Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources

Logan Ward*, J. Gregory Pauloski†, Valerie Hayot-Sasson†, Ryan Chard*, Yadu Babuji†, Ganesh Sivaraman*,
Sutanay Choudhury‡, Kyle Chard†, Rajeev Thakur*, and Ian Foster*
*Argonne National Laboratory, Lemont, IL 60439; †University of Chicago, Chicago, IL 60637;
‡Pacific Northwest National Laboratory, Richland, WA 99354

Abstract—**Applications that fuse machine learning and simulation can benefit from the use of multiple computing resources, with, for example, simulation codes running on highly parallel supercomputers and AI training and inference tasks on specialized accelerators. Here, we present our experiences deploying two AI-guided simulation workflows across such heterogeneous systems. A unique aspect of our approach is our use of cloud-hosted management services to manage challenging aspects of cross-resource authentication and authorization, function-as-a-service (FaaS) function invocation, and data transfer. We show that these methods can achieve performance parity with systems that rely on direct connection between resources. We achieve parity by integrating the FaaS system and data transfer capabilities with a system that passes data by reference among managers and workers, and a user-configurable steering algorithm to hide data transfer latencies. We anticipate that this ease of use can enable routine use of heterogeneous resources in computational science.**

Index Terms—**Heterogeneous Computing, Function-as-a-Service, Machine Learning, Distributed Systems, Computational Steering**

I. INTRODUCTION

Scientific applications that employ AI models often require or can benefit from the use of multiple, heterogeneous computing resources. For example, an application that uses AI-based perception to process data from a scientific instrument may engage a remote, more powerful computer for expensive tasks [1], while an AI-guided simulation application may run simulation codes on a conventional high performance computing (HPC) system but perform AI model training and inference on specialized accelerators [2]. While such multi-resource (or even multi-site) applications are far from new [3, 4], they are arguably becoming more commonplace due to more specialized computer architectures, especially for AI applications, increasingly portable code, and faster and more reliable networks.

The challenges around deploying a single application on multiple resources are well known. Modern workflow systems typically place services on each resource (e.g., pilot jobs) that connect back to a controller through an open port on a central server or via a secure tunnel. However, this approach requires either maintaining a server or establishing secure tunnels to each resource, introducing deployment complexity and single points of failure. Hybrid software-as-a-service approaches,

as used in Globus [5] and FuncX [6], reduce deployment complexity and increase reliability by using a cloud-hosted coordination service to manage authentication and respond appropriately to errors.

A second challenge in multi-resource deployments is to mitigate the costs of data transfer—a critical need in data-heavy AI applications. Most multi-resource workflows communicate data between resources via the workflow controller or a shared data store like a database or shared file system. Complexity arises, however, when data sizes increase and overwhelm the controller or resources differ in their access to shared data transfer mediums. Passing data by reference and employing peer-to-peer data transfers can reduce strain on the workflow controller (e.g., Brace et al. [2]). Separation allows data transfer via more efficient mechanisms (e.g., Parsl [7] uses Globus). Subsystems that enable robust and flexible data movement by reference are key to the efficacy of multi-resource workflows.

We describe here our successes in deploying two multi-resource applications (molecular design and surrogate training) with approaches that maximize performance and minimize deployment challenges. Both applications require using CPU resources to run simulation tasks and GPU resources to achieve timely solutions for AI tasks but vary in the frequency, duration, and data transfer requirements for each type of task. Our implementation uses cloud-managed services where possible to ensure reliability and simplify deployment of both secure function execution and data transfer between resources. Specifically, we chose a federated FaaS platform (FuncX) and a system (ProxyStore) that enables peer-to-peer data transfer via Globus with minimal changes to application code. We first characterize the performance of these tools using a synthetic application and then describe how we achieved performance parity between our cloud-managed solution and one using a conventional workflow system, Parsl.

II. RELATED WORK

Many capabilities are needed to implement multi-resource computational campaigns. Specifically, workflow systems must 1) execute tasks on resources acquired from multiple facilities; 2) exchange large task data between resources; 3)

express policies that avoid latencies inherent to orchestrating multiple systems. Here, we review the technologies available for each need.

A. Coordinating Remote Execution

Remote computing has often been performed via SSH connections. Grid computing introduced remote connectivity and web services to manage remote execution, for example via Globus Grid Resource Allocation Manager (GRAM) [8]. Recently, facilities such as NERSC and TACC have developed web APIs for submitting and managing batch jobs with Newt [9] and Tapis [10], respectively. These approaches add each new task to a global queue, which can result in significant delays due to the need to wait for batch jobs to be scheduled in applications where new tasks are created dynamically.

Many workflow systems have been developed to execute sets of tasks on both local and remote computing resources. Systems such as Pegasus [11], Dask [12], Parsl [7], Swift [13], and Galaxy [14] differ in various ways, but all provide: 1) a method for describing a workflow; 2) a data model for representing dependencies among tasks (e.g., a DAG); and 3) a runtime system for executing tasks on local and/or remote resources. When dispatching tasks to HPC, such systems often employ multi-level scheduling schemes that map individual tasks to workers deployed on HPC resources, an approach that allows dynamically generated tasks timely access to computing resources [15].

