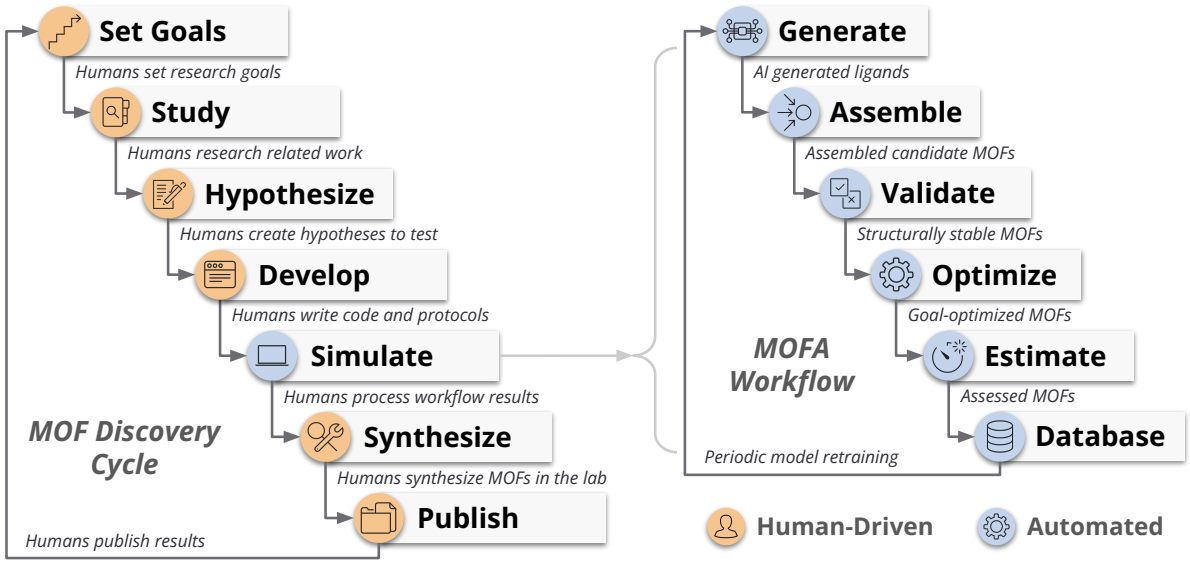


Figure 5.2: The discovery cycle of metal-organic frameworks (MOFs) for carbon capture is largely human-driven (orange stages). While some aspects have been automated (blue stages), such the AI generation and simulation of MOFs in the MOFA workflow, human responsibilities limit the rate of MOF discovery.

for better carbon capture materials so the benefits of this acceleration are poorly realized. Consider the steps that happen before or after a scientist executes MOFA, also depicted in Figure 5.2: domain experts distill techniques and results from the literature to inform hypotheses, programmers design more accurate and faster molecular dynamics simulations, ML practitioners investigate new generative model architectures, and chemists synthesize and evaluate MOFs in laboratories. The outcomes of each step influence each other, but propagation and application of outcomes is human-driven which introduces considerable latency. Accelerating an individual task is good, such as using MOFA for high-throughput screening, but accelerating decisions can have far greater impacts. Multi-agent systems are ideal for optimizing these research processes, retaining the autonomy of different research components while automating decision making.

In traditional workflows, policies dictate what tasks to run, when to run them, and where they should be executed. Expression of these policies must be global to construct and execute the task graph. This leads to two significant drawbacks: (1) modifying or

extending components requires altering global policies, and (2) components cannot improve their policies over time based on local information. Further, local state must be centrally aggregated to make updated decisions, which can be inefficient at scale or when components have high latency. In contrast, agents have autonomy over their policy expression and evolution. An agent can improve its policies over time and be updated by a human without needing to alter other agents. In MOFA, for example, this can manifest as agents using active learning to prioritize queues to better allocate dynamic computational resources towards MOFs with better heuristics.

Scaling out workflows to enable new capabilities is desirable, particularly by leveraging integrated research infrastructure (IRI), which combines experimental facilities, data repositories, and high-performance computing (HPC) systems. However, the traditional tightly coupled workflow model makes this challenging. Dynamic resource availability requires inefficient synchronization, component failures can disrupt dependent tasks, and state synchronization across federated environments is costly. An agent-based architecture naturally accommodates asynchronous execution, making it more resilient to variable workloads and resource fluctuations. By decentralizing decision-making and enabling adaptive execution strategies, multi-agent systems have the potential to revolutionize autonomous scientific discovery, ensuring that efforts in carbon capture and broader scientific domains are both efficient and scalable.

Closing the Loop with Agents: Scientific discovery is inherently iterative, involving goal definition, research, hypothesis generation, experimentation, analysis, and dissemination. Agents can play specialized roles at each phase while also transcending individual steps to coordinate broader research objectives. We discuss each of these roles in detail, including examples within our materials discovery use case, and summarize this ecosystem of cooperative agents in Figure 5.3.

We show in the figure a set of intelligent learning agents that transcend any one phase

Figure 5.3: The scientific method is an iterative process (stages depicted in the central loop). Specialized agents (depicted as boxes with corresponding stages indicated by color) can carry out the stages autonomously. Agents can also transcend stages to enable long-term planning, exploration, and safety.

of the method and instead guide the actions of other agents in the MAS. An **Exploration** agent steers the system, prioritizing breadth in early phases then transitioning towards more targeted exploration when promising avenues are discovered. A **Planning** agent is responsible for managing the environment and inherent tradeoffs in which the agent operates, for example, to guide allocation of shared resources, follow predefined policies, and so forth. An **Enforcement** agent plays the critical role of ensuring that agents' actions are safe, legal, and meet other regulatory requirements. These agents interact with many, if not all, of the agents in each phase. *In MOFA, agents can steer exploration of vast chemical design spaces, plan long-running experiments, allocate resources across specialized agents, and ensure that actions satisfy safety and technical constraints.*

The scientific process is framed around a particular goal. For example, to discover MOFs that are effective at capturing and storing CO_2, to understand the nature of the universe, or to characterize the molecular pathways that lead to a specific disease. As such, the

first phase of the method aims to define, with some degree of specificity, the question(s) that guide the subsequent phases. An **Objective** agent, when given a high level goal by a human, can derive a set of questions or pose conjectures. Enabling agent-to-agent or agent-to-human debate can improve the deliberation process of these agents, producing better or more refined questions. *In MOFA, an LLM-based AI agent can conjecture about possible base metal nodes of interest or ask questions about the performance impacts of specific structure geometries.*

Given a set of questions, the next phase in the process, study, seeks to identify knowledge that can help address the question. The **Knowledge** agent must mine literature, identify relevant prior experimental and simulation results, obtain published data, and establish linkages between potentially related information (for example by leveraging embedding databases and via retrieval augmented generation). *In MOFA, a knowledge agent can utilize retrieval retrieval-augmented generation (RAG) to investigate prior uses of metal nodes of interest, or leverage embedding databases to find similar structures of interest.*

The **Prediction** agent is a form of intelligent agent that synthesizes the questions, conjectures, and knowledge from prior agents into hypotheses that can be tested. This agent learns over time and incorporates a degree of creativity in its decision making. A key aspect of the learning process is improving the feasibility of hypotheses using feedback from experimental agents. *In MOFA, the prediction agent proposes changes to certain input parameters that will result in a desired effect.*

The experiment phase seeks to gather data that can prove or disprove hypotheses. These experiments may be conducted by one or more **Service** agents. They may encompass conducting physical experiments via automated laboratories (embodied agents), simulated experiments using HPC infrastructure (computational or code-generation agents), or observational data derived from sensors (observational agents). Service agents may self-coordinate in a peer-to-peer fashion, or depend on planning agents that break up tasks into smaller actions to be dispatched to the appropriate agent. *In MOFA, AI agents can generate candi-*

date ligands, computational agents can perform screening simulations, and embodied agents can synthesize materials and evaluate them within self-driving laboratories.

Data and observations from experiments are interpreted by **Analysis** agents. These agents look for trends or patterns in the data, may train or use models, and will ultimately interpret the data to derive findings. *In MOFA, statistical analysis and causal inference agents can use results to determine the veracity of hypotheses and review the efficiency of performed experiments (e.g., which assays were most indicative, resource utilization, etc.).*

Finally, the **Publish** agent will store and disseminate the results in the form of knowledge. Depending on the consumer, this knowledge may be communicated in different ways, for example writing to an embedding database or knowledge base for other agents, preparing a video to share with the public, or publishing a paper to share with experts. Importantly, this agent must capture the provenance of the research processes in a way that the results are verifiable, understandable, and ideally, reproducible. *In MOFA, the publish agent can write generated MOFs to a database and disseminates outcomes to the objective, knowledge, and prediction agents.*

Evolving Human Responsibilities: Realizing this world of agent-driven discovery does not negate the need for nor the utility of scientists. Rather, we envision that the responsibilities of scientists will transition away from mundane tasks, such as experiment execution, resource provisioning, and monitoring, to higher-level objectives. Strategic decision making, long-term objective setting, theoretical and conceptual development, interdisciplinary collaboration, verification and validation of findings, and general system design remain central tasks for scientists. The key distinction with these tasks is that they are not responsible for the overheads that impact the rate of discovery.

5.1.3 Key Technical Challenges

Achieving this vision of completely autonomous discovery requires addressing several critical technical challenges. We briefly highlight important challenges below.

Discovery and Interfaces: Composing autonomous discovery processes requires first identifying the agents that are necessary and perhaps evolving the set of participating agents over time. New discovery capabilities are needed to allow agents to discover other agents, determine what those agents can do, and how well they can do it. Agents may also consider other aspects such as cost, safety, reliability, ethics, and so forth. A second concern then is how agents may interact with one another, for example, via secure and self-descriptive interfaces. The significant prior work on actors provides a general basis for such interfaces; however, recent work with LLM and chat interfaces, agent-based frameworks like AutoGen, and remote computing, robot, and data interfaces must also be considered.

Access Control and Sharing: By definition, agents will interact with other agents. In many cases these agents will span resources, facilities, institutions, regions, and even countries. As a result, it is critical that rules and regulations are followed and enforced, for example, limiting what resources may be used, how those resources are shared, and what resources can be used for what purposes. Overarching policies, created by humans or machines, can be used to define these rules, and agents must incorporate mechanisms to enforce these policies. Further, it will be important to audit provenance traces to understand not only what was done, but why it was done and by whom.

Infrastructure: The discovery process may include a broad range of resources—computation, data, models, people—that span locations and administrative domains. Making effective use of these resources represents a considerable hurdle with respect to interfaces, policies, and usage requirements. Further, many scientific problems involve diverse data types (e.g., images, text, numerical data, sensor readings), unique instruments (e.g., microscopes, telescopes, gene sequencers), hardware (clusters, edge sensors, AI accelerators), and model

types that must be leveraged by agents. A final concern is agent resilience as agents may fail in myriad ways. Detecting, abstracting, and recovering from failures is a long standing challenge in distributed computing which is amplified by deployment of autonomous agents.

Agent Mobility: Multi-agent systems for autonomous discovery will rarely be static. Agents must be deployed, terminated, replaced, scaled, and on occasion, moved between locations. Such mobility must be a cornerstone of the infrastructure and communication methods described above. Prior work in mobile agents [201], often referred to as mobile code, provides a foundation for research in this area, considering, for example, code portability, *ad hoc* networking, and communication patterns.

Provenance and Reproducibility: The scientific method is one of proof and validation. It is critical that others be able to not only understand what has been done, but also to validate the results and methods applied, and to reproduce and extend those methods. The fact that learning agents may be prone to opaque decision making necessitates efforts focused on interpretability and explainability of individual decisions. However, considering the nascent state of scientific reproducibility, there is an opportunity for agentic discovery to rapidly improve the verifiability and reproducibility or research processes by integrating provenance as a first-class citizen in the discovery lifecycle. Such capabilities may rely on verifiable ledgers to document processes and decision making, the ability to introspect agents and their behaviors, and methods to reuse agents in different settings to reproduce results.

5.1.4 Uncertainties and Risks

Beyond addressing key technical challenges, the transition towards agent-driven discovery raises several uncertainties and risks. The greatest uncertainty—as with any large-scale endeavor—is garnering buy-in from stakeholders, the scientists, research institutions, and funding agencies that will devote time and resources towards enabling this future. We discuss such risks that, if ignored, will heighten stakeholder uncertainty.

Prior periods of research into agents, as mentioned earlier, have seen mixed success, and it is not unreasonable to be weary of history repeating itself. Research then, just as it is now, is ambitious and the reasons for past failure were complex: AI methods were primitive, software protocols were more fragmented, and hardware limitations prevented innovation. We believe that we have reached an inflection point where research in AI, software, and hardware has advanced to a point where their integration in the form of agents is feasible and will present a value add to science. Not least, lessons learned from past failures will inform future decisions.

Worse than absolute failure or success is the illusion of success. Autonomous systems are susceptible to security and safety risks, including adversarial attacks that could manipulate results or physical safety when autonomous actions are taken in the real world. These risks are not unique to agents but require deference as ambition and scale increase. In a similar vein, bias in any AI system must be characterized and controlled for, otherwise results may unknowingly be invalid. For example, as in our MOF design application, biases in training data could lead to skewed discoveries that favor certain materials or methods over others.

Scientific discovery thrives on collaboration, yet autonomous workflows may inadvertently promote isolation if proprietary models or siloed datasets dominate research. Open-source initiatives, shared repositories, and cross-disciplinary partnerships must be encouraged to prevent fragmentation and maximize the collective impact of agents. Collaborations between human researchers and autonomous agents will pose new challenges to determining proper attribution of outcomes (just as we have seen with LLMs and writing). Traditional academic crediting mechanisms may need to evolve to fairly recognize the tightly coupled contributions of humans and AI.

5.2 Formalization of Agent Systems

Agents encompass a rapidly expanding front for AI research, yet agent paradigms can address a breadth of challenges across the computational sciences. We begin with a definition of an agent—inspired by prior work—that is sufficiently generic to encompass the various semantic uses of the term. Then, we enumerate common high-level classes of agents and formalize agentic workflows, both of which we aim to support in the design of ACADEMY in Section 5.3.

An agent is a program that can perform actions independently or semi-autonomously on behalf of a client or another agent. This definition is imprecise but presents a powerful conceptual model for distributed computing. The agent concept originates from the actor model, a concurrent computing paradigm in which actors encapsulate a local state and communicate through message passing [133]. Agents extend the actor model with the notion of *agency*—the ability of the agent to engage independently with its environment.

An agent a is defined by its *behavior* B and *local state* S. The behavior of an agent encompasses a set of *actions* $x \in X$ (procedures that the agent can perform), and a set of *control loops* $c \in C$ that define the autonomous behavior [121]. Agents are often long-running, but may also be ephemeral—created to complete a specific task and then exiting. Clients and agents can request another agent to perform an action on their behalf through message passing. An action can be atomic or composite, invoking other actions on the same or remote agents. An agent with actions but no control loops (i.e., $|X| > 0$ and $|C| = 0$) reduces to an actor.

Agents come in many flavors (previously discussed in Section 5.1.1). *Intelligent (deliberative) agents* are goal-oriented and reason about what actions to take using internal models and external perception [257]. *AI agents*, a subset of intelligent agents, use AI methods to make decisions or perform actions. In contrast, *reactive (observer) agents* simply perceive their external environment and react to changes [188]. *Service agents* provide predefined services in response to action requests and come in many forms: *resource agents* manage

and grant access to resources, such as compute or storage, and *embodied agents* can act in the world, such as through physical actions when paired with a robot body. *Learning agents* adapt their behavior over time to improve performance, often through reinforcement learning [178, 179]. *Composite agents* exhibit two or more of these behaviors. For example, deliberative learning agents improve their reasoning or planning capabilities over time, and reactive service agents perform services in response to environmental changes. A *multi-agent system* can enable more complex behaviors than monolithic programs [190], which can lead to powerful emergent behavior [76].

