Empowering Scientific Workflows with Federated Agents

J. Gregory Pauloski
University of Chicago

Yadu Babuji
University of Chicago

Ryan Chard
Argonne National Laboratory

Mansi Sakarvadia
University of Chicago

Kyle Chard
University of Chicago
Argonne National Laboratory

Ian Foster
Argonne National Laboratory
University of Chicago

Abstract

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. Here we introduce Academy, 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, and dynamic resource availability. It provides abstractions for expressing stateful agents, managing inter-agent coordination, and integrating computation with experimental control. We 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, we also present case studies in materials discovery, decentralized learning, and information extraction in which agents are deployed across diverse HPC systems.

Keywords

Computational Workflows, Distributed Computing, Federated Computing, Multi-Agent Systems, Open-Source Software

1 Introduction

The desire to automate scientific processes has led to advancements in many fields, from artificial intelligence (AI) [72] and computational workflows [22] to research data management [4] and self-driving laboratories (SDL) [1], 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 [50].

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 [65] or encapsulate research cyberinfrastructure (e.g., computational resources and procedures, experimental instruments, data repositories) [30].

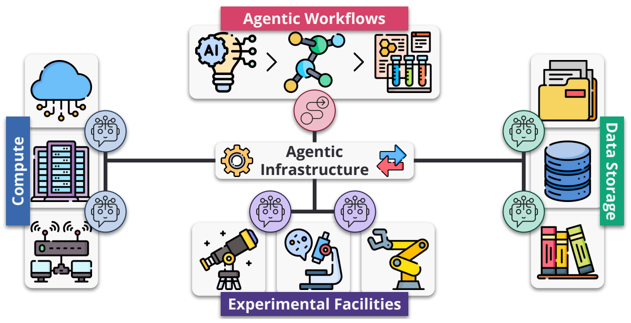

Figure 1: Cooperative agents, spanning federated research infrastructure (experimental facilities, computational systems, data storage), can enable agentic workflows that autonomously steer discovery.

Significant progress has been made towards developing AI agents that can act on behalf of humans for such tasks as literature synthesis [43], hypothesis generation [35], and data analysis and publication [39]. However, existing agent frameworks (e.g., AutoGen [69]) 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 1.

Frameworks for building agentic workflows are limited in scope and generally target conversational, cloud-native applications (e.g., LLM-based AI chatbots) [44, 54, 69]. 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:

- 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 3).
- Performance analysis of Academy in diverse scenarios yielding insights into the scalability and practical considerations of deploying agentic workflows (Section 4).
- Case studies demonstrating the utility of agentic workflow design and highlighting improvements in automation, resource utilization, and discovery acceleration (Section 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.

2 Background

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 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 [37]. 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 [33]. 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. *Intelligent (deliberative) agents* are goal-oriented and reason about what actions to take using internal models and external perception [68]. *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 [53]. *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 [51]. *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 [55], which can lead to powerful emergent behavior [21].

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* $d(A, R) : A \rightarrow R$ 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.

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.

3.1 Academy 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 [37].

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.

3.2 Academy 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 2, 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

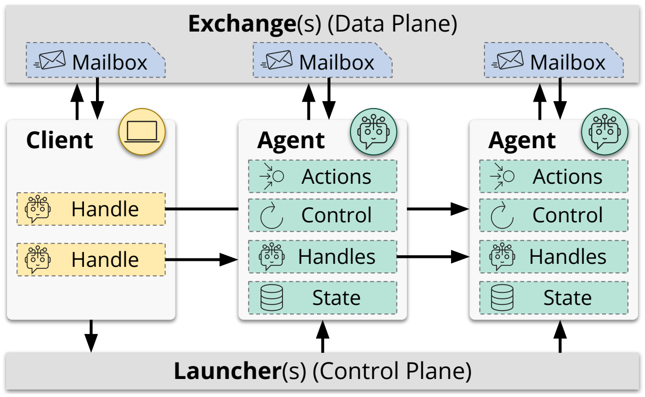

Figure 2: Agents and clients in Academy interact via handles to invoke actions asynchronously. Agents implement a behavior, defined by their actions, control loops, and state. Academy decouples the control and data planes through the launcher and exchange components that manage spawning agents and communication, respectively.