The serverless or FaaS paradigm popularized by hosted cloud platforms such as Amazon Lambda [16], Google Cloud Functions [17], and Azure Functions [18] is an attractive model for remote scientific computing as it allows large applications to be broken down into smaller, efficient, and adaptable functions [19–23]. However, the commercial FaaS offerings are proprietary systems that are restricted to a single cloud provider and do not support execution of functions on existing cyberinfrastructure.

Open source frameworks (e.g., Apache OpenWhisk [24], Fn [25], KNIX MicroFunctions [26], Abaco [27]) can be used for on-premise deployments. However, most use Docker and rely on Kubernetes to operate—an assumption that prohibits use on HPC systems that employ batch schedulers. Some systems, such as ChainFaaS [28] and DFaaS [29], support distributed function execution on personal computers and edge nodes. FuncX [6] is the only system that supports remote execution on a federated ecosystem of endpoints in a diverse research cyberinfrastructure spanning from HPC to edge.

B. Inter-Resource Data Fabrics

Workflow systems that couple diverse applications in distributed environments require a data fabric which provides consistent access to data regardless of location. In the tuple space model, originating in Linda [30], data producers and consumers use put and get operations on a shared distributed-memory space. Dataspaces [31] supports large-scale, dynamic scientific applications via a tuple-space-like model implemented using the Margo and Mercury RPC libraries [32, 33].

WA-Dataspaces [34] provides predictive prefetching and data staging support in wide area networks.

Multi-site workflows require a secure data fabric accessible by hosts behind different firewalls, features not supported by Linda, Dataspaces, or WA-Dataspaces. SSH tunnels can enable secure communication but can be cumbersome to establish and fragile to maintain. Science DMZs [35] provide secure sub-networks that span sites without firewalls but are only available at select computing sites. SciStream [36] uses gateway nodes provided by computing sites for fast memory-to-memory data streaming for high throughput science applications. Cloud services provide higher availability than the aforementioned but add additional network hops (which adds latency), fail to take advantage of high-performance connections between sites, and can be cost prohibitive.

C. AI-Integrated Workflows

AI-integrated workflows are emerging as a powerful tool across many scientific domains, with many use patterns [37, 38]. Following the lexicon of Jha et al. [37], there are at least "ML-in-HPC" applications where AI-based software are used inside conventional HPC applications (e.g., surrogates for expensive computational routines [39]) and "ML-out-HPC" where machine learning controls the execution of the application (e.g., steering a workflow via active learning [40–42]). The large diversity of application types is reflected in the many tools that support integration of AI in existing scientific codes or the creation of entirely new classes of applications. Most relevant to our work are variants that use AI to determine the next task to execute in a workflow.

Systems for steering AI-guided workflows vary in how they express coordination between the workflow and the intelligence system. Systems such as Supervisor [43] and DeepHyper [44] use a model where a single process running a steering algorithm receives results and submits new tasks to the workflow via a queue. In libEnsemble [45], a workflow is expressed as "simulation" tasks that produce data, "generator" tasks that produce new tasks, and "allocator" tasks that determine when to launch each type of task. Ray [46] and Decaf [47] allow a decentralized model where the steering logic is expressed in an agent-based programming model.

III. MOTIVATING APPLICATIONS

We study two applications that require CPU resources for simulation tasks, and GPU resources for AI tasks. Each have different coordination patterns and data transfer requirements, as shown in Fig. 1. The first application, Molecular Design, consistently runs tasks on a GPU and transfers $O(10)$ GB per batch of AI tasks. The second application, Surrogate Fine-tuning, runs AI tasks more sporadically and has an order of magnitude smaller data transfer requirements. The quality of results in both applications, however different, requires completing AI tasks rapidly after sufficient simulation tasks have completed.
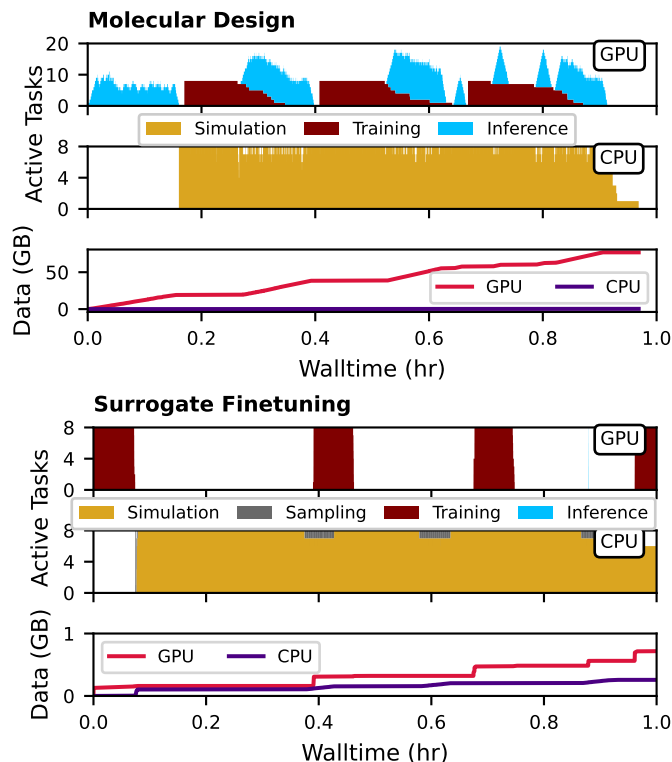
Fig. 1. Resource utilization for our example applications. We show the number of tasks running and the cumulative data transfered to each resource over time. Data were collected using 20 T4 GPUs and 8 KNL processors, with a workflow based on Parsl without pass-by-reference.