An agentic workflow can be formalized as a graph of *actions*—rather than tasks, as in typical DAG-based workflows—where agents request and perform actions on behalf of one another, enabling dynamic coordination and the collective pursuit of complex, distributed objectives. Let the *environment* E represent the external state space, influencing and influenced by the actions of agents and other entities. Agents in the environment are represented by a *deployment* $d(A, R) : A \to R$ of agents $a \in A$ on to *resources* $r \in R$. Each agent implements a behavior, and an agent that knows the behavior of a peer agent can request the peer to perform an action through message passing. Thus, there exists a directed graph representing the peer relationships between the agents; an edge $e = (a_i, a_j)$ in this graph implies that a_i knows of a_j and can request actions from a_j. Sink nodes in the graph represent agents that only perform atomic actions, whereas source nodes may represent deliberative or reactive agents that trigger actions on other agents. A cut vertex (articulation point) in the graph can represent an agent that serves as an interface or gateway to another, possibly more complex, multi-agent system.

The deployment of agents can execute workflows. An *agentic workflow* W is modeled as a directed graph where nodes are a tuple (x, a) of an action to perform and the agent performing the action, and edges representing the source agent and action that triggered the subsequent action. A workflow is typically implicitly encoded within the agent behaviors of

a multi-agent system. I.e., the graph W is not explicitly materialized and agents do not need to know W in order to execute. An agent only needs to be concerned with its local view of executing requested actions and requesting actions from peers. Thus, workflows may often be highly dynamic as agent behaviors react to changing states.

5.3 Academy Design

Designing a middleware that can express the diverse demands of scientific applications and leverage federated research infrastructure is challenging. In the design of Academy, we aim to address the following high-level challenges: How to represent, in code, the declaration of and interaction between agents? How to deploy agents across federated infrastructure? How to achieve performance across heterogeneous systems, networks, and storage? Thus, we begin by outlining key requirements, before we introduce the high-level architecture and detailed implementation choices. The name Academy alludes to societies of artists and scholars that, while independent, collaborate and share similar goals.

5.3.1 Requirements

Writing scientific applications as agentic workflows, rather than using traditional workflow models, can require a considerable shift in conceptual thinking. To reduce developer friction, our design emphasizes familiarity—using well-known programming patterns—and simplicity—providing a small set of primitives and inviting users to invent new patterns and techniques. With these principles in mind, we define requirements in four areas:

- **Representation:** Agent behavior must be expressed in code, supporting control loops, actions that can be performed, and local state. Multiple agents may be instantiated with the same behavior. Agents should not share state.
- **Interaction:** Agents and clients must be able to interact. They must be able to

address a specific agent, perform one or more actions, modify local state, or create and terminate agents.

- **Communication:** Agents and clients communicate asynchronously and are temporally decoupled (e.g., a message sent to an offline agent should be read when that agent is next online). Agents may be deployed in diverse environments with heterogeneous network environments (e.g., asymmetric networks and firewalls restricting connections).

- **Execution:** An agent performs actions in response to requests from clients or other agents. Agent control loops may run in perpetuity or exit before the end of the agent's lifetime. Agents may be launched using different mechanisms that are dependent on the application or environment.

These requirements are an extension of actor systems; therefore, our system inherits properties including simplified concurrency, message processing ordering, loose coupling, error isolation, and modularity [133].

Our implementation focuses on mechanism rather than policy. That is, we discuss how applications can use ACADEMY to achieve certain outcomes without prescribing what agents, or more generally, applications should do. This avoids constraining, or worse, alienating possible use cases and results in a flexible framework suitable for solving many disparate problems. Further, we describe the components within the architecture in terms of abstract interfaces (i.e., without mandating implementation details such as message protocols, state formats, or ordering) to enable further experimentation and optimization, but we still aim to provide implementations that are suitable for most use cases (as demonstrated in the evaluation). Features such as fault tolerance and resilience, resource allocation, and authentication and authorization, while important, are not listed as explicit requirements because applications have varying demands that preclude one-size-fits-all solutions.

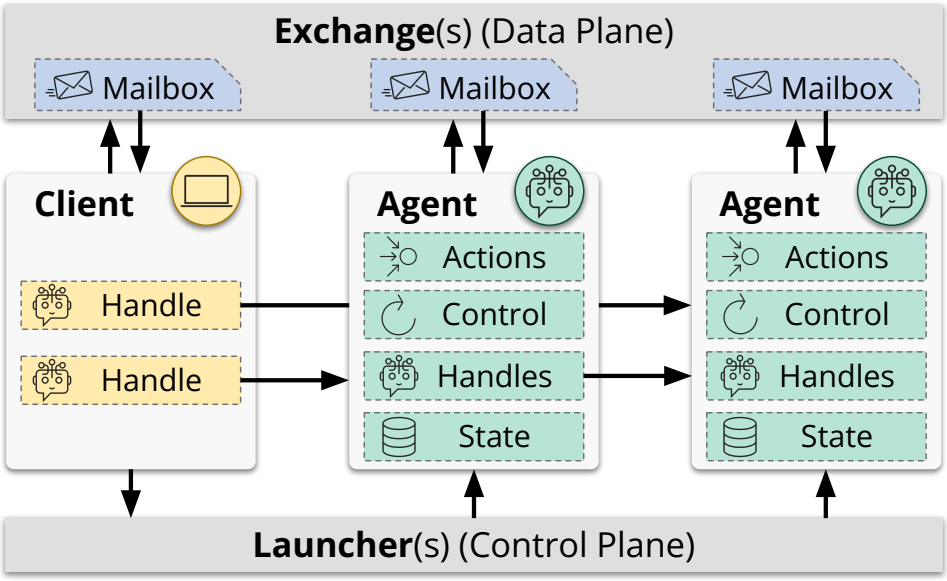


Figure 5.4: Agents and clients in ACADEMY interact via handles to invoke actions asynchronously. Agents implement a behavior, defined by their actions, control loops, and state. ACADEMY decouples the control and data planes through the launcher and exchange components that manage spawning agents and communication, respectively.

5.3.2 Architecture

ACADEMY is a middleware for expressing agentic workflows and deploying multi-agent systems across federated resources. Its architecture strongly decouples the implementation of agent behavior from execution and communication to simplify the development of new agents while maintaining flexibility in deployment.

As depicted at a high level in Figure 5.4, an ACADEMY deployment includes one or more *agents* and zero or more *clients*. An agent is a process that executes a *behavior*, where a behavior is defined by a local state, a set of actions, and a set of control loops. Agents are executed remotely using a launcher. Once running, an agent concurrently executes all of its control loops and listens for messages from clients, which can be other agents or programs.

A client interacts with an agent through a *handle*, a term we borrow from actor frameworks. A handle acts like a reference to the remote agent and translates method calls into action request messages. Each entity (i.e., client or agent) has an associated *mailbox* that

```python
import time, threading
from academy.behavior import Behavior, action, loop

class Example(Behavior):
    def __init__(self) -> None:
        # State stored as attributes
        self.count = 0

    def on_setup(self) -> None:
        pass

    def on_shutdown(self) -> None:
        pass

    @action
    def square(self, value: float) -> float:
        return value**2

    @loop
    def count(self, shutdown: threading.Event) -> None:
        while not shutdown.is_set():
            self.count += 1
            time.sleep(1)
```

Figure 5.5: Example agent behavior definition. State is stored as instance attributes, `@action` decorated methods define actions that other clients and agents can invoke, and `@loop` decorated methods define control loops that run when the agent starts. The `on_setup()` and `on_shutdown()` methods define callbacks invoked when the agent starts and stops, respectively.

maintains a queue of messages sent to that entity by other entities. Mailboxes are maintained by an *exchange* such that any client with access to a given exchange can send messages to the mailbox of another agent in the exchange and receive a response through its own mailbox.

5.3.3 Implementation Details

ACADEMY is implemented as an open-source Python library, available on GitHub [6]. We target Python for its broad compatibility with scientific workflow codes and libraries, but both the architecture and individual components could be implemented in other languages.

Behavior: An agent behavior is implemented as a Python class that inherits from the base `Behavior` type, as shown in Figure 5.5. This class-based approach is simple, so

existing code can be easily transformed into agents, and extensible through inheritance and polymorphism. Instance attributes maintain the agent's state, and methods define the actions and control loops.

The `@action` decorator marks a method as an action, allowing other entities to invoke it remotely. (In the future, we plan to support adding metadata to the `@action` behavior to aid discovery discussed in Figure 5.3.3.) A behavior can invoke actions on itself, as actions are simply Python methods. Methods not decorated as `@action` are private to the behavior. The `@loop` decorator marks a method as a control loop. Control loops are executed in separate threads, so a shared `threading.Event` is passed as an argument to each loop that signals when the agent is shutting down so that control loops can gracefully exit. A control loop can terminate early and the agent will remain running. Commonly, control loops are used to execute a routine on a regular interval, such as to check the state of the environment, or in response to an event. We provide two special control loop decorators, `@timer` and `@event`, that simplify behavior implementations for these scenarios.

Two special methods, `on_setup()` and `on_shutdown()`, allow behaviors to define callbacks when starting or shutting down, such as to load/store state or initialize/destroy resources. Multiple inheritance of behaviors enables the creation of composite agents.

Agent: An `Agent` is a multithreaded entity that executes a behavior and manages communication with other entities. It is instantiated with a behavior, unique identifier (the address of the agent's mailbox in the exchange), and exchange interface. An agent is a callable object that when run: (1) invokes the `on_setup()` callback of the behavior, (2) starts each `@loop` method in a separate thread, (3) spawns a thread to listen for new messages in the agent's mailbox, and (4) waits for the agent to be shut down. An `@action` method is executed in a thread pool when requested remotely so as to not block the handling of other messages. Behaviors can optionally specify the maximum action concurrency.

Agents are designed to be long-running, but can be terminated by sending a shutdown

request. Upon shutdown, the shutdown `Event`, passed to each `@loop`, is set; running threads are instructed to shutdown and waited on; and the `on_shutdown()` callback is invoked. Alternatively, an agent can terminate itself by setting the shutdown event. Similarly, an exception raised in an `@loop` method will shutdown the agent by default but can optionally be suppressed to keep the agent alive. Exceptions raised when executing `@action` methods are caught and returned to the remote caller.

The use of multi-threading means that behavior implementations must be aware of the caveats of Python's global interpreter lock (GIL). Compute-heavy actions can dispatch work to other parallel executors, such as process pools, Dask Distributed [219], Parsl [27], or Ray [184]. We discuss these patterns in more detail in Section 5.3.4. In the future, we would like to support `async` behaviors and exchanges for improved I/O performance, but scientific computing libraries in Python are not typically async compatible. In Python 3.13 and later, we support free-threading builds, which disable the GIL, enabling full multi-core performance. At this time, however, third-party library support for free-threading builds is limited.

Our decision to decouple behavior definitions from agent execution is deliberate. As behaviors encode application-level logic, we want them to be easily testable and reusable, independent of deployment details. Existing code bases can trivially transition an existing class definition into an agent by inheriting from `Behavior` and decorating with `@action` as needed, and behavior classes can still be used independently (i.e., not as a running agent).

Handles: Interacting with an agent is asynchronous; an entity sends a message to the agent's mailbox and waits to receive a response message in its own mailbox. A `handle` is a client interface to a remote agent used to invoke actions, ping, and shutdown the agent. Each handle acts as a reference to that agent, translating each method call into a request message that is sent via the exchange and returning a `Future`. The handle also listens for response messages and accordingly sets the result on the appropriate `Future`. Rather than

creating a return mailbox and listener thread for each handle that a client or agent may have, ACADEMY will multiplex communication for multiple handles within a single process through a single mailbox. This multiplexing ensures that only one mailbox listener thread is needed per process (i.e., agent or client). For example, consider an agent A with n handles to n other agents. It would be inefficient to create a new mailbox for each of A's n handles, so each handle is bound to A's mailbox at runtime.

Exchange: Entities communicate by sending and receiving messages to and from mailboxes. An exchange hosts these mailboxes, and the `Exchange` protocol defines the interface to an exchange. Namely, the `Exchange` defines methods for registering new agent or client mailboxes, sending and receiving messages, and creating handles to remote agents. Registering an agent or client involves creating a unique ID for the entity, which is also the address of its mailbox, and initializing that mailbox within the exchange.

A mailbox has two states: open and closed. Open indicates that the entity is accepting messages, even if, for example, an agent has not yet started or is temporarily offline. Closed indicates permanent termination of the entity and will cause `MailboxClosedError` to be raised by subsequent send or receive operations to that mailbox.

Exchanges also provide mechanisms for agent discovery by querying based on agent behaviors. This also works with superclasses of behaviors. Consider, for example, a behavior `ProteinFolder` that can fold proteins [19] and another behavior `OpenProteinFolder` that inherits from `ProteinFolder` and specifically uses the OpenFold model [7]. Querying for `ProteinFolder` would return the IDs of all agents inheriting from `ProteinFolder` whereas querying for `OpenProteinFolder` would return only specific agents using the OpenFold model. In the future, agents could provide additional metadata to enhance discovery.

Users can define custom exchanges to address specific hardware or application characteristics. We provide two exchange implementations for local and distributed agent deployments. The *local exchange* stores messages in-memory and is suitable for agents running in separate

threads of a single process, such as when testing.

The *distributed exchange* enables communication between entities across wide-area networks. Core to the distributed exchange is an object store that persists information about registered entities. A hybrid approach is used for message passing: direct messaging is preferred, and indirect message passing via the object store is available as a fallback. Upon startup, an entity writes its location (i.e., address and port) to the object store; peers that want to send a message can attempt to send directly to the entity's address. If the peer is offline or a direct connection fails, such as in the presence of NAT (network address translation) or firewall restrictions, messages are appended to the list of pending messages in the object store. Entities continuously listen to incoming messages from peers and pending messages in the object store. Entities cache successful communication routes locally to reduce queries to the object store. Our implementation use TCP (transmission control protocol) sockets for direct messaging and a Redis server as the object store. Redis provides low-latency communication and optional replication, but applications that need greater fault-tolerance could consider DHT-based (distributed hash table) object stores.

We optimize the exchange for low latency, as control messages are typically small: $O(100)$ bytes. However, action request and response messages can contain arbitrarily sized serialized values for arguments and results that can induce considerable overheads when messages are sent indirectly via the object store. To alleviate these overheads, we pass large values by reference and perform out-of-band data transfers by using ProxyStore [195, 197], introduced in Chapter 3 and which provides pass-by-reference semantics in distributed computing through proxy objects. Proxy objects act like references (cheap to serialize and communicate) and automatically de-reference themselves to the true object using performant data storage and communication methods. For example, ProxyStore can leverage RDMA (remote direct memory access) transfers via Mochi [220] and UCX [226], GridFTP via Globus Transfer [59], and reliable peer-to-peer UDP (user datagram protocol) through NAT hole-punching. ProxyS-

```python
from academy.exchange.thread import ThreadExchange
from academy.launcher.thread import ThreadLauncher
from academy.manager import Manager

with Manager(
    exchange=ThreadExchange(),  # Can be swapped with
    launcher=ThreadLauncher(),  # other implementations
) as manager:
    behavior = Example()  # From Listing 1
    handle = manager.launch(behavior)

    future = handle.square(2)
    assert future.result() == 4

    handle.shutdown()  # Or via the manager
    manager.shutdown(handle.agent_id, blocking=True)
```

Figure 5.6: Example of initialization, spawning, using, and shutting down and agent using the `Manager` interface.

tore also provides two key optimizations useful within ACADEMY: proxies can be forwarded to actions executed on other agents without incurring additional data transfers and proxies can be asynchronously resolved to overlap communication and computation.

Launcher: An agent can be run manually, but the intended method of execution is via a launcher, which manages the initialization and execution of agents on remote resources. The `Launcher` protocol defines a `launch()` method with parameters for the behavior, exchange, and agent ID and returns a handle to the launched agent. Users can create custom implementations; we provide the following four that cover most use cases:

- **Thread:** Runs agents in separate threads of the same process. Useful for local development and testing or for light-weight or I/O bound agents.
- **Process:** Runs agents in separate processes on same machine.
- **Parsl:** Runs agents across the workers of a Parsl Executor [27]. Parsl supports execution across local, remote, and batch compute systems.
- **Globus Compute:** Runs agents across Globus Compute Endpoints [61]. Globus

Compute is a cloud-managed function-as-a-service (FaaS) platform which can execute Python functions across federated compute systems.