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

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

    @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)
```

Listing 1: Example agent behavior definition.

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 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.

3.3 Academy Implementation Details

Academy is implemented as an open-source Python library, available on GitHub and PyPI.[1] 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.

[1]https://github.com/proxystore/academy

3.3.1 Behavior. An agent behavior is implemented as a Python class that inherits from the base `Behavior` type, as shown in Listing 1. 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 Section 3.3.4.) 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.

3.3.2 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 [60], Parsl [9], or Ray [52]. We discuss these patterns in more detail in Section 3.4.4. 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 provide experimental support for 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).

3.3.3 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).

3.3.4 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 [5] and another behavior `OpenProteinFolder` that inherits from `ProteinFolder` and specifically uses the OpenFold model [2]. 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 *thread 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 [57, 58], 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 [61] and UCX [62], GridFTP via Globus Transfer [17], and reliable peer-to-peer UDP (user datagram protocol) through NAT hole-punching. ProxyStore 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.

3.3.5 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 [9]. Parsl supports execution across local, remote, and batch compute systems.
- **Globus Compute:** Runs agents across Globus Compute Endpoints [18]. 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.

```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)
```

Listing 2: Example of initialization, spawning, using, and shutting down an agent using the `Manager` interface.

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 Listing 2.

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.

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 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).

3.4.5 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 [19], enabling applications to utilize longer-lived, stateful, and isolated processes on-demand.

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 [42]. Polaris has 560 nodes interconnected by an HPE Slingshot 11 network. 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.

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.

4.1.1 Agent Startup Time. We measure the time to spawn n agents in Figure 3 (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 Academy 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.

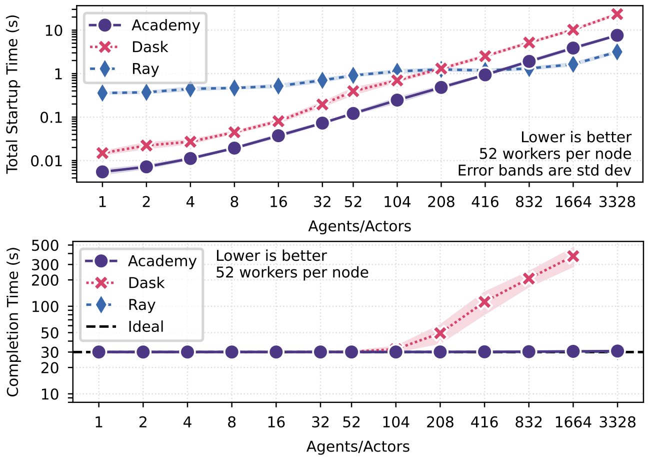

Figure 3: (Top) Warm-start time for *n* agents/actors between Academy (using the Parsl launcher), Dask Actors, and Ray Actors. Ray does not benefit from warm-starts because a new process is spawned for each actor. (Bottom) Time to execute 30 actions per agent/actor (weak scaling). Each action sleeps for 1 s. Note the Academy and Ray lines are overlapped.