Full implementations are available on GitHub (https://github.com/exalearn/multi-site-campaigns) and the Materials Data Facility [48].

A. Application: Molecular Design

This application seeks to identify molecules within a candidate set (here, 1 115 321 molecules from the MOSES dataset [49], as collected in the nCov-Group Data Repository [50]) that have desirable properties. Properties of any individual molecule can be determined via an expensive quantum chemistry simulation, so we use active learning [51] to select simulations to perform. Results from previous simulations are used to train a machine-learned surrogate model, which is then used to infer scores for the remaining molecules; those scores are used to determine which simulations to be performed. This process repeats until our computational budget is exhausted. The application thus performs *simulation*, *training*, and *inference* tasks.

Simulation The simulation task computes a molecule's ionization potential (IP), a key design property for organic electrolytes [52], using tight binding, an inexpensive quantum-mechanical simulation method. First, RDKit [53] is used to generate an initial 3D geometry for a molecule from its bonding connectivity. Then, geoMETRIC [54] determines the equilibrium geometry/energy for the neutral and charged states using the energies and forces computed with the QCEngine

interface [55] of eXtended Tight Binding (xTB) [56]. All computations take ∼60 s on a CPU and produce 1 MB data.

Training The surrogate model used to predict a molecule's IP given its bonding connectivity comprises an ensemble of eight message-passing neural networks [57] (MPNNs). Each model has the same architecture but is trained on a different, randomly-selected subset of the training data. Training each model requires 340 s on a GPU and generates 10 MB, and models can be trained in parallel.

Inference The inference tasks use the ensemble of MPNN models from the training task to estimate IPs. After predicting all molecule IPs with each model, the molecules are ranked by the Upper Confidence Bound (UCB) of the predictions, which is the sum of the mean and standard deviations of the model predictions. Scoring the full dataset per model takes 390 s on a GPU (3000 inferences/s) and requires transfer of 2.3 GB (model weights, molecule inputs, outputs).

Success is measured by how many molecules we find with high IPs after a certain amount of compute has been expended.

B. Application: Surrogate Fine-tuning

This application trains a surrogate model for expensive quantum mechanical computations. Our goal is to produce a model capable of replicating the energies and forces from Density Functional Theory (DFT) calculations on clusters of water surrounding a methane solute. We first train a model on the energies of many water clusters computed using an approximate method (Thole-Type Models [58], HydroNet [59] Dataset) and then refine the model with a small number of energies and forces of solvated methane calculated using DFT. We use an active learning approach similar to the molecular design application to guide the choice of training data: we run the DFT calculations on the structures where the surrogate model is least certain. Beyond the inference, training, and simulation tasks required by active learning, we also use a *sampling* task to produce new structures.

Training We train surrogate models using the SchNet architecture [60] as implemented in PyTorch-Geometric. A training task requires an average of 4 minutes on a GPU and transmitting 21 MB. We train an ensemble of 8 SchNet models where each is trained on a different, randomly-selected subset of the training data.

Inference An inference tasks on a batch of 500 structures takes an average of 1.2 s on a GPU and involves transmitting 10 MB (including all inputs and outputs).

Simulation Psi4 [61] via the Atomic Simulation Environment (ASE) interface [62] is used to compute the energy and forces acting on a cluster of atoms using the PBE0 exchange-correlation function and the aug-cc-pvdz basis set. Each task takes a mean of 360 s on CPU and produces 30 kB.

Sampling Sampling tasks produce new structures using a trained SchNet model. We create new structures by initializing the temperature of a structure of water-solvated methane to 100K, then running molecular dynamics for a set number of timesteps. Choosing the number of timesteps involves a tradeoff: too few produces too little diversity in structures, too
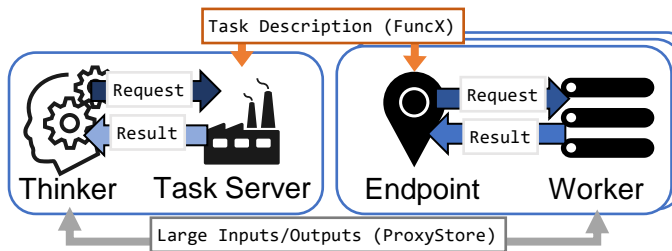
Fig. 2. Our implementation of a multi-site workflow. A **Thinker** expressed using Colmena controls when tasks are executed by communicating with a **Task Server** that manages submitting task via FuncX. FuncX routes task requests through a cloud provider and uses endpoints deployed on multiple resources that communicate with workers deployed on compute nodes. Large data is passed directly between the Thinker and workers using ProxyStore backed by Globus.

many produces unrealistic structures due the accumulation of model errors over many steps. Our first sampling tasks use only 20 timesteps because our model is not yet capable of producing realistic dynamics. We gradually increase to 1000 over the course of the run. Sampling tasks take between 1 and 20 s on a CPU and require transmitting 3 MB.