The last three launchers support mechanisms to automatically restart agents if they exit unexpectedly. It is common for different agents in an application to be executed with different launchers, but all agents must be registered to the same exchange to interact.

A `Manager` combines an exchange and one or more launchers to provide a single interface for launching, using, and managing agents. Each manager has a single mailbox in the exchange and multiplexes that mailbox across handles to all of the agents that it manages. This reduces boilerplate code, improves communication efficiency, and ensures stateful resources and threads are appropriately cleaned up. An end-to-end example is provided in Figure 5.6.

5.3.4 Common Patterns

We have introduced basic building blocks necessary to build multi-agent systems and deploy agents across federated infrastructure. Now we discuss several common patterns that highlight features of ACADEMY and guide users in building new agentic workflows.

State Checkpoints: Research infrastructure can fail; thus, agents may want to perform periodic state checkpointing. The framework does not enforce a specific checkpointing mechanism, as the format, location, and frequency of checkpoints are highly application specific, but `on_startup()` callbacks can be used to restore state automatically. For convenience, we provide a `State` API that provides a dictionary-like interface and persists values to the local file system.

Migration: Research infrastructure is typically static, so ACADEMY does not require that the launcher provide mechanisms for automatic agent migration. Some launchers, such as Parsl, will restart agents on different workers if node-level failures cause agents to crash. Applications can also manually migrate agents across different launchers using agent shutdown and checkpointing mechanisms. These features are sufficient for users to implement

custom launchers that enable automatic migration, such as to load-balance across resource pools.

Agent Hierarchies: Agents may dynamically need to create and manage child agents, either to offload tasks or to access new behaviors. A parent agent can create new child agents by using the same launcher used to create the parent, or by creating a new launcher. The use of different launchers is common in scenarios where parent agents want to initialize a local multi-agent system. For example, a client may launch an initial set of agents across federated resources using Globus Compute, and then those initial agents spawn more agents on local resources through Parsl.

Resource Pools: High-performance workflows may need to distribute work across many computers. In an agentic model, resource pool allocation can take two forms: *agent managed resource pools* or *agents as resource pools*. In the former, an agent allocates a pool of resources using a parallel computing framework, such as Parsl or Ray, and the agent's actions dispatch work to resources in the pool. In the second pattern, we deploy identical agents across a set of resources and then route action requests across this agent pool (akin to worker pools in HTTP frameworks).

Process-as-a-Service: FaaS systems, such as Globus Compute, provide optimized execution of short-lived, stateless, and ephemeral functions. ACADEMY agents can extend FaaS systems with process-as-a-service capabilities [72], enabling applications to utilize longer-lived, stateful, and isolated processes on-demand.

5.4 Evaluation

We studied the performance characteristics of ACADEMY to answer key questions including: How well does the system scale? How fast can agents be deployed? What is the messaging latency? We also make comparisons to Dask and Ray, two popular frameworks with support for distributed actors in Python. Although ACADEMY agents provide a superset of features

provided by actors, these evaluations contextualize the performance of the framework. In these comparisons, we use the terms agent and actor interchangeably.

We conducted experiments using the Aurora supercomputer at the Argonne Leadership Computing Facility (ALCF), unless otherwise stated. Aurora has 10 624 nodes interconnected by an HPE Slingshot 11 network and a high performance DAOS storage system. Aurora nodes contain two Intel Xeon Max CPUs, each with 52 physical cores and 64 GB of high-bandwidth memory; 512 GB of DDR5 memory per socket; and six 128 GB Intel Data Center Max GPUs. In some cases we also use the Polaris supercomputer at ALCF and the `compute-zen-3` nodes of Chameleon Cloud's CHI@TACC cluster [146]. Polaris has 560 nodes interconnected by an HPE Slingshot 11 network and a 100 PB Lustre file system. Polaris nodes contains one AMD EPYC Milan processor with 32 physical cores, 512 GB of DDR4 memory, and four 40 GB NVIDIA A100 GPUs. Each `compute-zen-3` node contains two 64-core CPUs and 256 GB memory. Experiments were performed using Python 3.10, AutoGen 0.5.1, Dask 2025.2.0, Globus Compute 3.5.0, Parsl 2025.03.03, and Ray 2.43.0.

5.4.1 Weak Scaling

We measure weak scaling performance from two aspects: agent startup and action completion time. The object store of the exchange is located on the head node of the Aurora batch job to best match the behavior of Dask and Ray.

Agent Startup Time: We measure the time to spawn n agents in Figure 5.7 (top). We pre-warm the worker processes by starting and stopping n agents, then record the average startup time over five runs. Specifically, we measure the time between submitting the first agent to receiving a ping from all agents to ensure that they have finished their startup sequence. We configured ACADEMY to use Parsl's High-throughput Executor as the launcher. Ray always spawns a new process per actor and thus does not benefit from pre-warmed workers leading to high startup overheads at smaller scales. The cold start time with ACAD-

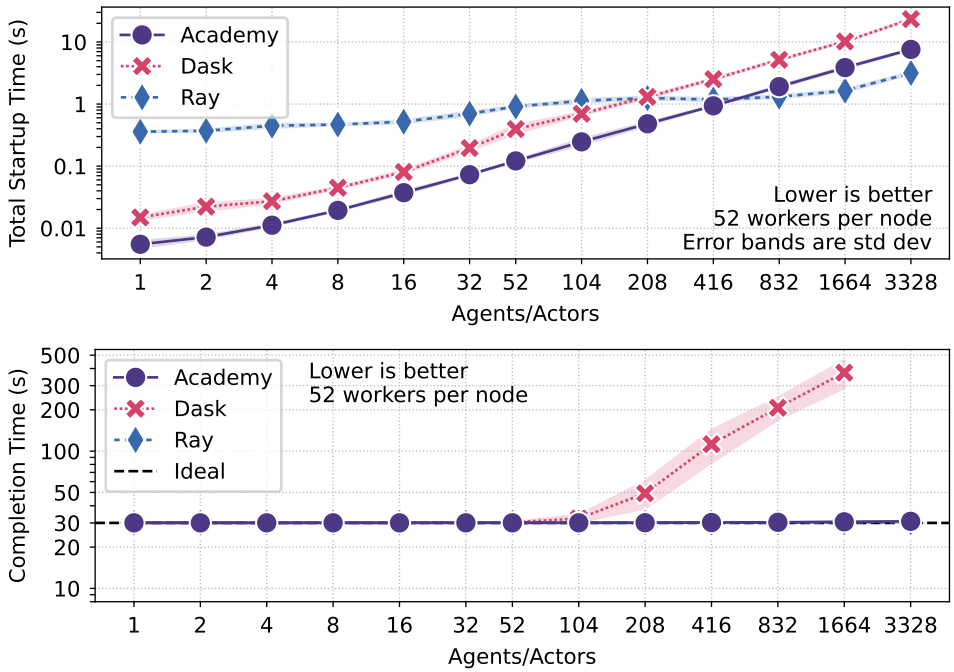


Figure 5.7: (Top) Warm-start time for n agents/actors between ACADEMY (using the Parsl launcher), Dask Actors, and Ray Actors. Ray does not benefit from warm-starts because a new process is spawned for each actor. (Bottom) Time to execute 30 actions per agent/actor (weak scaling). Each action sleeps for 1 s. Note the ACADEMY and Ray lines are overlapped.

EMY and Dask is comparable to that of Ray and dominated by loading libraries from the shared file system. With warm starts, ACADEMY starts a single actor in 5.5 ms, 2.8× faster than Dask. ACADEMY scales well, starting 3328 actors in 7.6 s compared to Dask's 23.4 s, but Ray demonstrates an advantage at this scale with a 3.2 s startup. Since ACADEMY can leverage many launcher types, applications requiring frequent startup of agents can utilize Parsl for low-latency, and applications launching thousands of long-running agents could use Ray.

Action Completion Time: In Figure 5.7 (bottom), we execute 30 sleep tasks (1 s) per agent and record the total completion time. We set the maximum concurrency to 1 for all agents to ensure that tasks are processed sequentially. Completion time remains constant for ACADEMY and Ray up to 3328 agents while the performance of Dask degrades starting at 104 actors.

5.4.2 Distributed Exchange

Next, we study the performance of the distributed exchange.

Data Transfer: We first investigate the pass-by-reference and direct communication optimizations of the distributed exchange. In *baseline*, all message data are communicated indirectly between peers via the exchange's object store. The object store is located remotely on a Chameleon Cloud node. In *pass-by-ref*, messages are still communicated with the object store, but action arguments and results are replaced with references using ProxyStore. ProxyStore is configured to use ZeroMQ (see Section 3.3.1) and ProxyStore's P2P endpoints (see Section 3.3.2) for intra-site and inter-site transfer of referenced objects, respectively. In *direct*, messages are communicated directly between peers, circumventing the cloud-hosted object store; this is only possible when peers are located within the same site.

In Figure 5.8 (top), we measure the time it takes for a client to invoke a no-op action on an agent as a function of input and output payload size. We compare *baseline*, *pass-*

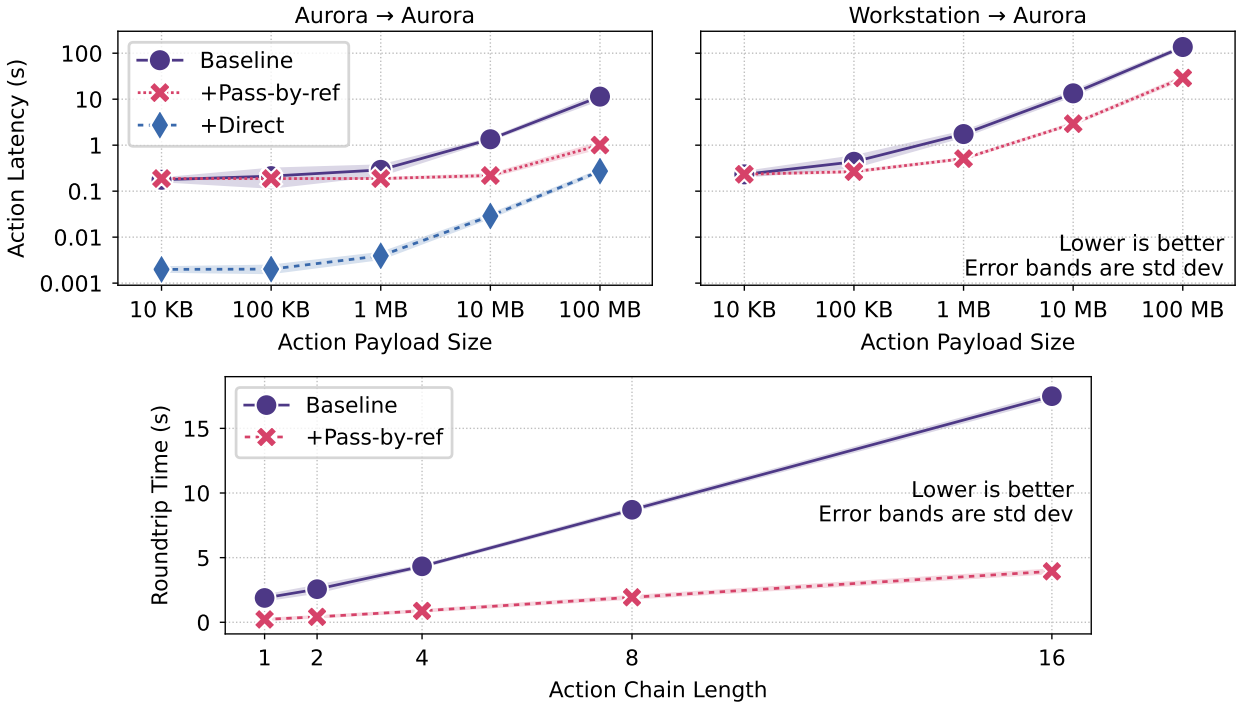


Figure 5.8: (Top) Time for a client to invoke a no-op action on an actor as a function of input and output payload size with different optimizations enabled on the distributed exchange. Two scenarios are considered: client and agent are at the same site (left) and different sites (right). (Bottom) Time for a client to invoke a chain of n actions across n agents with a payload size of 10 MB. Each action in the chain is a no-op that passes the input data along to the next agent, and returns the resulting data. The pass-by-reference optimization reduces communication costs among intermediate actions.

by-ref, and *direct* across two scenarios: *Aurora* $\rightarrow$ *Aurora*, where the client and the agent are located on two different Aurora nodes, and *Workstation* $\rightarrow$ *Aurora*, where the client is located on a personal workstation and the agent is located on an Aurora node. The latencies between the three sites are Aurora to Chameleon: 31 ms; Aurora to Workstation: 12 ms; and Workstation to Chameleon: 42 ms. The workstation is limited to an 800 Mbps internet connection.

We observe that network latency to the exchange object store limits performance at smaller payload sizes ($\leq$ 100 KB). *Direct*, which is possible only in the intra-site scenario, circumvents these latencies. In both scenarios, *pass-by-ref* alleviates overheads of data transfer to and from the object store by communicating data directly between the client and agent via ProxyStore. For intra-site transfers, *pass-by-ref* and *direct* improve action latency compared to the baseline by 91.2% and 97.6%, respectively, with 100 MB payloads. For inter-site transfers, *pass-by-ref* improves action latency by 78.8%.

Pass-by-ref also reduces overheads when actions pass data to subsequent actions, a common pattern in multi-agent systems. We evaluated this optimization by measuring the round-trip time of *action chains* in which data are passed through n actions, each invoked on a separate agent, and results are returned through each agent as well. *Pass-by-ref* reduces the size of messages communicated via the exchange, as indicated by the shallower slope in Figure 5.8 (bottom). Data are only communicated once to the agent that uses the data (here, the last agent in the chain).

Handle Multiplexing: As described in Figure 5.3.3, the communication of multiple handles within a process is multiplexed through a single mailbox. Without this optimization, each handle held by a client process or agent would create a thread for communication. We evaluated this optimization in Figure 5.9 by creating one agent that submits a bag-of-tasks to n worker agents and comparing the task throughput with (*multiplex*) and without (*baseline*) mailbox multiplexing. Multiplexing improves throughput by 41.7% with 52 worker agents

Figure 5.9: Maximum no-op action throughput for a single agent requesting actions from n worker agents. The handle multiplexing optimization improves performance by reducing the number of mailbox listener threads from n to 1.

due to reduced threading overheads.

5.4.3 Agent Messaging

Here, we investigate the performance of agent messaging. As in Section 5.4.1, the object store of the exchange is located on the head node of the Aurora batch job.

Action Latency: In Figure 5.10 (top), we show action latency—the time between sending an action request and receiving a result— between two agents on different nodes. We vary the input/output payload size to understand data transfer overheads. The mean and standard deviation roundtrip latencies are 385±301 μs in ACADEMY, 1186±1059 μs in Dask, and 526±308 μs in Ray for the smallest 10 KB payloads, with latencies increasing with payload size.