4.1.2 Action Completion Time. In Figure 3 (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.

4.2 Academy Distributed Exchange

Next, we study the performance of the distributed exchange.

4.2.1 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 and ProxyStore's P2P endpoints 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.

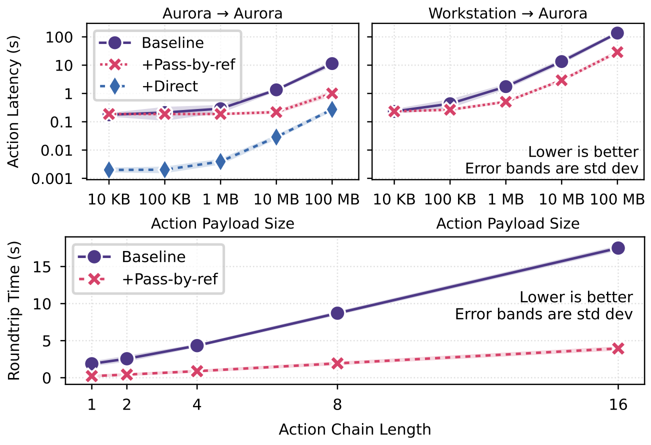

Figure 4: (Top) Time for a client to invoke a no-op action on an actor as a function of input and output payload size with different optimizations enabled on the distributed exchange. Two scenarios are considered: client and agent are at the same site (left) and different sites (right). (Bottom) Time 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.

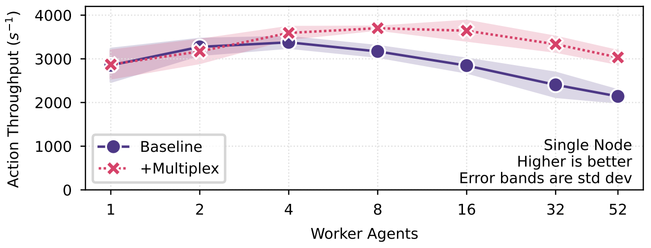

Figure 5: Maximum no-op action throughput for a single agent requesting actions from *n* worker agents. The handle multiplexing optimization improves performance by reducing the number of mailbox listener threads from *n* to 1.

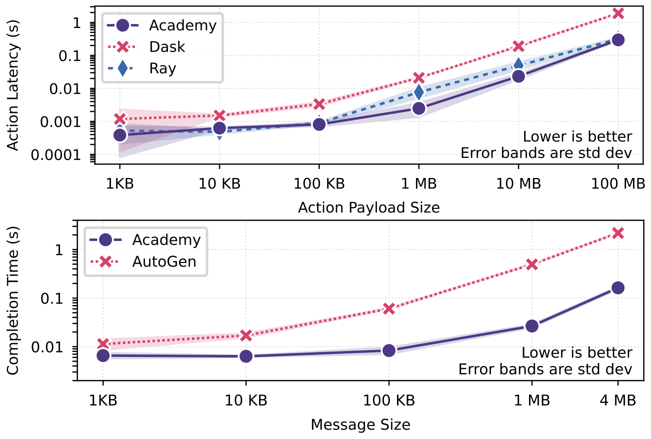

Figure 6: (Top) No-op action latency between two agents/actors running on separate Aurora nodes versus action input and output payload size. (Bottom) Completion time for a simulated two agent chat where agents send ten messages back and forth with varied message sizes. ACADEMY is compared to AutoGen's distributed runtime.

In Figure 4 (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-by-ref*, and *direct* across two scenarios: *Aurora* ⟶ *Aurora*, where the client and the agent are located on two different Aurora nodes, and *Workstation* ⟶ *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* reduce action latency compared to the baseline by 91.2% and 97.6%, respectively, with 100 MB payloads. For inter-site transfers, *pass-by-ref* reduces 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 4 (bottom). Data are only communicated once to the agent that uses the data (here, the last agent in the chain).

4.2.2 *Handle Multiplexing.* As described in Section 3.3.3, 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 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 due to reduced threading overheads.

4.3 Agent Messaging

Here, we investigate the performance of agent messaging. As in Section 4.1, the object store of the exchange is located on the head node of the Aurora batch job.

4.3.1 *Action Latency.* In Figure 6 (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

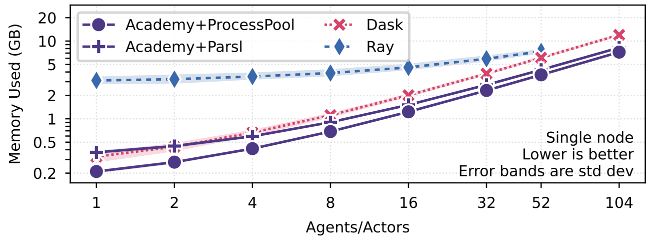

Figure 7: Memory used by n agents/actors. We encountered Ray crashes when deploying 104 actors on a single Aurora node (i.e., all cores on both sockets).

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.

4.3.2 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 3.4.4). The pool contains 208 agents across four nodes to ensure 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 3.4K, 185, and 14.1K action/s, respectively. Academy is 18× faster than Dask but 4× slower than Ray; however, this is a worst case scenario with no-op tasks and we believe >3K actions/s to be sufficient for real-world agents.

4.3.3 Agent Conversations. In Figure 6 (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.

4.4 Memory Overhead

We show memory used as a function of number of agents in Figure 7; 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 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.