Our steering agent selects the next simulation to perform from two pools, each designed to provide unique structures to add to the training set. The *audit* pool contains the last structure from each sampling task, which should be the most different from the training set by nature of being farthest along a time-evolution pathway. The *uncertainty* pool contains the structures for which our models have the least-certain predictions across all structures produced during sampling. We re-populate the uncertainty pool each time 500 new structures have been sampled by performing inference on each newly-sampled structure with each model in our ensemble and ranking structures based on the variance in predicted energy.

We start by training our model ensemble on 1720 previously collected structures and continue the active learning until 500 new structures have been added to the set. Our steering algorithm balances the number of workers devoted to simulation and sampling to maintain a constant number of structures in the audit pool. The algorithm begins a new training run after 25 new structures are added to the training set.

Success is measured by the accuracy of a model trained on all DFT calculations available after a run using test sets created from data unseen during training. We produced the test set by running molecular dynamics for 10 structures of solvate methane using DFT to compute energies and forces at starting temperatures of 100K, 300K, and 900K for 32 timesteps.

IV. TECHNICAL APPROACH

We built our AI-guided applications using tools selected to achieve several goals: security, performance and scalability, minimial network configuration, robustness, and portability. These objectives lead us to use FuncX to provide secure and scalable remote execution across resources, ProxyStore to offload data transfer to different data fabrics (e.g., Globus),

and Colmena to express AI-guided workflows. We describe each component below.

A. Cloud-hosted management services

Our approach centers around the use of cloud-hosted services to manage data and compute across distributed resources. This approach enables our applications to communicate with the reliable and accessible cloud-hosted Globus Transfer and FuncX services irrespective of data or compute location.

1) Security: We require a security model that ensures only authenticated and authorized users can execute tasks and access data. A multi-resource or -site workflow requires a solution that supports different identity management systems and authentication models, stringent security requirements (e.g., two-factor authentication, short authorization lifetimes), the need to use different accounts to access different resources, and resource sharing among groups of users.

Globus and FuncX fulfill such security requirements through Globus Auth [63], a general identity and access management platform. Globus Auth is widely adopted in the scientific community, implements standard protocols (e.g., OAuth 2), and enables secure delegation so workflows (e.g., Colmena) can securely leverage other services like FuncX and Globus.

2) Network simplicity: Both Globus and FuncX implement a hybrid deployment model, relying on users or administrators to deploy lightweight software (Globus Connect or FuncX Agents) on remote resources. Communication between cloud and endpoints use inbound HTTPS for Globus and outbound RabbitMQ TCP sockets for FuncX. These endpoints are securely paired with the cloud platforms and subsequent access is permitted only to authenticated and authorized users. Network connections between resources are not required.

3) Robustness: Cloud-hosted management services are highly available and provide high levels of robustness compared to locally-managed services. For example, both FuncX and Globus's services accept and store tasks (and results) even while remote endpoints (or clients) are unavailable resuming tasks when endpoints reconnect to the cloud.

B. FuncX: Federated Function-as-a-Service

FuncX is a federated FaaS platform that combines a hosted cloud service for registering Python functions and dispatching function invocations with an ecosystem of user-managed endpoints deployed on arbitrary computing systems. Users can invoke functions by passing the function body and input arguments to the cloud service via an implementation of Python's `concurrent.futures.Executor` interface. Unlike traditional FaaS systems, a FuncX user also supplies an endpoint ID that determines where the function will be executed. The FuncX endpoint is user-deployed and is responsible for provisioning resources and managing function execution. The endpoint code can interface with different schedulers (e.g., Slurm and PBS) to provision resources. FuncX manages serializing the function body and arguments, sending them to the remote endpoint, and execution on the remote endpoint.

C. ProxyStore

Multi-resource workflows require a resource-spanning data fabric for data exchange. Architecting this system element is challenging because capabilities and performance characteristics vary across resources and a single workflow may have many distinct data patterns (e.g., both frequent small transfers and infrequent large transfers). Any data transfer mechanism provided by the compute fabric is typically designed to support the most common use case: e.g., FuncX communicates function inputs and results via the cloud which adds overheads and incurs financial costs, so that payloads are limited to 10 MB. Decoupling the data fabric from compute fabric can enable greater flexibility in data management. Here, we use ProxyStore, a system that enables pass-by-reference functionality in a diverse range of distributed applications without any modifications to application code.

Passing data by reference is key to dissociating data transfer from control flow. References are small so can be efficiently moved along with function bodies, and can be passed between many resources without having to incur costs for the movement of the actual data. That is, the data are only dereferenced once on the target resource regardless of how many intermediate resources through which the reference was passed. This ensures processes or services responsible for control flow do not become an I/O bottleneck.