Action Throughput: We measure the maximum action throughput for a single agent by submitting a bag of no-op tasks to a pool of worker agents (following the "agents as process pools" pattern from Section 5.3.4) in Figure 5.10 (middle). Increasing the number of agents in the pool ensures that each worker agent is not over-saturated with work. That is, the single submitter agent is the limiting factor for performance. ACADEMY, Dask, and Ray achieve maximum throughputs of 3.6K, 280, and 17.6K action/s, respectively. ACADEMY is 13× faster than Dask but 5× slower than Ray; however, this is a worst case scenario and we

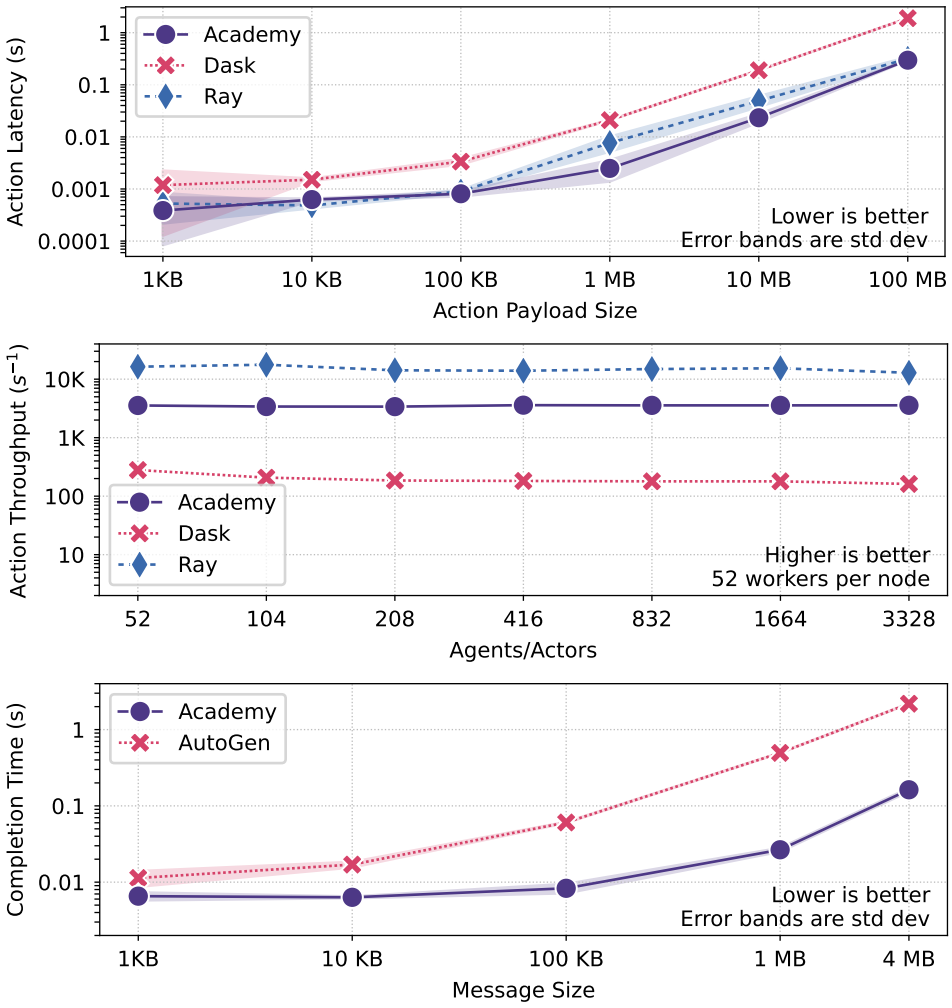


Figure 5.10: (Top) No-op action latency between two agents/actors running on separate Aurora nodes versus action input and output payload size. (Middle) Sustained no-op action throughput for a single worker submitting tasks to a pool of workers. (Bottom) Completion time for a simulated two agent chat where agents send ten messages back and forth with varied message sizes. ACADEMY is compared to AutoGen's distributed runtime.

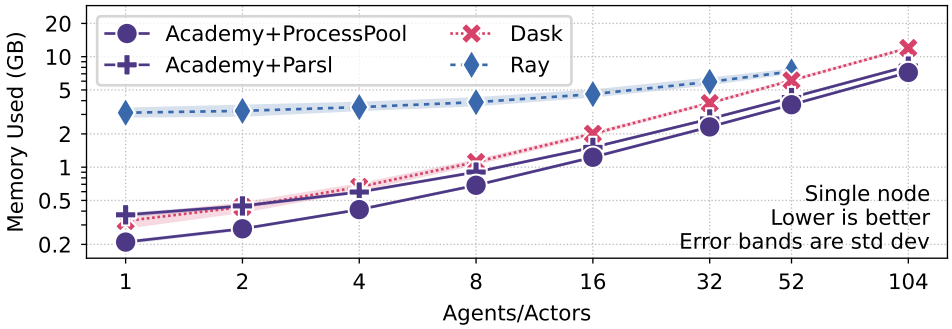


Figure 5.11: Memory used by n agents/actors. We encountered Ray crashes when deploying 104 actors on a single Aurora node (i.e., all cores on both sockets).

believe this performance to be sufficient to not limit realistic applications using ACADEMY.

Agent Conversations: In Figure 5.10 (bottom), we simulate a common pattern in LLM agents where two agents have a back-and-forth conversation. We compare ACADEMY to AutoGen, a popular framework for creating multi-agent AI applications. Each agent is run in a different process on the same node. Agents send ten messages back-and-forth, and we repeat with varying message sizes to simulate different kinds of conversations (i.e., text-only versus multi-modal). AutoGen's distributed agent runtime uses gRPC which has a maximum message size of 4 MB. ACADEMY has comparatively lower overhead messaging in distributed settings.

5.4.4 Memory Overhead

We show memory used as a function of number of agents in Figure 5.11; for ACADEMY, we compare two launchers: a low-overhead but single-node process-pool and Parsl's High-throughput Executor. For fairness, we disable features in Dask and Ray that may reduce performance, such as dashboards, and set the initial Ray object store size to the smallest possible value. ACADEMY agents have low memory overheads, making them suitable for memory-constrained devices, such as when deployed across edge devices via the Globus Compute launcher. The Ray cluster head worker has high memory overhead, but that initial

overhead is amortized as the number of actors is increased, indicating that each actor has modest overhead.

5.5 Case Studies

We use three applications to demonstrate the practicality, generality, and robustness of our system in real-world settings. These examples illustrate how ACADEMY integrates with existing research infrastructure, supports a range of computational patterns, and adapts to the varying demands of scientific applications. They validate key design choices, uncover integration challenges, and provide guidance to researchers building agentic workflows.

5.5.1 Materials Discovery

MOFA [263] is an online learning application for generating, screening, and evaluating metal organic frameworks (MOFs) that couples generative AI methods with computational chemistry. MOFs are polymers composed of inorganic metal clusters and organic ligands; their porosity and large surface area make them suitable for gas adsorption applications such as carbon capture [112]. The goal of MOFA is to generate high-performing candidates by intelligently navigating the intractable combinatorial space of possible MOF structures. MOFA is representative of a broad class of scientific workflows that require careful integration of heterogeneous tasks spanning AI and simulation.

MOFA involves five stages, also summarized in Section 5.1: (1) a generative AI model produces candidate ligands; (2) these ligands are combined with predefined metal clusters to assemble candidate MOFs; (3) the candidates undergo iterative screening and validation using a series of molecular dynamics simulations; (4) CO_2 adsorption properties of the most promising structures are simulated and recorded in a database; and (5) the generative model is periodically retrained on the accumulated results to enhance its performance over time. MOFA utilizes Colmena [249] to coordinate the flow of data between stages and to distribute

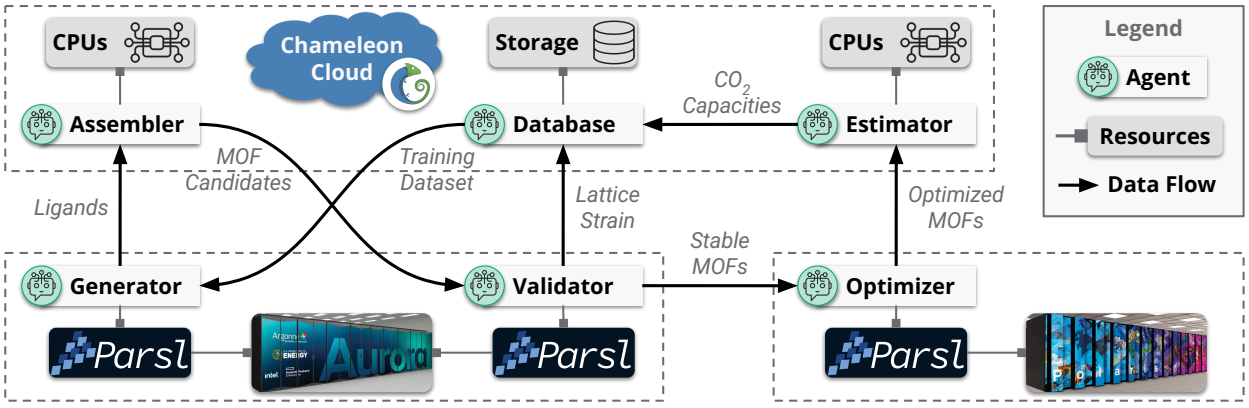


Figure 5.12: We execute MOFA by deploying agents across federated infrastructure using Globus Compute. The Assembler, Database, and Estimator run on Chameleon Cloud nodes with fast single-core performance, the Generator and Validator run on login nodes of Aurora and execute AI and simulation tasks on compute nodes, and the Optimizer runs on a login node of Polaris and executes simulation tasks on compute nodes. Each agent is responsible for a single MOFA stage, and agents cooperate through message passing, such as to request more work and trigger periodic events. The agents on Aurora and Polaris can scale resources in and out based on workload using Parsl.

computations across CPU and GPU resources within a single batch job. However, this design has key limitations: stages cannot be deployed across heterogeneous resources, such as to leverage hardware best optimized for the specific computations; stages cannot independently scale in or out—resources are bound by the size of a single job; integrating new components within tightly coupled code is challenging; and integration with asynchronous processes, such as synthesis in a real laboratory, are infeasible.

MOFA is an excellent candidate for an agentic workflow, as we demonstrate by porting MOFA to use ACADEMY and deploying the workflow across federated resources: see Figure 5.12. We express MOFA through six agents: Database, Generator, Assembler, Validator, Optimizer, and Estimator. Each agent is responsible for a different component of the workflow and manages its own resources (i.e., storage and compute). Agents are remotely deployed across Chameleon Cloud nodes and the login nodes of Aurora and Polaris via Globus Compute, and communicate via the distributed exchange backed by a Redis instance in Chameleon Cloud.

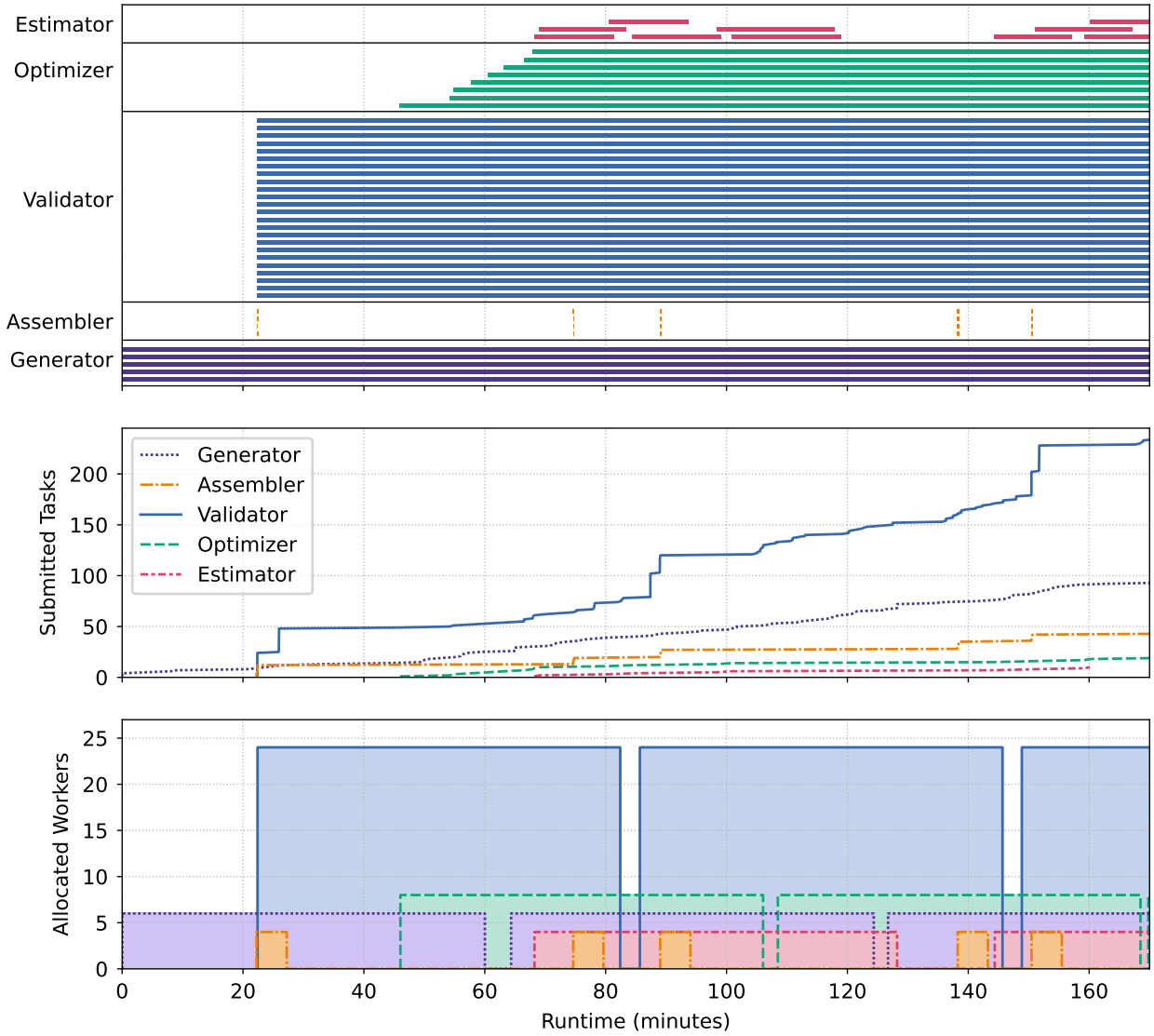


Figure 5.13: Execution trace of the agentic MOFA workflow of Figure 5.12 over three hours. (Top) Active tasks per agent. The vertical axis height represents the maximum size of the resource pool allocated by each agent (i.e., CPUs or GPUs). Assembler tasks are short and infrequent. (Middle) Cumulative tasks submitted per agent. (Bottom) Active workers allocated in each agent's resource pool. Worker allocations vary with demand (as in Assembler and Estimator) or batch job wall times (as in Generator, Validator, and Optimizer).

An execution trace of the agentic MOFA workflow (Figure 5.13) shows how each agent scales out its allocated resources as work becomes available, and in the case of Assembler and Estimator, scale down when their workload decreases. The Generator, Validator, and Optimizer consistently have work to do but their batch jobs within which workers run have 60 minute wall times that expire and then must be resubmitted, causing temporary drops the the number of workers. Active tasks that are killed are automatically restarted in the next job. This separation of concerns is key to enabling long-running workflows—resource infrastructure is not persistently available and agents will need to be able to adapt to that varying availability. A second benefit of this model is the loose coupling between agents. For example, the specific implementation of a given agent can be trivially swapped provided the behavior (i.e., the API that agents expose) remains the same. In addition, it becomes easier to integrate future agents, such as to incorporate embodied agents that interact with self-driving labs to synthesize and evaluate the best-performing MOFs in the real-world. While automated MOF synthesis is not yet practical, the capabilities of self-driving labs are rapidly improving [5, 243], and it is tangible to envision a future where these loosely coupled agentic workflows incorporate services provided by self-driving labs through embodied agents.

5.5.2 Decentralized Learning

In decentralized machine learning a set of models learn collaboratively across distributed datasets [130]. This paradigm is particularly relevant today as data are increasingly generated in decentralized settings and transfer to a centralized location can be infeasible for cost and privacy reasons. Each device in a decentralized learning workflow performs three steps: (1) train a model on local data for a set number of iterations; (2) receive models from neighboring devices and send its own model to neighbors; and (3) update the local model via an all reduce operation performed across its own and received models. Reframing the decentralized learning workflow as an agentic workflow is a natural and powerful extension.