5.1 Materials Discovery

MOFA [71] 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 [29]. 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: (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 [66] to coordinate the flow of data between stages and to distribute 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 8. 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.

An execution trace of the agentic MOFA workflow (Figure 9) 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

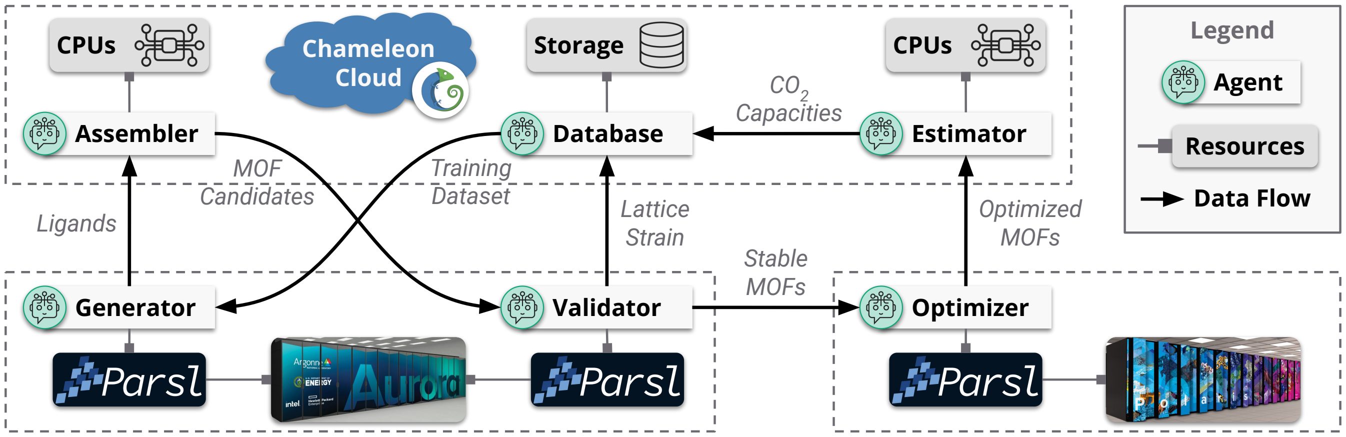

Figure 8: We execute MOFA by deploying agents across federated infrastructure with Globus Compute. The Assembler, Database, and Estimator run on Chameleon Cloud nodes with fast single-core performance; the Generator and Validator run on Aurora login nodes and execute AI and simulation tasks, respectively, on Aurora compute nodes; and the Optimizer runs on a Polaris login node and executes simulation tasks on Polaris 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 use Parl to scale resources up and down based on workload needs.

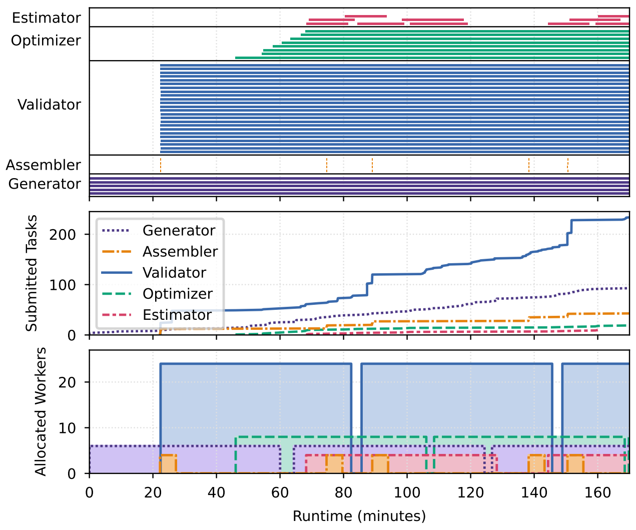

Figure 9: Execution trace of the agentic MOFA workflow of Figure 8 over three hours. (Top) Active tasks per agent. The vertical axis height represents the maximum size of the resource pool allocated by each agent (i.e., CPUs or GPUs). Assembler tasks are short and infrequent. (Middle) Cumulative tasks 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).

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 [1, 64], and it is tangible to envision a future where these loosely coupled agentic workflows incorporate services provided by self-driving labs through embodied agents.

5.2 Decentralized Learning

In decentralized machine learning a set of models learn collaboratively across distributed datasets [36]. 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.

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 [38]. 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 [24]. 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 10. We do not show

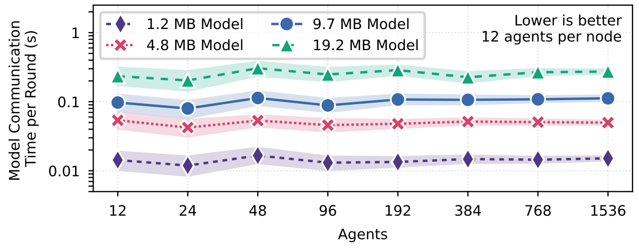

Figure 10: Model communication time to an agent's neighbors averaged over five rounds of decentralized training. Training time and aggregation time are excluded since they are nearly constant.

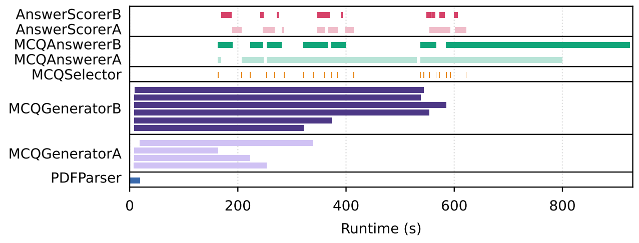

Figure 11: Execution trace of the agentic MCQ workflow processing 10 manuscripts to generate and validate questions and answers over 15 minutes. The figure shows the active agents and the duration of their tasks. Agents employ either the Mistral-7B-Instruct-v0.3 or Meta-Llama-3-70B-Instruct model, denoted A and B, respectively.

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 10).

5.3 Information Extraction

Exponential growth in scientific publications [12] 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 [14, 63].

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 [15, 16]. The workflow 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 [41] and Meta-Llama-3-70B-Instruct [8] 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 11 an execution trace from a run in which the agents just listed were run concurrently to generate and validate MCQs for 10 publications.