Rather than exposing traditional `get`/`set` operations, ProxyStore uses the *proxy* model for seamless pass-by-reference functionality in Python. This easy-to-use programming paradigm allows users to dynamically change transfer methods without needing to modify task code. ProxyStore supports many backend object stores and transfer methods to provide efficient transfer of objects between processes located within the same resource or across different resources.

Proxies are used to intercept and redefine operations on a *target* object. ProxyStore uses lazy transparent object proxies that behave identically to the target object which is achieved by the proxy forwarding all operations on itself to the target. Lazy proxies provide just-in-time *resolution* of the target via a *factory* function. Factories resolve the target when called; a proxy, initialized with a factory, calls the factory and retrieves the target the first time the proxy is accessed.

When the `proxy()` function is called on a target object, the object is placed in a backend store, a factory capable of resolving the object from the store is created, and a proxy, initialized with the factory, is returned. The resulting proxy is the lightweight reference that can be efficiently transmitted. A function that receives a proxy uses the proxy as if it were the target object due to the proxy's transparent nature so no modification of function code is needed.

In this work, we use the Redis, file system, and Globus backends for ProxyStore. The Redis backend is ideal when resources exist within a single, fast network. The file system backend supports scenarios where separate systems have access to a shared file system. The Globus backend is used for multi-resource applications that lack a shared file system.

D. Colmena: Steering Policies as Cooperative Agents

Colmena is a Python library for expressing the steering of dynamic workflows as a collection of interacting agents, which are known collectively as a *Thinker*. The Thinker controls what tasks are performed and how resources are allocated by a workflow system over time based on behavior defined in the agents. For example, one agent may submit a retraining task after a certain number of simulation tasks complete, and another may submit a new simulation when resources are available. Agents, each running as a separate Python thread, interact via Python's threading primitives (e.g., the simulation agent may consume tasks from a queue populated from a task scoring agent). A Thinker communicates task requests to a *Task Server* that employs some compute fabric (e.g., Parsl or FuncX) to execute tasks on distributed resources and return results back to the Thinker.

Colmena integrates support for ProxyStore by automatically creating proxies for objects larger than a user-specified size. The threshold size and ProxyStore backend can vary between tasks types, and users can also proxy objects manually before submitting the proxies to tasks. The flexibility in how proxies are generated makes it possible to adopt different data fabrics between sites and to cache data needed for a computation ahead of time. Regardless of whether Colmena's automatic proxying is used or the user manually proxies objects, no changes to user task code or the compute fabric (i.e., FuncX) are necessary with ProxyStore.

V. Results and Discussion

We started by evaluating the data fabric in a simplified, synthetic application (§V-C), then evaluated our cloud-managed approach in depth for one of the two applications (§V-D). Finally, we contrast the performance differences between a cloud-hosted approach and a baseline that lacks our advanced data fabric using both motivating applications (§V-E).

A. Computational Resources and Software Configuration

We use the computational resources of the Argonne Leadership Computing Facility (ALCF) and Computing, Environment, and Life Sciences (CELS) directorate for our experiments. The Knights Landing nodes of ALCF's Theta supercomputer are used to perform the "simulation" components of each workflow. We use two NVIDIA DGX servers, one with twenty T4 GPUs (known as Venti) and another with eight V100 GPUs (Lambda) for "AI" tasks (e.g., training, inference). The Lambda system represents an "off site" resource because, even though Lambda is housed in the same building, it exists on a separate network, does not have access to any Theta file systems, and uses a different authentication procedure.

The Colmena Thinker and Task Server reside on a login node of Theta.

B. Workflow Configurations

We compare three different workflow system configurations. Our two baselines use Parsl, which requires open ports or a
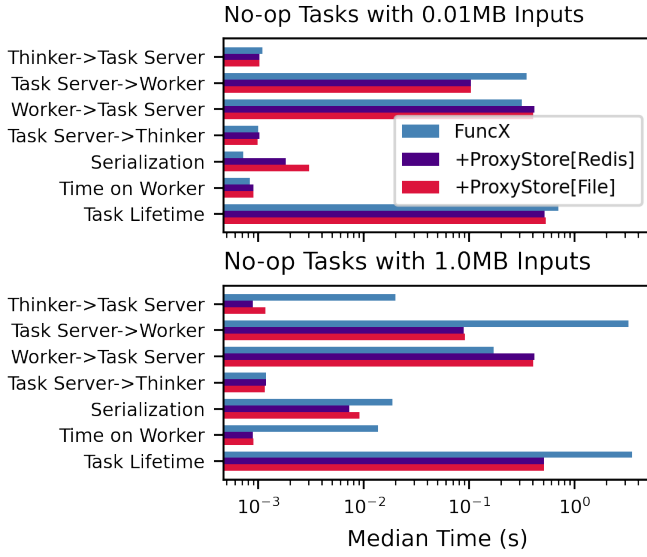
Fig. 3. Median times for different components of the end-to-end execution of a no-op task with Colmena. The use of ProxyStore to transfer objects reduces communication costs for both small (10 kB) and large (1 MB) task inputs. ProxyStore can avoid repeated serializations and deserializations of object transmitted through the Task Server and FuncX service.

tunnel to each computing resource to communicate task information. All configurations require creating a Python virtual environment on each computing resource.