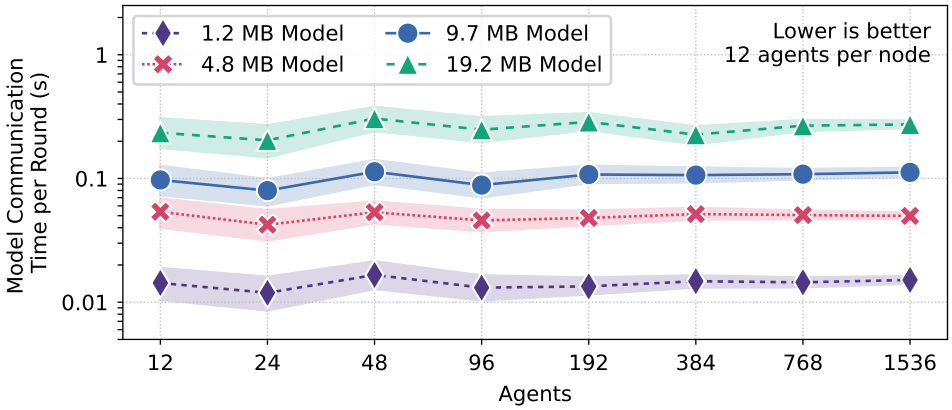


Figure 5.14: Model communication time to an agent's neighbors averaged over five rounds of decentralized training. Training time and aggregation time are excluded since they are nearly constant.

We implement a decentralized and asynchronous machine learning exemplar using ACADEMY. The agents and the communication channels between them can be represented as a graph where nodes are agents and edges are communication channels. We choose a powerlaw cluster graph to approximate real-world networks [136]. Each agent is responsible for training its local model, receiving neighboring agents' models, and aggregating received models with its own model on a periodic basis. Each agent uses a copy of the MNIST dataset [87]. The agents are configured to use *pass-by-ref* with ProxyStore as the transfer backend. Thus data communication between agents follows the network topology. We investigate the cost of distributing updates from all agents as we scale the size of the graph for different model sizes in Figure 5.14. We do not show training and aggregation time as it is approximately the same for all model sizes and does not increase with the number of agents. The agents are deployed on Aurora using Parsl, where each agent is pinned to a single GPU tile (two tiles per physical GPU), allowing 12 agents per node. Our results demonstrate ACADEMY's ability to support more than 1500 autonomous agents working collaboratively with no client coordination (as can be seen by the constant time in Figure 5.14).

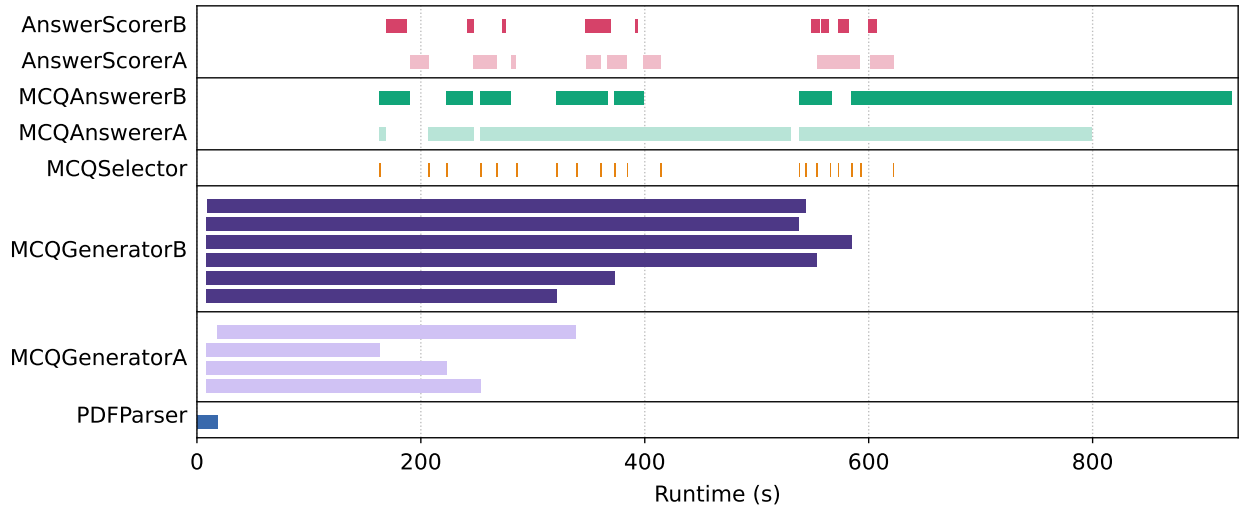


Figure 5.15: Execution trace of the agentic MCQ workflow processing 10 manuscripts to generate and validate questions and answers over 15 minutes. The figure shows the active agents and the duration of their tasks. Agents employ either the Mistral-7B-Instruct-v0.3 or Meta-Llama-3-70B-Instruct model, denoted A and B, respectively.

5.5.3 Information Extraction

Exponential growth in scientific publications [44] creates potential for cross-disciplinary insights that are largely untapped due to the limitations of manual literature review. Automating information extraction from this vast and varied body of work using AI is crucial to accelerate scientific progress. AI methods can be employed to identify and synthesize key findings, methodologies, and datasets across fields and thus to identify connections and facilitate the cross-pollination of ideas that would otherwise go unnoticed [48, 232].

Agentic workflows that leverage LLMs present a transformative new approach to engage with scientific literature. Employing autonomous agents with specific roles and capabilities makes it possible to automate the extraction of information and generation of structured datasets that represent key concepts and findings. Such datasets can be used to fine-tune models and enhance their ability to understand scientific text, answer domain-specific queries, and potentially contribute to tasks like hypothesis generation or literature summarization.

To explore the potential of agentic workflows for thus analyzing the scientific literature we used ACADEMY to implement a system for generating and validating multi-choice questions (MCQs) from research publications [57]. The workflow, based on Catlett and Foster 2025, includes a PDFParser agent to extract text from a manuscript; two Generator agents that use different LLMs to generate MCQs; an MCQSelector to choose subsets of questions to evaluate; and two MCQAnswerers and two AnswerScorers (again, each with a different LLM) to generate and validate, respectively, answers to questions. The agents use the Mistral-7B-Instruct-v0.3 [143] and Meta-Llama-3-70B-Instruct [125] models, denoted A and B, respectively.

The beauty of this architecture is that alternative tasks and LLMs are easily integrated by defining new agents; agents can scale up and down in response to demand; and different agents can run concurrently or at different times. We show in Figure 5.15 an execution trace from a run in which the agents just listed were run concurrently to generate and validate MCQs for 10 publications.

5.6 Related Work

A **workflow** is a structured sequence of tasks, typically a directed acyclic graph (DAG), designed to achieve a specific goal, often involving data transformation, analysis, or computational modeling. Frameworks for building workflows take many forms. Parallel computing libraries, such as Dask [219] and Ray [184], provide mechanisms for executing functions in parallel across local resources or distributed systems. Similarly, workflow management systems (WMSs) can execute tasks in parallel but also provide mechanisms for defining, optimizing, and monitoring DAG execution (e.g., Airflow [22], Fireworks [141], Makeflow [13], Nextflow [89], Parsl [27], Pegasus [86], Swift [255]). WMSs can be differentiated by how dependency graphs are defined [196]: static configurations files, such as CWL [75], XML, or YAML; general purpose languages (GPLs); domain specific languages (DSLs); or procedu-

rally through the dynamic execution of a program. The class of workflows supported by these frameworks have two key limitations that we address: tasks are assumed to be pure (i.e., no side-effects) and programs are static, i.e., they cannot adapt to changing environments over time.

Actors are computational entities that enable concurrent computing through message passing [133]. In response to a message, an actor can alter its local state, send messages to other actors, and create new actors. This conceptual model is simple; no global state means locks and other synchronization primitives are not required. Actors can enable stateful computations within traditionally stateless programming models, and are supported in parallel computing frameworks (e.g., Akka [10], Dask, Orleans [35], Ray) and function-as-a-service (FaaS) platforms (e.g., Abaco [116], Azure Service Fabric [26], PraaS [72]). Actor models have been investigated as alternatives for designing computational workflows where communication and coordination are decoupled [45]. Our system extends the actor model to support autonomous behaviors and federated deployments.

Multi-agent systems can enhance or automate scientific processes. Early work investigated cooperative agent environments for distributed problem solving with minimal human intervention [94, 93]. Recent work focuses on improving the reasoning capabilities of LLM-backed agents through ontological knowledge graphs and multi-agent systems [118] and tool-augmented LLMs [167]. Increasingly popular is the use of multi-agent conversations, in which multiple role-specialized agents interact to collaborate, coordinate, or compete towards goals [259]. These systems enhance LLM-based tools through better reasoning [96], validation [260], and divergent thinking [162], prompting rapid development of frameworks such as LangGraph [155], Microsoft AutoGen [259], OpenAI Swarm [189], and Pydantic Agents [208]. Subsequently, interest in standardizing agent protocols has developed. Anthropic's Model Context Protocol (MCP) defines structured interaction between humans/-tools and AI models. Google's Agent2Agent (A2A) Protocol [122] focuses on structured

interaction between autonomous agents; each agent serves an HTTP endpoint which is impracticable for many scientific workflows [20]. Multi-agent conversations can proxy scientists in iterative scientific processes—brainstorming ideas, planning experiments, and reasoning about results [42, 245, 115, 124]—but these aforementioned systems are designed for local or cloud-native applications and lack the features necessary to deploy agents across federated research infrastructure. We focus on the systems-level challenges of representing and deploying diverse agent types and agentic workflows across heterogeneous environments rather than the applied use of LLMs for workflow steering.

5.7 Summary

Advancements in AI, coupled with concurrent advancements in self-driving laboratories, high performance computing, and research data management, open the door for truly autonomous scientific discovery. Realizing this grand vision requires mechanisms for the seamless and dynamic integration of research software and infrastructure. To that end, we introduced Academy, a middleware for developing agentic workflows that engage multi-agent systems spanning federated reeearch infrastructure. This framework enables scalable and flexible orchestration of intelligent agents across heterogeneous resources. We presented solutions to three key challenges: representing and programming agents; communicating among agents; and executing agents across diverse resources. Our evaluations demonstrate that Academy can support high-performance workflows, and three case studies highlight the advantages of agentic workflow design.

In future work, we will explore scoped authentication to control which agents can invoke others, enabling the creation of agent marketplaces where access can be granted, revoked, or delegated. We also plan to expand agent discovery with additional metadata to support AI-steered workflows in which LLMs autonomously identify and use available agents. Recording the relative ordering of agent events (i.e., messages received and state transitions), as in

Instant Replay [158], can support provenance within agentic workflows. Last but not least, we will work across scientific research communities to assemble agents for different purposes, and with research facilities to identify obstacles to agent use that may motivate further developments in ACADEMY.

CHAPTER 6

SUMMARY AND IMPACT

In this thesis, I explored new programming techniques with the goal of enabling and accelerating task-centric scientific applications that leverage the computing continuum. By addressing key challenges in distributed execution paradigms, data flow management, and agentic workflows, this work has contributed foundational methodologies and practical implementations that enhance the flexibility, scalability, and performance of distributed scientific applications.

In TaPS, I introduced a benchmarking suite designed to evaluate parallel task execution frameworks. The lack of evaluation standards has historically hindered comparisons between existing systems, making it difficult to assess their limitations and potential improvements. By providing reference workloads and a unified interface for evaluating execution frameworks and data management systems, TaPS has supported continued research and development in task-based execution models.

The increasing complexity of modern scientific workflows necessitates a rethinking of traditional techniques and abstractions for building distributed and task-parallel applications. Building on insights from TaPS and numerous case studies, I introduced a new paradigm that addresses the key challenge of data flow in such environments—namely, the efficient and scalable exchange of intermediate data between tasks. Through the development of ProxyStore's transparent object proxy paradigm, I have demonstrated how data flow can be decoupled from control flow, enabling both pass-by-reference and pass-by-value semantics that simplify application design while improving performance. Further extending the proxy paradigm, I presented high-level patterns that extend the utility of proxies in distributed computing, including patterns for distributed futures, streaming, and ownership. These patterns provide developers with tools to streamline the creation of scalable, efficient applications in a variety of scientific domains.

Recognizing the shift toward intelligent, autonomous workflows in scientific discovery, I also explored the role of multi-agent systems in computational science. The development of ACADEMY, a middleware solution for expressing and deploying autonomous agents across federated resources, represents a step toward fully automated scientific workflows. By enabling agent-based execution models, this work lays the foundation for more adaptive and self-sustaining computational science applications.

The experimental evaluations and real-world applications presented in this thesis confirm the effectiveness of the proposed techniques, facilitating new discoveries in fields ranging from materials science to bioinformatics. Within a short period of time, the impact of PROXYSTORE has been notable, yielding strong collaborations across a breadth of ongoing projects and featuring in three Gordon Bell Prize submissions, including the project that won the 2022 Gordon Bell Special Prize for COVID-19 Research. TAPS has found immediate success in supporting research in novel data management systems, resilient computing, and cost-aware scheduling.

These contributions collectively advance the state of the art in computational science, enabling researchers to tackle larger, more complex problems with greater efficiency and ease. Looking ahead, the continued evolution of these techniques will further accelerate scientific discovery, ultimately pushing the boundaries of what is computationally possible.

REFERENCES

[1] 1000 Genomes Project, n.d. https://www.internationalgenome.org/1000-genomes-summary/. Accessed Mar 2024.

[2] 1000 Genomes Workflow, n.d. https://github.com/pegasus-isi/1000genome-workflow/. Accessed Mar 2024.

[3] 2MASS Image Dataset, n.d. http://montage.ipac.caltech.edu/docs/Kimages.tar. Accessed May 2024.

[4] 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 *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[5] Milad Abolhasani and Eugenia Kumacheva. The rise of self-driving labs in chemical and materials sciences. *Nature Synthesis*, 2(6):483–492, 2023.

[6] Academy GitHub, n.d. https://github.com/proxystore/academy. Accessed Apr 2025.

[7] Gustaf Ahdritz, Nazim Bouatta, Christina Floristean, Sachin Kadyan, Qinghui Xia, William Gerecke, Timothy J O'Donnell, Daniel Berenberg, Ian Fisk, Niccolò Zanichelli, et al. OpenFold: Retraining AlphaFold2 yields new insights into its learning mechanisms and capacity for generalization. *Nature Methods*, 21(8):1514–1524, 2024.

[8] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(08):26–34, 1986. doi:10.1109/MC.1986.1663305.

[9] AioRTC Package, n.d. https://github.com/aiortc/aiortc. Accessed Sep 2024.

[10] Akka: the Actor Model on the JVM, n.d. https://akka.io/. Accessed Sep 2024.

[11] Mehmet Fatih Aktas, Javier Diaz-Montes, Ivan Rodero, and Manish Parashar. WA-Dataspaces: Exploring the Data Staging Abstractions for Wide-Area Distributed Scientific Workflows. In *46th International Conference on Parallel Processing*, pages 251–260, 2017. doi:10.1109/ICPP.2017.34.

[12] Aymen Al-Saadi, Dong H Ahn, Yadu Babuji, Kyle Chard, James Corbett, Mihael Hategan, Stephen Herbein, Shantenu Jha, Daniel Laney, Andre Merzky, Todd Munson, Michael Salim, Mikhail Titov, Matteo Turilli, and Justin M. Wozniak. ExaWorks: Workflows for exascale. In *IEEE Workshop on Workflows in Support of Large-Scale Science*, pages 50–57. IEEE, 2021.

[13] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450318761. doi:10.1145/2443416.2443417.

[14] Bryce Allen, John Bresnahan, Lisa Childers, Ian Foster, Gopi Kandaswamy, Raj Kettimuthu, Jack Kordas, Mike Link, Stuart Martin, Karl Pickett, and Steven Tuecke. Software as a Service for Data Scientists. *Communications of the ACM*, 55(2):81–88, Feb 2012. ISSN 0001-0782. doi:10.1145/2076450.2076468.

[15] Bryce Allen, John Bresnahan, Lisa Childers, Ian Foster, Gopi Kandaswamy, Raj Kettimuthu, Jack Kordas, Mike Link, Stuart Martin, Karl Pickett, and Steven Tuecke. Software as a Service for Data Scientists. *Communications of the ACM*, 55(2): 81–88, feb 2012. ISSN 0001-0782. doi:10.1145/2076450.2076468. URL `https://doi.org/10.1145/2076450.2076468`.

[16] Aymen Alsaadi, Logan Ward, Andre Merzky, Kyle Chard, Ian Foster, Shantenu Jha, and Matteo Turilli. Radical-Pilot and Parsl: Executing heterogeneous workflows on HPC platforms. *arXiv preprint arXiv:2105.13185*, 2021.