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 [60] and Ray [52], 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 [7], Fireworks [40], Makeflow [3], Nextflow [25], Parsl [9], Pegasus [23], Swift [67]). WMSs can be differentiated by how dependency graphs are defined [56]: static configurations files, such as CWL [20], XML, or YAML; general purpose languages (GPLs); domain specific languages (DSLs); or procedurally 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 [37]. In response to a message, an actor can alter its local state, send messages to other actors, and create new actors. 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 [47], Dask, Orleans [10], Ray) and function-as-a-service (FaaS) platforms (e.g., Abaco [31], Azure Service Fabric [49], PraaS [19]). Actor models have been investigated as alternatives for designing computational workflows where communication and coordination are decoupled [13]. 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 [26, 27]. Recent work focuses on improving the reasoning capabilities of LLM-backed agents through ontological knowledge graphs and multi-agent systems [32] and tool-augmented LLMs [48]. Increasingly popular is the use of multi-agent conversations, in which multiple role-specialized agents interact to collaborate, coordinate, or compete towards goals [69]. These systems enhance LLM-based tools through better reasoning [28], validation [70], and divergent thinking [46], prompting rapid development of frameworks such as LangGraph [44], Microsoft AutoGen [69], OpenAI Swarm [54], and Pydantic Agents [59]. Subsequently, interest in standardizing agent protocols has developed. Anthropic's Model Context Protocol (MCP) [6] defines structured interaction between humans/tools and AI models. Google's Agent2Agent (A2A) Protocol [34] focuses on structured interaction between autonomous agents; each agent serves an HTTP endpoint which is impracticable for many scientific workflows. Multi-agent conversations can proxy scientists in iterative scientific processes—brainstorming ideas, planning experiments, and reasoning about results [11, 30, 35, 65]—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.

7 Conclusion & Future Work

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 research 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 [45], 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.

8 Acknowledgments

This research was supported in part by the National Science Foundation under Grants 2004894 and 2209919. We used resources provided by the Argonne Leadership Computing Facility (ALCF), a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory supported under Contract DE-AC02-06CH11357, and the Chameleon testbed supported by the National Science Foundation.

References

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

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

[3] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. 2012. 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* (Scottsdale, Arizona, USA) *(SWEET '12)*. Association for Computing Machinery, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/2443416.2443417

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

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

[6] Anthropic. 2024. Model Context Protocol (MCP). Retrieved Apr 2025 from https://modelcontextprotocol.io/

[7] Apache. 2015. Airflow. Retrieved Sep 2024 from https://airflow.apache.org/