1) **Parsl**: Our baseline without ProxyStore. Requires two ports for Parsl to communicate tasks to, and results from, a remote system.

2) **Parsl+Redis**: Our baseline with ProxyStore using Redis to communicate task data across sites and the file system for local tasks. Requires a third port for Redis in addition to two for Parsl.

3) **FuncX+Globus**: Uses FuncX to communicate task instructions, and ProxyStore with Globus for task data across sites and the file system for local tasks. Requires no open ports besides those used by Globus which are already configured at most computing centers.

C. Evaluating Data Transfer using Synthetic Applications

We first investigate using pass-by-reference to reduce overheads in Colmena and FuncX. Then we profile ProxyStore backends to guide our deployment strategies.

1) Avoiding FuncX Overheads with ProxyStore: We first compare the performance achieved when communicating task inputs with FuncX and two ProxyStore backends: shared file system and Redis. Specifically, we aim to quantify overhead improvements to the Task Server and FuncX by transferring large objects via alternative means through ProxyStore.

We 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 characteristics of FuncX. FuncX stores function arguments and results smaller than 20 kB in an Amazon ElastiCache Redis store and objects greater than 20 kB in Amazon S3. In all experiments, the

Thinker and Task Server are located on a Theta login node, and we use a FuncX 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 Fig. 3 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.

Communication between the Task Server and worker dominates the overall task lifetime because this communication must go through FuncX's cloud service. For small 10 kB inputs, passing objects via proxies can reduce the Task Server to worker communication time by 2–3× and as much as 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 backend performs similar to without ProxyStore, but larger object 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 FuncX. ProxyStore avoids additional deserialization and serialization of the input data because the data is replaced by a lightweight proxy.

A workflow using Colmena and FuncX for data movement will not be able to respond to or initiate tasks with low latency as data sizes increase. Having an alternative method of object communication that can enable direct exchange of objects between the Thinker and workers is vital to avoid overloading the intermediate systems.

2) ProxyStore Backends: As discussed in §IV-C, we employ multiple ProxyStore backends to support a range of task characteristics and computing environments. For example, Redis can be faster than file system I/O for certain object sizes, but opening ports for Redis may not be possible in all environments. Here we benchmark the ProxyStore backends to guide the deployment of our motivating applications.

Specifically, we measure components of a task's life cycle with three different ProxyStore backends, Redis, file system, and Globus, across a range of task input sizes varying from 10 kB to 100 MB. We use the same no-op and no-output tasks
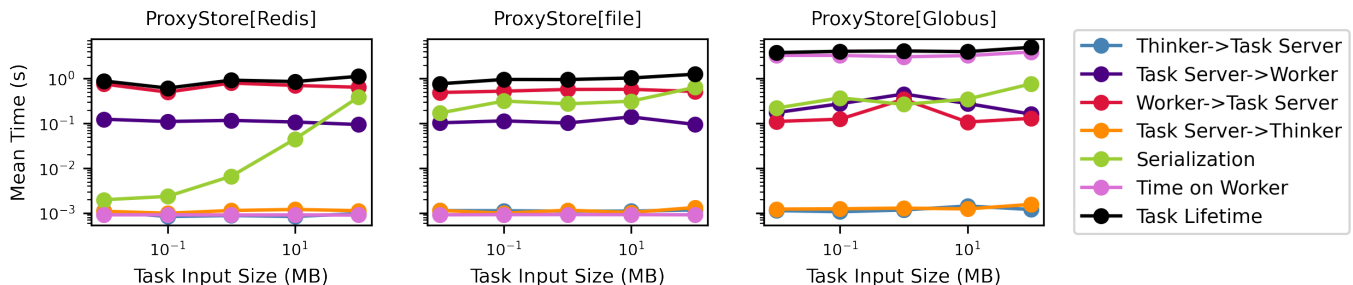
Fig. 4. Mean times of components in the lifetime of a Colmena task. The Redis, file system, and Globus ProxyStore backends are used to proxy inputs, ranging in size from 10 kB to 100 MB, to no-op tasks. The Redis backend provides low latency while the file system backend performs well with large objects (∼100 MB). Mean time of a task spent on a worker increases with Globus because the task must wait on the web-based data transfer to complete.

as in the prior experiment as well as the same Theta KNL node endpoint. The Redis and file system backends enable intra-site communication, so we place the Thinker and Task Server on a Theta login node. The Globus backend is used for inter-site communication so the Thinker and Task Server reside on a login node of a cluster at the University of Chicago.

Comparing the performance of Redis-backed and file system-backed ProxyStore in Fig. 4, we find the only significant differences to be in serialization time, which includes storing objects in the ProxyStore backend. Latencies are much lower when using Redis in the case of smaller objects, but are comparable across Redis and file system backends for larger objects. The "time on worker" is consistent between the Redis and file system backends and across task input sizes.