[17] Amazon Lambda, n.d. `https://aws.amazon.com/lambda`. Accessed Sep 2024.

[18] Rachana Ananthakrishnan, Yadu Babuji, Josh Bryan, Kyle Chard, Ryan Chard, Ben Clifford, Ian Foster, Lev Gorenstein, Kevin Hunter Kesling, Chris Janidlo, Daniel Katz, Reid Mello, J. Gregory Pauloski, and Lei Wang. Establishing a High-Performance and Productive Ecosystem for Distributed Execution of Python Functions Using Globus Compute. In *IEEE/ACM International Workshop on HPC User Support Tools (HUST)*, 2024. doi:10.1109/SCW63240.2024.00083.

[19] Christian B Anfinsen. Principles that govern the folding of protein chains. *Science*, 181(4096):223–230, 1973.

[20] Anthropic. Model Context Protocol (MCP), 2024. URL `https://modelcontextprotocol.io/`.

[21] Anthropic. Tool use with Claude, 2025. URL `https://docs.anthropic.com/en/docs/build-with-claude/tool-use/overview`.

[22] Apache Airflow, n.d. `https://airflow.apache.org/`. Accessed Sep 2024.

[23] Apache Kafka, n.d. `https://kafka.apache.org/`. Accessed Sep 2024.

[24] Apache OpenWhisk, n.d. `http://openwhisk.apache.org/`. Accessed Sep 2024.

[25] AutoDock Vina: Python Scripting, n.d. `https://github.com/ccsb-scripps/AutoDock-Vina/tree/develop/example/python_scripting`. Accessed May 2024.

[26] Azure: Service Fabric Reliable Actors, n.d. https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-actors-introduction. Accessed Sep 2024.

[27] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Luksaz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive Parallel Programming in Python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2019. doi:10.1145/3307681.3325400.

[28] Timon Back and Vasilios Andrikopoulos. Using a Microbenchmark to Compare Function as a Service Solutions. In Kyriakos Kritikos, Pierluigi Plebani, and Flavio de Paoli, editors, *Service-Oriented and Cloud Computing*, pages 146–160, Cham, 2018. Springer International Publishing. ISBN 978-3-319-99819-0.

[29] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[30] Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, page 55–59, New York, NY, USA, 1977. Association for Computing Machinery. ISBN 9781450378741. doi:10.1145/800228.806932.

[31] André Bauer, Haochen Pan, Ryan Chard, Yadu Babuji, Josh Bryan, Devesh Tiwari, Ian Foster, and Kyle Chard. The Globus Compute Dataset: An open function-as-a-service dataset from the edge to the cloud. *Future Generation Computer Systems*, 153:558–574, 2024. ISSN 0167-739X. doi:https://doi.org/10.1016/j.future.2023.12.007. URL https://www.sciencedirect.com/science/article/pii/S0167739X23004703.

[32] Georg Bauer. Linda Tuple Spaces for Python, n.d. https://pypi.org/project/lindypy/. Accessed Sep 2024.

[33] Olivier Beaumont, Lionel Eyraud-Dubois, Mathieu Vérité, and Julien Langou. I/O-Optimal Algorithms for Symmetric Linear Algebra Kernels. In *ACM Symposium on Parallelism in Algorithms and Architectures*, 2022. URL https://hal.inria.fr/hal-03580531.

[34] Juan Benet. IPFS – Content Addressed, Versioned, P2P File System, 2014. https://arxiv.org/abs/1407.3561.

[35] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSRTR2014*, 41, 2014.

[36] GB Berriman, JC Good, DW Curkendall, JC Jacob, DS Katz, TA Prince, and R Williams. An On-Demand Image Mosaic Service for the NVO. In *Astronomical Data Analysis Software and Systems XII*, volume 295, page 343, 2003.

[37] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão, and NicholasD.Lane. Flower: A friendly federated learning framework. *arXiv preprint arXiv:2007.14390*, 2022.

[38] David I Bevan. Distributed garbage collection using reference counting. In *PARLE Parallel Architectures and Languages Europe: Volume II: Parallel Languages Eindhoven, The Netherlands, June 15–19, 1987 Proceedings 1*, pages 176–187. Springer, 1987.

[39] Tekin Bicer, Doga Gursoy, Rajkumar Kettimuthu, Ian T. Foster, Bin Ren, Vincent De Andrede, and Francesco De Carlo. Real-Time Data Analysis and Autonomous Steering of Synchrotron Light Source Experiments. In *IEEE 13th International Conference on e-Science*, pages 59–68, 2017. doi:10.1109/eScience.2017.53.

[40] Gregory Biegel, Vinny Cahill, and Mads Haahr. A Dynamic Proxy Based Architecture to Support Distributed Java Objects in a Mobile Environment. In *OTM Conferences / Workshops*, volume 2519, pages 809–826, 11 2002. ISBN 978-3-540-00106-5. doi:10.1007/3-540-36124-3_54.

[41] Niklas Blum, Serge Lachapelle, and Harald Alvestrand. WebRTC: Real-Time Communication for the Open Web Platform. *Communications of the ACM*, 64(8):50–54, jul 2021. ISSN 0001-0782. doi:10.1145/3453182. URL `https://doi.org/10.1145/3453182`.

[42] Daniil A Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. Autonomous chemical research with large language models. *Nature*, 624(7992):570–578, 2023.

[43] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 836–850. ACM, 2021. ISBN 9781450387095. doi:10.1145/3477132.3483540.

[44] Lutz Bornmann, Robin Haunschild, and Rüdiger Mutz. Growth rates of modern science: A latent piecewise growth curve approach to model publication numbers from established and new literature databases. *Humanities and Social Sciences Communications*, 8(1):1–15, 2021.

[45] Shawn Bowers and Bertram Ludäscher. Actor-Oriented Design of Scientific Workflows. In Lois Delcambre, Christian Kop, Heinrich C. Mayr, John Mylopoulos, and Oscar Pastor, editors, *Conceptual Modeling – ER 2005*, pages 369–384, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32068-5.

[46] A. Brace, I. Yakushin, H. Ma, A. Trifan, T. Munson, I. Foster, A. Ramanathan, H. Lee, M. Turilli, and S. Jha. Coupling streaming AI and HPC ensembles to achieve

100–1000× faster biomolecular simulations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 806–816, Los Alamitos, CA, USA, jun 2022. IEEE Computer Society. doi:10.1109/IPDPS53621.2022.00083. URL https://doi.ieeecomputersociety.org/10.1109/IPDPS53621.2022.00083.

[47] Alexander Brace, Logan Ward, Heng Ma, and Arvind Ramanathan. DeepDriveMD, n.d. URL https://github.com/ramanathanlab/deepdrivemd. Accessed Mar 2024.

[48] Rüdiger Buchkremer, Alexander Demund, Stefan Ebener, Fabian Gampfer, David Jägering, Andreas Jürgens, Sebastian Klenke, Dominik Krimpmann, Jasmin Schmank, Markus Spiekermann, Michael Wahlers, and Markus Wiepke. The application of artificial intelligence technologies as a substitute for reading and to support and enhance the authoring of scientific review articles. *IEEE access*, 7:65263–65276, 2019.

[49] C++ std::futures, n.d. https://en.cppreference.com/w/cpp/thread/future. Accessed Oct 2024.

[50] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *23rd ACM Symposium on Operating Systems Principles*, page 143–157. ACM, 2011. ISBN 9781450309776. doi:10.1145/2043556.2043571.

[51] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14, September-October:20 – 53, 2016. URL http://queue.acm.org/detail.cfm?id=3022184.

[52] Nicholas J Carriero, David Gelernter, Timothy G Mattson, and Andrew H Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–655, 1994. ISSN 0167-8191. doi:https://doi.org/10.1016/0167-8191(94)90032-9.

[53] Lorenzo Casalino, Abigail C Dommer, Zied Gaieb, Emilia P Barros, Terra Sztain, Surl-Hee Ahn, Anda Trifan, Alexander Brace, Anthony T Bogetti, Austin Clyde, et al. AI-driven multiscale simulations illuminate mechanisms of SARS-CoV-2 spike dynamics. *The International Journal of High Performance Computing Applications*, 35(5):432–451, 2021.

[54] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014. URL http://hal.inria.fr/hal-01017319.

[55] Henri Casanova, Rafael Ferreira da Silva, Ryan Tanaka, Suraj Pandey, Gautam Jethwani, William Koch, Spencer Albrecht, James Oeth, and Frédéric Suter. Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH. *Future Generation Computer Systems*, 112:162–175, 2020. doi:10.1016/j.future.2020.05.030.

[56] Charlie Catlett and Ian Foster. Creating and Scoring Multiple Choice Questions (MCQs) from Papers, 2025. URL `https://github.com/auroraGPT-ANL/MCQ-and-SFT-code`.

[57] Dhawaleswar Rao Ch and Sujan Kumar Saha. Automatic multiple choice question generation from text: A survey. *IEEE Transactions on Learning Technologies*, 13(1): 14–25, 2018.

[58] K Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 124–144. Springer, 1992.

[59] Kyle Chard, Steven Tuecke, and Ian Foster. Efficient and Secure Transfer, Synchronization, and Sharing of Big Data. *IEEE Cloud Computing*, 1(3):46–55, 2014. doi:10.1109/MCC.2014.52.

[60] Kyle Chard, Eli Dart, Ian Foster, David Shifflett, Steven Tuecke, and Jason Williams. The Modern Research Data Portal: A design pattern for networked, data-intensive science. *PeerJ Computer Science*, 4:e144, 2018.

[61] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. funcX: A Federated Function Serving Fabric for Science. In *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2020. doi:10.1145/3369583.3392683.

[62] Kenneth Chiu, Madhusudhan Govindaraju, and Dennis Gannon. The Proteus multiprotocol message library. In *ACM/IEEE Conference on Supercomputing*, pages 30–30. IEEE, 2002.

[63] Jong Youl Choi, Tahsin Kurc, Jeremy Logan, Matthew Wolf, Eric Suchyta, James Kress, David Pugmire, Norbert Podhorszki, Eun-Kyu Byun, Mark Ainsworth, Manish Parashar, and Scott Klasky. Stream processing for near real-time scientific data analysis. In *2016 New York Scientific Data Summit (NYSDS)*, pages 1–8, 2016. doi:10.1109/NYSDS.2016.7747804.

[64] Joaquin Chung, Wojciech Zacherek, AJ Wisniewski, Zhengchun Liu, Tekin Bicer, Rajkumar Kettimuthu, and Ian Foster. SciStream: Architecture and Toolkit for Data Streaming between Federated Science Instruments. In *31st International Symposium on High-Performance Parallel and Distributed Computing*, page 185–198. ACM, 2022. ISBN 9781450391993. doi:10.1145/3502181.3531475.

[65] Michele Ciavotta, Davide Motterlini, Marco Savi, and Alessandro Tundo. DFaaS: Decentralized Function-as-a-Service for Federated Edge Computing. In *10th International Conference on Cloud Networking*, 2021.

[66] Keith Clark and Steve Gregory. PARLOG: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[67] Keith L Clark and Steve Gregory. A relational language for parallel programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 171–178, 1981.

[68] Austin Clyde, Xuefeng Liu, Thomas Brettin, Hyunseung Yoo, Alexander Partin, Yadu Babuji, Ben Blaiszik, Jamaludin Mohd-Yusof, Andre Merzky, Matteo Turilli, et al. AI-accelerated protein-ligand docking for SARS-CoV-2 is 100-fold faster with no significant change in detection. *Scientific Reports*, 13(1):2105, 2023.

[69] Tainã Coleman, Henri Casanova, Loïc Pottier, Manav Kaushik, Ewa Deelman, and Rafael Ferreira da Silva. WfCommons: A framework for enabling scientific workflow research and development. *Future Generation Computer Systems*, 128:16–27, 2022. ISSN 0167-739X. doi:https://doi.org/10.1016/j.future.2021.09.043.

[70] Nicholson Collier, Justin M. Wozniak, Abby Stevens, Yadu Babuji, Mickaël Binois, Arindam Fadikar, Alexandra Würth, Kyle Chard, and Jonathan Ozik. Developing Distributed High-performance Computing Capabilities of an Open Science Platform for Robust Epidemic Analysis, 2023.

[71] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.

[72] Marcin Copik, Alexandru Calotoiu, Rodrigo Bruno, Gyorgy Rethy, Roman Böhringer, and Torsten Hoefler. Process-as-a-Service: Elastic and Stateful Serverless with Cloud Processes. Technical report, ETH Zürich, 01 2022.

[73] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. FMI: Fast and Cheap Message Passing for Serverless Functions. In *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, page 373–385, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700569. doi:10.1145/3577193.3593718. URL `https://doi.org/10.1145/3577193.3593718`.

[74] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 897–907, 2023. doi:10.1109/IPDPS54959.2023.00094.

[75] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojša Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilović, Carole

Goble, and The CWL Community. Methods included: standardizing computational reuse and portability with the Common Workflow Language. *Commun. ACM*, 65(6): 54–63, May 2022. ISSN 0001-0782. doi:10.1145/3486897. URL https://doi.org/10.1145/3486897.

[76] Felipe Cucker and Steve Smale. On the mathematics of emergence. *Japanese Journal of Mathematics*, 2:197–227, 2007.

[77] Eli Dart, Lauren Rotman, Brian Tierney, Mary Hester, and Jason Zurawski. The Science DMZ: A Network Design Pattern for Data-Intensive Science. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13. ACM, 2013. ISBN 9781450323789. doi:10.1145/2503210.2503245.

[78] Anirban Das, Stacy Patterson, and Mike Wittie. EdgeBench: Benchmarking Edge Computing Platforms. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 175–180, 2018. doi:10.1109/UCC-Companion.2018.00053.

[79] Dask Benchmarks, n.d. https://github.com/dask/dask-benchmarks. Accessed Sep 2024.

[80] Dask Best Practices, n.d. https://docs.dask.org/en/stable/best-practices.html#load-data-with-dask. Accessed Sep 2024.

[81] Dask GitHub, n.d. https://github.com/dask/dask. Accessed Sep 2024.

[82] Dask PyPI Statistics, n.d. https://pypistats.org/packages/distributed. Accessed Sep 2024.

[83] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association. URL https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters.

[84] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: The Montage example. In *SC'08: ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008. doi:10.1109/SC.2008.5217932.

[85] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.

[86] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. ISSN 0167-739X. doi:https://doi.org/10.1016/j.future.2014.10.008.

[87] Li Deng. The MNIST Database of Handwritten Digit Images for Machine Learning Research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[88] Gautham Dharuman, Logan Ward, Heng Ma, Priyanka V. Setty, Ozan Gokdemir, Sam Foreman, Murali Emani, Kyle Hippe, Alexander Brace, Kristopher Keipert, Thomas Gibbs, Ian Foster, Anima Anandkumar, Venkatram Vishwanath, and Arvind Ramanathan. Protein Generation via Genome-scale Language Models with Bio-physical Scoring. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 95–101, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707858. doi:10.1145/3624062.3626087. URL https://doi.org/10.1145/3624062.3626087.

[89] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[90] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows. In *19th ACM International Symposium on High Performance Distributed Computing*, page 25–36. ACM, 2010. ISBN 9781605589428. doi:10.1145/1851476.1851481.

[91] Abigail Dommer, Lorenzo Casalino, Fiona Kearns, Mia Rosenfeld, Nicholas Wauer, Surl-Hee Ahn, John Russo, Sofia Oliveira, Clare Morris, Anthony Bogetti, et al. # COVIDisAirborne: AI-enabled multiscale computational microscopy of delta SARS-CoV-2 in a respiratory aerosol. *The International Journal of High Performance Computing Applications*, 37(1):28–44, 2023.

[92] J. J. Dongarra. The LINPACK Benchmark: An Explanation. In *1st International Conference on Supercomputing*, page 456–474, Berlin, Heidelberg, 1988. Springer-Verlag. ISBN 0387189912.