[8] The Llama Team at Meta. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[9] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3307681.3325400

[10] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Technical Report MSR-TR-2014-41. Microsoft.

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

[12] Lutz Bornmann, Robin Haunschild, and Rüdiger Mutz. 2021. 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 (2021), 1–15.

[13] Shawn Bowers and Bertram Ludäscher. 2005. Actor-Oriented Design of Scientific Workflows. In *Conceptual Modeling – ER 2005*, Lois Delcambre, Christian Kop, Heinrich C. Mayr, John Mylopoulos, and Oscar Pastor (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–384.

[14] 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. 2019. 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 (2019), 65263–65276.

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

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

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

[18] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. funcX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC*

'20). Association for Computing Machinery, New York, NY, USA, 65–76. https://doi.org/10.1145/3369583.3392683

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

[20] 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. 2022. Methods included: standardizing computational reuse and portability with the Common Workflow Language. *Commun. ACM* 65, 6 (May 2022), 54–63. https://doi.org/10.1145/3486897

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

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

[23] 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. 2015. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems* 46 (2015), 17–35. https://doi.org/10.1016/j.future.2014.10.008

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

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

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

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

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

[29] Hiroyasu Furukawa, Kyle E Cordova, Michael O'Keeffe, and Omar M Yaghi. 2013. The chemistry and applications of metal-organic frameworks. *Science* 341, 6149 (2013), 1230444.

[30] Shanghua Gao, Ada Fang, Yepeng Huang, Valentina Giunchiglia, Ayush Noori, Jonathan Richard Schwarz, Yasha Ektefaie, Jovana Kondic, and Marinka Zitnik. 2024. Empowering biomedical discovery with AI agents. *Cell* 187, 22 (2024), 6125–6151. https://doi.org/10.1016/j.cell.2024.09.022

[31] Christian Garcia, Joe Stubbs, Julia Looney, Anagha Jamthe, and Mike Packard. 2020. Abaco–A Modern Platform for High Throughput Parallel Scientific Computations.

[32] Alireza Ghafarollahi and Markus J Buehler. 2024. SciAgents: Automating Scientific Discovery Through Bioinspired Multi-Agent Intelligent Graph Reasoning. *Advanced Materials* (2024), 2413523.

[33] Richard Goodwin. 1995. Formalizing properties of agents. *Journal of Logic and Computation* 5, 6 (1995), 763–781.

[34] Google. 2025. Agent2Agent Protocol (A2A). Retrieved Apr 2025 from https://github.com/google/A2A

[35] Google Research. 2025. Accelerating scientific breakthroughs with an AI co-scientist. Retrieved Feb 2025 from https://research.google/blog/accelerating-scientific-breakthroughs-with-an-ai-co-scientist/