The serialization time is similar between the file system and Globus backends. Objects are still written to the shared file system prior to starting a Globus transfer operation. Thus, the serialization performance in both cases is a reflection of the I/O performance of the file system. The Globus backend does incur higher overheads in the time spent on the worker, because a proxy must wait for the Globus transfer of its target object to finish before the proxy can be resolved. This can take up to a few seconds, but the performance is constant with task input size (up to 100 MB), which indicates the communication bottleneck is due to the Globus web service latency rather than bandwidth limitations of the Thinker.

Overall, there are clear trade offs between backends for ProxyStore. A Redis backend is optimal for low latency access to small objects within a single site, whereas a shared file system backend is easier to use and provides good performance for large objects such as ML models. Globus can enable multi-site workloads with low overheads when networking restrictions limit other options, and is competitive with Redis for dataset sizes beyond 10 MB. All of these choices are accessible by changing a single line of code in ProxyStore, enabling workflows to be easily optimized.

D. Assessing System Performance in a Science Application

We next implement our science applications using our multi-resource workflow system and ensure that communication overheads are not bottlenecks in system performance. An effective steering system must minimize three forms of latency:
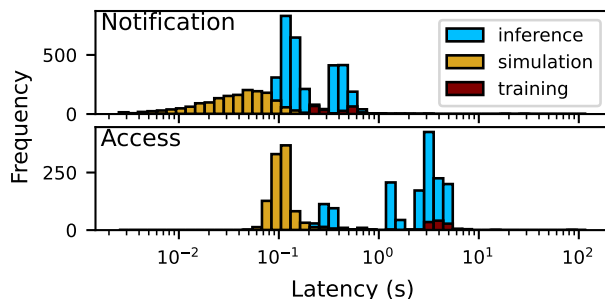

Fig. 5. Result notification timings for Molecular Design application. Above: Time between when a tasks finishes computing and when the Thinker is notified that the computation has finished. Below: How long the Thinker waits to access the result data.

(a) between results completing and being available to the Thinker (reaction time), (b) between results being received and new decisions being made (decision time), and (c) between decisions made and new computations started (dispatch time). We quantify these latencies by studying our molecular design application (§III-A). These experiments deploy the Thinker on a Theta login node, simulations on Theta KNL compute nodes, and use Lambda for AI tasks.

1) Reaction Time: Reaction time latency has two components: the time to notify the Thinker process of a task and the time to make the result data available. Our use of ProxyStore means that the two processes occur separately.

Notification includes serializing the data, creating a proxy object, and communicating the result proxy to the Thinker. As shown in Fig. 5, notification for simulation tasks is faster (median 50 ms) than inference or training as it does not require initializing a Globus transfer because the Thinker and simulation worker share a file system. Starting a transfer between resources requires an HTTPS request to Globus which takes ∼100 ms for inference and training tasks.

Data transfer latency—the time taken to access a result— is only greater than 1 s when data is transferred between resources, as when moving between CPU and GPU machines for inference and training. Transferring between resources requires a Globus transfer, which typically completes in 1–5 s, depending on data transfer nodes utilization and concurrent

transfer limits per user. Some Globus transfers complete while the Thinker is processing results of previous tasks, which is visible in inference data latencies that are sub-500 ms.

2) Decision Time: Our steering policy is designed to minimize the points where decision making is time-critical.

The first latency-sensitive decision is to start a new simulation after another completes. This decision can be made rapidly because it does not require accessing the result data. The median time between a Thinker receiving a completed simulation and sending the next to the Task Server is only 4 ms, which is negligible in comparison to Reaction Time.

Decisions based on model training and inference results involve reading the contents of a result, which requires deserializing the message and resolving any proxies. Responding to model training and inference events takes a median of 4 s and 3 s, respectively; both times are primarily waiting for the data transfer to complete (as in Fig. 5). Thus the time to receive data from remote systems is, again, the most important factor in application performance.

3) Dispatch Time: The time for a chosen task to begin executing on a remote system, the *dispatch time*, is the final type of latency we consider. We hide the dispatch latency in many places with a backlog of queued tasks, but this latency cannot be hidden in three places: when we begin training models, when a batch of inference tasks are submitted after the first model completes training, and when a simulation worker completes. In each case, we know at least one worker on a resource is available, and we want to provide this worker a new task as fast as possible.

Data transfer across multiple resources is the primary source of latency for inference and training tasks. The mean latencies for starting the first inference task of a batch and a training task are 21 and 11 s, respectively; resolving the data proxy accounts for an average of 20.9 s (99%) and 9 s (82%) of those times. Each of these latencies is significant in terms of the average task times. The inference latency, in particular, is almost half of the average runtime of the task (52 s). The effect of data movement latency is less severe for later inference tasks (an average of 8 s for the second inference) because the Globus transfer completes while the first task is in progress. Further, 25% of the second tasks read from ProxyStore's in memory cache and gather inputs in <1 ms.

The latency of the simulation dispatch times is minimal. The time is dominated by communicating the task request via FuncX, which is a median of 330 ms. The dispatch time is less than 1% of the total task runtime and, consequently, not our main target for improvement.