[93] Tzvetan Drashansky, Elias N Houstis, Naren Ramakrishnan, and John R Rice. Networked agents for scientific computing. *Communications of the ACM*, 42(3):48–ff, 1999.

[94] Tzvetan T Drashansky, Anupam Joshi, and John R Rice. SciAgents-an agent based environment for distributed, cooperative scientific computing. In *7th IEEE International Conference on Tools with Artificial Intelligence*, pages 452–459. IEEE, 1995.

[95] Ron O Dror, Robert M Dirks, JP Grossman, Huafeng Xu, and David E Shaw. Biomolecular simulation: A computational microscope for molecular biology. *Annual review of biophysics*, 41:429–452, 2012.

[96] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving Factuality and Reasoning in Language Models through Multiagent Debate, 2023. URL https://arxiv.org/abs/2305.14325.

[97] Greg Eisenhauer, Norbert Podhorszki, Ana Gainaru, Scott Klasky, Philip E Davis, Manish Parashar, Matthew Wolf, Eric Suchtya, Erick Fredj, Vicente Bolea, et al. Streaming Data in HPC Workflows Using ADIOS. *arXiv preprint arXiv:2410.00178*, 2024.

[98] Enron. Enron Email Corpus, n.d. `https://www.cs.cmu.edu/~enron/`. Accessed May 2024.

[99] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003. ISSN 0360-0300. doi:10.1145/857076.857078.

[100] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. Godot: All the Benefits of Implicit and Explicit Futures. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:28, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-111-5. doi:10.4230/LIPIcs.ECOOP.2019.2.

[101] Rafael Ferreira da Silva, Weiwei Chen, Gideon Juve, Karan Vahi, and Ewa Deelman. Community Resources for Enabling Research in Distributed Scientific Workflows. In *IEEE 10th International Conference on eScience*, volume 1, 10 2014. doi:10.1109/eScience.2014.44.

[102] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011. URL `http://www.rfc-editor.org/rfc/rfc6455.txt`. `http://www.rfc-editor.org/rfc/rfc6455.txt`.

[103] Rosa Filguiera, Iraklis Klampanos, Amrey Krause, Mario David, Alexander Moreno, and Malcolm Atkinson. dispel4py: A Python Framework for Data-Intensive Scientific Computing. In *International Workshop on Data Intensive Scalable Computing Systems*, pages 9–16, 2014. doi:10.1109/DISCS.2014.12.

[104] FN Project, n.d. `https://fnproject.io`. Accessed Sep 2024.

[105] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-Peer Communication across Network Address Translators. In *USENIX Annual Technical Conference*, page 13, USA, 2005. USENIX Association. doi:10.48550/ARXIV.CS/0603074. `https://arxiv.org/abs/cs/0603074`.

[106] Ian Foster. Globus Online: Accelerating and Democratizing Science through Cloud-Based Services. *IEEE Internet Computing*, 15(3):70–73, 2011. doi:10.1109/MIC.2011.64.

[107] Ian Foster and Nicholas T Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *ACM/IEEE Conference on Supercomputing*, pages 46–46. IEEE, 1998. doi:10.1109/SC.1998.10051.

[108] Ian Foster and Stephen Taylor. *Strand: New concepts in parallel programming*. Prentice-Hall, Inc., 1989.

[109] Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, 1992.

[110] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steven Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40(1):35–48, 1997.

[111] Daniel P. Friedman and David Wise. Aspects of Applicative Programming for Parallel Processing. *IEEE Transactions on Computers*, C-27(4):289–296, 1978. doi:10.1109/TC.1978.1675100.

[112] Hiroyasu Furukawa, Kyle E Cordova, Michael O'Keeffe, and Omar M Yaghi. The chemistry and applications of metal-organic frameworks. *Science*, 341(6149):1230444, 2013.

[113] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. doi:10.1007/978-3-540-30218-6_19.

[114] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612. URL `http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1`.

[115] Shanghua Gao, Ada Fang, Yepeng Huang, Valentina Giunchiglia, Ayush Noori, Jonathan Richard Schwarz, Yasha Ektefaie, Jovana Kondic, and Marinka Zitnik. Empowering biomedical discovery with AI agents. *Cell*, 187(22):6125–6151, 2024. ISSN 0092-8674. doi:https://doi.org/10.1016/j.cell.2024.09.022. URL `https://www.sciencedirect.com/science/article/pii/S0092867424010705`.

[116] Christian Garcia, Joe Stubbs, Julia Looney, Anagha Jamthe, and Mike Packard. Abaco–A Modern Platform for High Throughput Parallel Scientific Computations. In *International Workshop on Science Gateways*, 2020.

[117] Sara Ghaemi, Hamzeh Khazaei, and Petr Musilek. ChainFaaS: An open blockchain-based serverless platform. *IEEE Access*, 8, 2020.

[118] Alireza Ghafarollahi and Markus J Buehler. SciAgents: Automating Scientific Discovery Through Bioinspired Multi-Agent Intelligent Graph Reasoning. *Advanced Materials*, page 2413523, 2024.

[119] Carole Goble, Stian Soiland-Reyes, Finn Bacall, Stuart Owen, Alan Williams, Ignacio Eguinoa, Bert Droesbeke, Simone Leo, Luca Pireddu, Laura Rodríguez-Navas, José Mª Fernández, Salvador Capella-Gutierrez, Hervé Ménager, Björn Grüning, Beatriz Serrano-Solano, Philip Ewels, and Frederik Coppens. Implementing FAIR Digital Objects in the EOSC-Life Workflow Collaboratory, 2021. URL https://doi.org/10.5281/zenodo.4605654.

[120] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX*, 12: 100561, 2020. ISSN 2352-7110. doi:https://doi.org/10.1016/j.softx.2020.100561. URL https://www.sciencedirect.com/science/article/pii/S2352711019302560.

[121] Richard Goodwin. Formalizing properties of agents. *Journal of Logic and Computation*, 5(6):763–781, 1995.

[122] Google. Agent2Agent Protocol (A2A), 2025. URL https://github.com/google/A2A.

[123] Google Cloud, n.d. https://cloud.google.com/functions/. Accessed Sep 2024.

[124] Google Research. Accelerating scientific breakthroughs with an AI co-scientist, 2025. URL https://research.google/blog/accelerating-scientific-breakthroughs-with-an-ai-co-scientist/.

[125] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi,

Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Pre-

sani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang,

Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The Llama 3 Herd of Models, 2024. URL https://arxiv.org/abs/2407.21783.

[126] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *12th ACM Symposium on Operating Systems Principles*, page 202–210, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913388. doi:10.1145/74850.74870.

[127] Mark Handley, Van Jacobson, and Colin Perkins. SDP: Session description protocol. RFC 4566, IETF, 2006. https://www.rfc-editor.org/rfc/rfc4566.html.

[128] Mark A. Hanson, Pablo Gómez Barreiro, Paolo Crosetto, and Dan Brockington. The strain on scientific publishing. *Quantitative Science Studies*, 5(4):823–843, 11 2024. ISSN 2641-3337. doi:10.1162/qss_a_00327. URL https://doi.org/10.1162/qss_a_00327.

[129] Hassan Harb, Sarah N. Elliott, Logan Ward, Ian T. Foster, Stephen J. Klippenstein, Larry A. Curtiss, and Rajeev Surendran Assary. Uncovering novel liquid organic hydrogen carriers: a systematic exploration of chemical compound space using cheminformatics and quantum chemical methods. *Digital Discovery*, 2:1813–1830, 2023. doi:10.1039/D3DD00123G. URL http://dx.doi.org/10.1039/D3DD00123G.

[130] István Hegedűs, Gábor Danner, and Márk Jelasity. Gossip learning as a decentralized alternative to federated learning. In *Distributed Applications and Interoperable Systems: 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 19*, pages 74–90. Springer, 2019.

[131] Michael Hennecke. DAOS: A Scale-out High Performance Storage Stack for Storage Class Memory. *Supercomputing frontiers*, 40, 2020.

[132] Chathura Herath and Beth Plale. Streamflow Programming Model for Data Streaming in Scientific Workflows. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 302–311, 2010. doi:10.1109/CCGRID.2010.116.

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

[134] Tony Hey, Stewart Tansley, Kristin Tolle, and Jim Gray. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, October 2009. ISBN 978-0-9825442-0-4. URL https://www.microsoft.com/en-us/research/publication/fourth-paradigm-data-intensive-scientific-discovery/.

[135] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[136] Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Physical review E*, 65(2):026107, 2002.

[137] Adam Hospital, Josep Ramon Goñi, Modesto Orozco, and Josep L Gelpí. Molecular dynamics simulations: Advances and applications. *Advances and Applications in Bioinformatics and Chemistry*, pages 37–47, 2015.

[138] Nathaniel Hudson, Valerie Hayot-Sasson, Yadu Babuji, Matt Baughman, J. Gregory Pauloski, Ryan Chard, Ian Foster, and Kyle Chard. Flight: A FaaS-Based Framework for Complex and Hierarchical Federated Learning, 2024. URL `https://arxiv.org/abs/2409.16495`.

[139] William Humphrey, Andrew Dalke, and Klaus Schulten. VMD: Visual Molecular Dynamics. *Journal of Molecular Graphics*, 14(1):33–38, 1996.

[140] Tal Ifargan, Lukas Hafner, Maor Kern, Ori Alcalay, and Roy Kishony. Autonomous LLM-Driven Research—from Data to Human-Verifiable Research Papers. *NEJM AI*, 2(1):AIoa2400555, 2025.

[141] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, Daniel Gunter, and Kristin A. Persson. FireWorks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015.

[142] Emmanuel Jeannot. Performance Analysis and Optimization of the Tiled Cholesky Factorization on NUMA Machines. In *5th International Symposium on Parallel Architectures, Algorithms and Programming*, pages 210–217, 2012. doi:10.1109/PAAP.2012.38.

[143] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B, 2023. URL `https://arxiv.org/abs/2310.06825`.

[144] Alok Kamatar, Mansi Sakarvadia, Valerie Hayot-Sasson, Kyle Chard, and Ian Foster. Lazy Python Dependency Management in Large-Scale Systems. In *IEEE 19th International Conference on e-Science*, pages 1–10. IEEE, 2023.

[145] Daniel S. Katz, Andre Merzky, Zhao Zhang, and Shantenu Jha. Application skeletons: Construction and use in eScience. *Future Generation Computer Systems*, 59:114–124, 2016. ISSN 0167-739X. doi:https://doi.org/10.1016/j.future.2015.10.001.

[146] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons Learned from the Chameleon Testbed. In *USENIX Annual Technical Conference*. USENIX Association, July 2020.

[147] Ari Keranen, Christer Holmberg, and Jonathan Rosenberg. Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal. RFC 8445, IETF, 2018. URL `https://www.rfc-editor.org/rfc/rfc8445.html`.

[148] Carl Kesselman, Robert Schuler, and Ian Foster. Let's Put the Science in eScience. In *IEEE 19th International Conference on e-Science*, pages 1–3, 2023. doi:10.1109/e-Science58273.2023.10254914.

[149] Jeongchul Kim and Kyungyong Lee. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019. doi:10.1109/CLOUD.2019.00091.

[150] KNIX MicroFunctions, n.d. `https://github.com/knix-microfunctions/knix`. Accessed Sep 2024.

[151] Nikita Kotsehub, Matt Baughman, Ryan Chard, Nathaniel Hudson, Panos Patros, Omer Rana, Ian Foster, and Kyle Chard. FLoX: Federated Learning with FaaS at the Edge. In *IEEE 18th International Conference on e-Science (e-Science)*, pages 11–20, 2022. doi:10.1109/eScience55777.2022.00016.

[152] Alex Krizhevsky, Geoffrey Hinton, et al. Learning Multiple Layers of Features from Tiny Images, 2009.

[153] Jakub Lála, Odhran O'Donoghue, Aleksandar Shtedritski, Sam Cox, Samuel G Rodriques, and Andrew D White. PaperQA: Retrieval-Augmented Generative Agent for Scientific Research, 2023.

[154] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi:10.1145/279227.279229. URL `https://doi.org/10.1145/279227.279229`.

[155] LangChain. LangGraph, 2024. URL `https://www.langchain.com/langgraph`.

[156] LangChain. How to do tool/function calling, 2025. URL `https://python.langchain.com/docs/how_to/function_calling/`.

[157] Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.

[158] Thomas Leblanc and John Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 100(4):471–482, 1987.

[159] H. Lee, M. Turilli, S. Jha, D. Bhowmik, H. Ma, and A. Ramanathan. DeepDriveMD: Deep-Learning Driven Adaptive Molecular Simulations for Protein Folding. In *IEEE/ACM Third Workshop on Deep Learning on Supercomputers*, pages 12–19, Los Alamitos, CA, USA, nov 2019. IEEE Computer Society. doi:10.1109/DLS49591.2019.00007. URL https://doi.ieeecomputersociety.org/10.1109/DLS49591.2019.00007.

[160] Claus-Werner Lermen and Dieter Maurer. A protocol for distributed reference counting. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, page 343–350, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 0897912004. doi:10.1145/319838.319875.

[161] Liang Liang, Rosa Filgueira, Yan Yan, and Thomas Heinis. Scalable adaptive optimizations for stream-based workflows in multi-HPC-clusters and cloud infrastructures. *Future Generation Computer Systems*, 128:102–116, 2022. ISSN 0167-739X. doi:https://doi.org/10.1016/j.future.2021.09.036.

[162] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. Encouraging Divergent Thinking in Large Language Models through Multi-Agent Debate. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 17889–17904, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi:10.18653/v1/2024.emnlp-main.992. URL https://aclanthology.org/2024.emnlp-main.992/.

[163] Zhen Liang, Johann Lombardi, Mohamad Chaarawi, and Michael Hennecke. DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In Dhabaleswar K. Panda, editor, *Supercomputing Frontiers*, pages 40–54, Cham, 2020. Springer International Publishing. ISBN 978-3-030-48842-0.

[164] libp2p: the Peer-to-Peer Network Stack, n.d. https://libp2p.io/. Accessed Sep 2024.

[165] Steven S Lumetta, Alan M Mainwaring, and David E Culler. Multi-protocol active messages on a cluster of SMP's. In *ACM/IEEE Conference on Supercomputing*, pages 1–22, 1997.

[166] Jinfeng Ma, Lin Li, Haofan Wang, Yi Du, Junjie Ma, Xiaoli Zhang, and Zhenliang Wang. Carbon Capture and Storage: History and the Road Ahead. *Engineering*, 14: 33–43, 2022.

[167] Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, Aixin Sun, Hany Awadalla, et al. SciAgent: Tool-augmented language models for scientific reasoning. *arXiv preprint arXiv:2402.11451*, 2024.

[168] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[169] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. FaaSdom: A Benchmark Suite for Serverless Computing. In *14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380287. doi:10.1145/3401025.3401738.

[170] Nikolay Malitsky, Ralph Castain, and Matt Cowan. Spark-MPI: Approaching the Fifth Paradigm of Cognitive Applications, 2018. URL `https://arxiv.org/abs/1806.01110`.

[171] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.

[172] Alberto Riccardo Martinelli, Massimo Torquati, Marco Aldinucci, Iacopo Colonnelli, and Barbara Cantalupo. CAPIO: a Middleware for Transparent I/O Streaming in Data-Intensive Workflows. In *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2023. doi:10.1109/HiPC58850.2023.00031.

[173] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. MLPerf training benchmark. *Proceedings of Machine Learning and Systems*, 2:336–349, 2020.

[174] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

[175] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, USA, 1994.

[176] Microsoft Azure, n.d. `https://azure.microsoft.com/en-us/services/functions/`. Accessed Sep 2024.

[177] William L. Miller, Deborah Bard, Amber Boehnlein, Kjiersten Fagnan, Chin Guok, Eric Lançon, Sreeranjani Ramprakash, Mallikarjun Shankar, Nicholas Schwarz, and

Benjamin L. Brown. Integrated Research Infrastructure Architecture Blueprint Activity (Final Report 2023). Technical report, US Department of Energy (USDOE), Washington, DC (United States). Office of Science; Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA (United States), 07 2023. URL https://www.osti.gov/biblio/1984466.