[36] István Hegedűs, Gábor Danner, and Márk Jelasity. 2019. 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*. Springer, 74–90.

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

[38] Petter Holme and Beom Jun Kim. 2002. Growing scale-free networks with tunable clustering. *Physical review E* 65, 2 (2002), 026107.

[39] Tal Ifargan, Lukas Hafner, Maor Kern, Ori Alcalay, and Roy Kishony. 2025. Autonomous LLM-Driven Research—from Data to Human-Verifiable Research Papers. *NEJM AI* 2, 1 (2025), AIoa2400555.

[40] 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. 2015. FireWorks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 5037–5059.

[41] 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. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[42] 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. 2020. Lessons Learned from the Chameleon Testbed. In *USENIX Annual Technical Conference*. USENIX Association.

[43] Jakub Lála, Odhran O'Donoghue, Aleksandar Shtedritski, Sam Cox, Samuel G Rodriques, and Andrew D White. 2023. PaperQA: Retrieval-Augmented Generative Agent for Scientific Research.

[44] LangChain. 2024. LangGraph. Retrieved Jan 2025 from https://www.langchain.com/langgraph

[45] Thomas Leblanc and John Mellor-Crummey. 1987. Debugging parallel programs with instant replay. *IEEE Trans. Comput.* 100, 4 (1987), 471–482.

[46] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2024. Encouraging Divergent Thinking in Large Language Models through Multi-Agent Debate. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 17889–17904. https://doi.org/10.18653/v1/2024.emnlp-main.992

[47] Lightbend. 2009. Akka: the Actor Model on the JVM. Retrieved Sep 2024 from https://akka.io/

[48] Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, Aixin Sun, Hany Awadalla, et al. 2024. SciAgent: Tool-augmented language models for scientific reasoning. *arXiv preprint arXiv:2402.11451* (2024).

[49] Microsoft. 2017. Azure: Service Fabric Reliable Actors. Retrieved Sep 2024 from https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-actors-introduction

[50] William L. Miller, Deborah Bard, Amber Boehnlein, Kjiersten Fagnan, Chin Guok, Eric Lançon, Sreeranjani Ramprakash, Mallikarjun Shankar, Nicholas Schwarz, and Benjamin L. Brown. 2023. *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). https://doi.org/10.2172/1984466

[51] 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. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529–533.

[52] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 561–577.

[53] Hyacinth S. Nwana. 1996. Software agents: An overview. *The Knowledge Engineering Review* 11, 3 (1996), 205–244. https://doi.org/10.1017/S026988890000789X

[54] OpenAI. 2024. Swarm. Retrieved Jan 2025 from https://github.com/openai/swarm

[55] Liviu Panait and Sean Luke. 2005. Cooperative multi-agent learning: The state of the art. *Autonomous agents and multi-agent systems* 11 (2005), 387–434.

[56] J. Gregory Pauloski, Valerie Hayot-Sasson, Maxime Gonthier, Nathaniel Hudson, Haochen Pan, Sicheng Zhou, Ian Foster, and Kyle Chard. 2024. TaPS: A Performance Evaluation Suite for Task-based Execution Frameworks. In *IEEE 20th International Conference on e-Science*. IEEE, New York, NY, USA, 1–10. https://doi.org/10.1109/e-Science62913.2024.10678702