[178] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[179] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra andShane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[180] MOF Generation on HPC, n.d. https://github.com/globus-labs/mof-generation-at-scale/. Accessed Mar 2024.

[181] Montage. Montage: An Astronomical Image Mosaic Engine, n.d. http://montage.ipac.caltech.edu/. Accessed Mar 2024.

[182] Montage Tutorial. Getting Started: Creating Your First Montage Mosaic, n.d. http://montage.ipac.caltech.edu/docs/first_mosaic_tutorial.html. Accessed May 2024.

[183] Luc Moreau and Jean Duprat. A construction of distributed reference counting. *Acta Informatica*, 37:563–595, 2001.

[184] 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 *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association. ISBN 9781931971478.

[185] Randall Munroe. How Standards Proliferate, Jul 2011. https://xkcd.com/927/. Accessed Jan 2025.

[186] Mypy: Static Typing for Python, n.d. https://github.com/python/mypy. Accessed Apr 2025.

[187] Rishiyur S Nikhil and Keshav K Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[188] Hyacinth S. Nwana. Software Agents: An Overview. *The Knowledge Engineering Review*, 11(3):205–244, 1996. doi:10.1017/S026988890000789X.

[189] OpenAI. Swarm, 2024. URL https://github.com/openai/swarm.

[190] Liviu Panait and Sean Luke. Cooperative multi-agent learning: The state of the art. *Autonomous agents and multi-agent systems*, 11:387–434, 2005.

[191] Dhabaleswar Kumar Panda, Hari Subramoni, Ching-Hsiang Chu, and Mohammadreza Bayatpour. The MVAPICH project: Transforming research into high-performance MPI library for HPC community. *Journal of Computational Science*, 52:101208, 2021. ISSN 1877-7503. doi:https://doi.org/10.1016/j.jocs.2020.101208.

[192] Papers with Code. Papers with Code, n.d. https://paperswithcode.com/datasets. Accessed May 2024.

[193] Hyun Park, Xiaoli Yan, Ruijie Zhu, Eliu A. Huerta, Santanu Chaudhuri, Donny Cooper, Ian Foster, and Emad Tajkhorshid. A generative artificial intelligence framework based on a molecular diffusion model for the design of metal-organic frameworks for carbon capture. *Communications Chemistry*, 7(1), February 2024. ISSN 2399-3669. doi:10.1038/s42004-023-01090-2. URL http://dx.doi.org/10.1038/s42004-023-01090-2.

[194] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

[195] J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Nathaniel Hudson, Charlie Sabino, Matt Baughman, Kyle Chard, and Ian Foster. Accelerating Communications in Federated Applications with Transparent Object Proxies. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. ISBN 9798400701092. doi:10.1145/3581784.3607047. URL https://doi.org/10.1145/3581784.3607047.

[196] J. Gregory Pauloski, Valerie Hayot-Sasson, Maxime Gonthier, Nathaniel Hudson, Haochen Pan, Sicheng Zhou, Ian Foster, and Kyle Chard. TaPS: A Performance Evaluation Suite for Task-based Execution Frameworks. In *IEEE 20th International Conference on e-Science*, pages 1–10, New York, NY, USA, 2024. IEEE. doi:10.1109/e-Science62913.2024.10678702.

[197] J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Alexander Brace, André Bauer, Kyle Chard, and Ian Foster. Object Proxy Patterns for Accelerating Distributed Applications. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–13, 2024. doi:10.1109/TPDS.2024.3511347.

[198] J. Gregory Pauloski, Klaudiusz Rydzy, Valerie Hayot-Sasson, Ian Foster, and Kyle Chard. Accelerating Python Applications with Dask and ProxyStore, 2024. URL https://arxiv.org/abs/2410.12092.

[199] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[200] Pegasus Examples, n.d. `https://github.com/pegasus-isi/ACCESS-Pegasus-Examples/`. Accessed Sep 2024.

[201] Vu Anh Pham and A. Karmouch. Mobile software agents: An overview. *IEEE Communications Magazine*, 36(7):26–37, 1998. doi:10.1109/35.689628.

[202] James C Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V Kalé. NAMD: Biomolecular simulation on thousands of processors. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 36–36. IEEE, 2002.

[203] José M Piquer. Indirect Reference Counting: A Distributed Garbage Collection Algorithm. In *Parallel Architectures and Languages Europe*, pages 150–165, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-662-25209-3.

[204] ProxyStore GitHub, n.d. `https://github.com/proxystore`. Accessed July 2024.

[205] ProxyStore PyPI, n.d. `https://pypi.org/project/proxystore/`. Accessed July 2024.

[206] Py-Margo, n.d. `https://github.com/mochi-hpc/py-mochi-margo`. Accessed Sep 2024.

[207] PyBullet, n.d. `https://github.com/bulletphysics/bullet3`. Accessed Jul 2024.

[208] Pydantic. Agents, 2024. URL `https://ai.pydantic.dev/agents/`.

[209] Python Concurrent Execution, n.d. `https://docs.python.org/3/library/concurrent.futures.html`. Accessed Sep 2024.

[210] PyTorch RPC, n.d. `https://pytorch.org/docs/2.2/rpc.html#torch.distributed.rpc.rpc_async`. Accessed Sep 2024.

[211] RabbitMQ, n.d. `https://rabbitmq.com/`. Accessed Sep 2024.

[212] Nick Radcliffe, Kent Lee, and Pete Mendygral. Dragon Proxy Runtimes and Multi-system Workflows. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 648–651, New York, NY, USA, 2023. Association for Computing Machinery.

[213] John Raicu, Valerie Hayot-Sasson, Kyle Chard, and Ian Foster. Navigating the Molecular Maze: A Python-Powered Approach to Virtual Drug Screening, 2023. `https://github.com/Parsl/parsl-docking-tutorial`. Accessed May 2024.

[214] Raghunathan Ramakrishnan, Pavlo O Dral, Matthias Rupp, and O Anatole von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1, 2014.

[215] Anand S Rao and Michael P Georgeff. Modeling rational agents within a BDI-architecture. *Readings in agents*, pages 317–328, 1997.

[216] RAPIDS Development Team, n.d. `https://rapids.ai`. Accessed Sep 2024.

[217] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, mesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. MLPerf Inference Benchmark. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture*, pages 446–459. IEEE, 2020.

[218] Redis, n.d. `https://redis.io/`. Accessed Sep 2024.

[219] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *14th Python in Science Conference*, volume 130, page 136, 2015.

[220] Robert Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Zheng Qing. Mochi: Composing Data Services for High-Performance Computing Environments. *Journal of Computer Science and Technology*, 35(1):121 – 144,, Jan 2020. 10.1007/s11390-020-9802-0.

[221] Rust Documentation: The Dot Operator, n.d. `https://doc.rust-lang.org/nomicon/dot-operator.html`. Accessed Sep 2024.

[222] Michael Salim, Thomas Uram, J. Taylor Childers, Venkatram Vishwanath, and Michael Papka. Balsam: Near Real-Time Experimental Data Analysis on Supercomputers. In *1st Annual Workshop on Large-scale Experiment-in-the-Loop Computing*. IEEE, November 2019. doi:10.1109/xloop49562.2019.00010.

[223] Larissa Schmid, Marcin Copik, Alexandru Calotoiu, Laurin Brandner, Anne Koziolek, and Torsten Hoefler. SeBS-Flow: Benchmarking Serverless Cloud Function Workflows, 2024. URL `https://arxiv.org/abs/2410.03480`.

[224] Marten Seemann, Max Inden, and Dimitris Vyzovitis. Decentralized Hole Punching. In *IEEE 42nd International Conference on Distributed Computing Systems Workshops*, pages 96–98. IEEE, 2022.

[225] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2018. doi:10.1109/TPDS.2017.2766062.

[226] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. UCX: An open source framework for HPC network APIs and beyond. In *IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43. IEEE, 2015.

[227] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[228] Barry Sly-Delgado, Thanh Son Phung, Colin Thomas, David Simonetti, Andrew Hennessee, Ben Tovar, and Douglas Thain. TaskVine: Managing In-Cluster Storage for High-Throughput Data Intensive Workflows. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 1978–1988, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400707858.

[229] Snap Inc. KeyDB: A Database Built for Scale, n.d. `https://github.com/Snapchat/KeyDB`. Accessed Sep 2024.

[230] Nikhila Somu, Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 144–151, 2020. doi:10.1109/COMSNETS48256.2020.9027346.

[231] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. Mercury: Enabling remote procedure call for high-performance computing. In *IEEE International Conference on Cluster Computing*, pages 1–8, 2013. doi:10.1109/CLUSTER.2013.6702617.

[232] Jamshid Sourati and James A Evans. Accelerating science with human-aware artificial intelligence. *Nature Human Behaviour*, 7(10):1682–1696, 2023.

[233] Dante Sánchez-Gallegos, José Luis Compeán, Maxime Gonthier, Valerie Hayot-Sasson, Gregory Pauloski, Haochen Pan, Kyle Chard, Ian Foster, and Jesus Carretero Perez. D-Rex: Heterogeneity-Aware Reliability Framework and Adaptive Algorithms for Distributed Storage. In *25th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2025.

[234] TaPS GitHub, n.d. `https://github.com/proxystore/taps`. Accessed July 2024.

[235] The Rust Language, n.d. https://rust-lang.org. Accessed Sep 2024.

[236] Transaction Processing Performance Council, n.d. https://www.tpc.org/. Accessed May 2024.

[237] Anda Trifan, Defne Gorgun, Michael Salim, Zongyi Li, Alexander Brace, Maxim Zvyagin, Heng Ma, Austin Clyde, David Clark, David J Hardy, et al. Intelligent resolution: Integrating Cryo-EM with AI-driven multi-resolution simulations to observe the severe acute respiratory syndrome coronavirus-2 replication-transcription machinery in action. *The International Journal of High Performance Computing Applications*, 36 (5-6):603–623, 2022.

[238] Oleg Trott and Arthur J Olson. AutoDock Vina: Improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of Computational Chemistry*, 31(2):455–461, 2010.

[239] Steven Tuecke, Rachana Ananthakrishnan, Kyle Chard, Mattias Lidman, Brendan McCollam, Stephen Rosen, and Ian Foster. Globus Auth: A research identity and access management platform. In *IEEE 12th International Conference on e-Science*, pages 203–212, 2016. doi:10.1109/eScience.2016.7870901.

[240] UCX-Py, n.d. https://ucx-py.readthedocs.io/en/latest/. Accessed Sep 2024.

[241] UnixBench, n.d. https://github.com/kdlucas/byte-unixbench. Accessed May 2024.

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

[243] Rafael Vescovi, Tobias Ginsburg, Kyle Hippe, Doga Ozgulbas, Casey Stone, Abraham Stroka, Rory Butler, Ben Blaiszik, Tom Brettin, Kyle Chard, et al. Towards a modular architecture for science factories. *Digital Discovery*, 2(6):1980–1998, 2023.

[244] Adam Walker and Michael J Wooldridge. Understanding the Emergence of Conventions in Multi-Agent Systems. In *ICMAS*, volume 95, pages 384–389, 1995.

[245] Hanchen Wang, Tianfan Fu, Yuanqi Du, Wenhao Gao, Kexin Huang, Ziming Liu, Payal Chandak, Shengchao Liu, Peter Van Katwyk, Andreea Deac, Anima Anandkumar, Karianne J. Bergen, Carla P. Gomes, Shirley Ho, Pushmeet Kohli, Joan Lasenby, Jure Leskovec, Tie-Yan Liu, Arjun K. Manrai, Debora S. Marks, Bharath Ramsundar, Le Song, Jimeng Sun, Jian Tang, Petar Velickovic, Max Welling, Linfeng Zhang, Connor W. Coley, Yoshua Bengio, and Marinka Zitnik. Scientific discovery in the age of artificial intelligence. *Nature*, 620:47–60, 2023. URL https://api.semanticscholar.org/CorpusID:260384616.

[246] 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. USENIX Association, April 2021. ISBN 978-1-939133-21-2. URL https://www.usenix.org/conference/nsdi21/presentation/cheng.

[247] Logan Ward and Arham Khan. Real-Time Defect Identification, n.d. https://github.com/ivem-argonne/real-time-defect-analysis. Accessed Mar 2023.

[248] Logan Ward, J. Gregory Pauloski, Valerie Hayot-Sasson, Yadu Babuji, Alexander Brace, Ryan Chard, Kyle Chard, Rajeev Thakur, and Ian Foster. Employing Artificial Intelligence to Steer Exascale Workflows with Colmena. *The International Journal of High Performance Computing Applications*, 0(0):10943420241288242, 0. doi:10.1177/10943420241288242. URL https://doi.org/10.1177/10943420241288242.

[249] Logan Ward, Ganesh Sivaraman, J. Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C. Redfern, Rajeev S. Assary, Kyle Chard, Larry A. Curtiss, Rajeev Thakur, and Ian Foster. Colmena: Scalable Machine-Learning-Based Steering of Ensemble Simulations for High Performance Computing. In *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, pages 9–20, 2021. doi:10.1109/MLHPC54614.2021.00007.

[250] Logan Ward, Kyle Chard, and Ben Clifford. Molecular Design in Parsl, 2023. https://github.com/ExaWorks/molecular-design-parsl-demo. Accessed May 2024.

[251] Logan Ward, J. Gregory Pauloski, Valerie Hayot-Sasson, Ryan Chard, Yadu Babuji, Ganesh Sivaraman, Sutanay Choudhury, Kyle Chard, Rajeev Thakur, and Ian Foster. Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources, 2023. URL https://arxiv.org/abs/2303.08803.

[252] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[253] WebRTC, n.d. https://www.w3.org/TR/webrtc/. Accessed Sep 2024.

[254] Jinfeng Wen and Yi Liu. A Measurement Study on Serverless Workflow Services. In *2021 IEEE International Conference on Web Services (ICWS)*, pages 741–750, 2021. doi:10.1109/ICWS53863.2021.00102.

[255] Michael Wilde, Ian Foster, Kamil Iskra, Pete Beckman, Zhao Zhang, Allan Espinosa, Mihael Hategan, Ben Clifford, and Ioan Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42(11):50–60, 2009.

[256] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37 (9):633–652, 2011. ISSN 0167-8191. doi:https://doi.org/10.1016/j.parco.2011.05.005.

[257] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(2):115–152, 1995.

[258] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 95–102, 2013. doi:10.1109/CCGrid.2013.99.

[259] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[260] Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, and Chi Wang. MathChat: Converse to Tackle Challenging Math Problems with LLM Agents, 2024. URL https://arxiv.org/abs/2306.01337.

[261] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv:1708.07747*, 2017.

[262] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms, 2017. https://doi.org/10.48550/arXiv.1708.07747.

[263] Xiaoli Yan, Nathaniel Hudson, Hyun Park, Daniel Grzenda, J. Gregory Pauloski, Marcus Schwarting, Haochen Pan, Hassan Harb, Samuel Foreman, Chris Knight, Tom Gibbs, Kyle Chard, Santanu Chaudhuri, Emad Tajkhorshid, Ian Foster, Mohamad Moosavi, Logan Ward, and E. A. Huerta. MOFA: Discovering Materials for Carbon Capture with a GenAI- and Simulation-Based Workflow, 2025. URL https://arxiv.org/abs/2501.10651.

[264] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi:10.1145/3419111.3421280. URL https://doi.org/10.1145/3419111.3421280.

[265] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[266] Sicheng Zhou, Zhuozhao Li, Valérie Hayot-Sasson, Haochen Pan, Maxime Gonthier, J. Gregory Pauloski, Ryan Chard, Kyle Chard, and Ian Foster. WRATH: Workload Resilience Across Task Hierarchies in Task-based Parallel Programming Frameworks. In *25th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2025. doi:10.48550/arXiv.2503.12752.

[267] Maxim Zvyagin, Alexander Brace, Kyle Hippe, Yuntian Deng, Bin Zhang, Cindy Orozco Bohorquez, Austin Clyde, Bharat Kale, Danilo Perez-Rivera, Heng Ma, et al. GenSLMs: Genome-scale language models reveal SARS-CoV-2 evolutionary dynamics. *The International Journal of High Performance Computing Applications*, 37 (6):683–705, 2023.