[57] J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Alexander Brace, André Bauer, Kyle Chard, and Ian Foster. 2025. Object Proxy Patterns for Accelerating Distributed Applications. *IEEE Transactions on Parallel and Distributed Systems* 36, 2 (2025), 253–265. https://doi.org/10.1109/TPDS.2024.3511347

[58] J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Nathaniel Hudson, Charlie Sabino, Matt Baughman, Kyle Chard, and Ian Foster. 2023. Accelerating Communications in Federated Applications with Transparent Object Proxies. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. ACM, New York, NY, USA, Article 59, 15 pages. https://doi.org/10.1145/3581784.3607047

[59] Pydantic. 2024. Agents. Retrieved Jan 2025 from https://ai.pydantic.dev/agents/

[60] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). SciPy, Austin, TX, USA, 126 – 132. https://doi.org/10.25080/Majora-7b98e3ed-013

[61] 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. 2020. Mochi: Composing Data

Services for High-Performance Computing Environments. *Journal of Computer Science and Technology* 35, 1 (Jan 2020), 121 – 144,. 10.1007/s11390-020-9802-0.

[62] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. 2015. UCX: An open source framework for HPC network APIs and beyond. In *IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.

[63] Jamshid Sourati and James A Evans. 2023. Accelerating science with human-aware artificial intelligence. *Nature Human Behaviour* 7, 10 (2023), 1682–1696.

[64] Rafael Vescovi, Tobias Ginsburg, Kyle Hippe, Doga Ozgulbas, Casey Stone, Abraham Stroka, Rory Butler, Ben Blaiszik, Tom Brettin, Kyle Chard, et al. 2023. Towards a modular architecture for science factories. *Digital Discovery* 2, 6 (2023), 1980–1998.

[65] 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. 2023. Scientific discovery in the age of artificial intelligence. *Nature* 620 (2023), 47–60. https://api.semanticscholar.org/CorpusID:260384616

[66] 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. 2021. 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*. IEEE, 9–20. https://doi.org/10.1109/MLHPC54614.2021.00007

[67] Michael Wilde, Ian Foster, Kamil Iskra, Pete Beckman, Zhao Zhang, Allan Espinosa, Mihael Hategan, Ben Clifford, and Ioan Raicu. 2009. Parallel scripting for applications at the petascale and beyond. *Computer* 42, 11 (2009), 50–60.

[68] Michael Wooldridge and Nicholas R. Jennings. 1995. Intelligent agents: Theory and practice. *The Knowledge Engineering Review* 10 (1995), 115 – 152. https://api.semanticscholar.org/CorpusID:221342993

[69] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs.AI] https://arxiv.org/abs/2308.08155

[70] Yiran Wu, Feiran Jia, Shaokun Zhang, Hangyu Li, Erkang Zhu, Yue Wang, Yin Tat Lee, Richard Peng, Qingyun Wu, and Chi Wang. 2024. MathChat: Converse to Tackle Challenging Math Problems with LLM Agents. arXiv:2306.01337 [cs.CL] https://arxiv.org/abs/2306.01337

[71] 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. 2025. MOFA: Discovering Materials for Carbon Capture with a GenAI- and Simulation-Based Workflow. arXiv:2501.10651 [cs.DC] https://arxiv.org/abs/2501.10651

[72] Maxim Zvyagin, Alexander Brace, Kyle Hippe, Yuntian Deng, Bin Zhang, Cindy Orozco Bohorquez, Austin Clyde, Bharat Kale, Danilo Perez-Rivera, Heng Ma, et al. 2023. GenSLMs: Genome-scale language models reveal SARS-CoV-2 evolutionary dynamics. *The International Journal of High Performance Computing Applications* 37, 6 (2023), 683–705.