4) Assessment: The latency of responding to a completed calculation across multiple resources is typically below 1 s but can be as high as 10 s under worst-case conditions. As illustrated in the previous sections, Colmena is alerted of a new result within 100 s and acting on the result can take up to 3 s if data must be transferred from a remote system. Starting a new task on a new system can be lower than 1 s, unless dataset transfer times become limiting. As suggested from our synthetic application, the time to start a Globus transfer is our
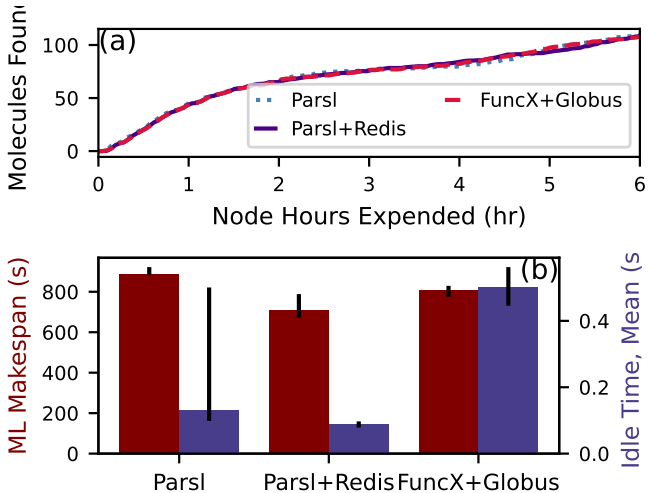

Fig. 6. Comparison of different implementations of the multi-site active learning pipeline. An implementation using only Parsl, Parsl using ProxyStore with Redis, and FuncX using ProxyStore with Globus. (a) Number of top-performing molecules found as a function of simulation time expended.(b) Median time required to reorder a task queue and median idle time for simulation workers. Error bars represent the 40th and 60th percentiles.

rate-limiting step of no less than a few seconds for smaller data.

E. Comparing System Performance

We study the performance differences between workflow systems (see §V-B) with two metrics: scientific performance and responsiveness of the workflow system. We study both of our applications and use Venti for training and inference tasks, and Theta for simulation and sampling tasks.

1) Molecular Design: We compared scientific outcomes by measuring how many molecules with IP > 14 were found after 6 node hours of compute and averaged the value over three runs of each implementation. Our best effort with open ports, Parsl+Redis, has equivalent outcomes to FuncX+Globus (108.0 vs. 108.3). The results are within the margin of error as the Parsl+Redis version varied between 105 and 111 suitable molecules found. We therefore conclude that direct connections between computing providers are an unnecessary complication to deploying our multi-resource application.

The key responsiveness needed by our application is the time to update the task list given new data and rate at which new simulations are evaluated. We define the time to update the task list as the time from when the steering policy requests models to be retrained to when the results from all inference tasks are used to reprioritize the task queue. As shown in Fig. 6a, the FuncX implementation completes the ML tasks faster (in 806 s, on average) than native Parsl (880 s) but is slower than Parsl+ProxyStore backed by Redis (708 s). There is a clear advantage to using pass-by-reference and, given the moderate data sizes (10-100 MB) of our ML tasks, the low-latency transfer of Redis is a better choice for the data fabric.

Our other key latency metric is CPU utilization. CPU nodes are idle between one simulation completing and the next

starting, which requires two fast operations: notifying the Thinker of a completion and dispatching the next task. We find the average idle time between tasks is less than 500 ms for FuncX and below 100 ms for Parsl with a Redis ProxyStore, and in both cases overall CPU utilization is >99%.

2) Surrogate Finetuning: The surrogate models produced using our cloud-managed workflows are indistinguishable from those produced using a conventional workflow solution. The Root Mean Squared Deviation (RMSD) of the forces predicted using the models from the FuncX solution are 1.56 ± 0.12 eV/Å, compared to 1.61 ± 0.17 eV/Å for Parsl with ProxyStore and 1.71 ± 0.09 eV/Å without. Run-to-run variations are larger than variation between the applications.

While the three approaches show comparable scientific performance, the task overhead when using FuncX and Globus is clearly larger. We measure this overhead as the time between when a task was created and when the result was read that is not the task running. When tasks are run on remote GPUs, overhead is dominated by the time to submit a request to transfer data to Globus.[1] The second largest component is the data transfer time, which is roughly 1.5 s per direction and consistent with the results of synthetic experiments (Fig. 4). Transferring to remote sites via Redis is faster, but requires configuring a tunnel between resources.

CPU task overheads are dominated by the time to notify the Thinker of task completion for FuncX and data transfer for Parsl, regardless of task data size. In contrast, the overhead for Parsl without pass-by-reference is strongly dependent on data size; 820 ms for sampling tasks (3 MB) and 20 ms for simulation tasks (20 kB). The data transfer times for Parsl with pass-by-reference are consistent for these two task types at 200 and 170 ms for the sampling and simulation, respectively. We can see the benefit of pass-by-reference for larger messages but also that our application could be accelerated by avoiding the overhead of proxying small messages.

F. Recommendations

Our experience leads to several recommendations.

- Use pass-by-reference and intelligent steering policies to achieve 10x reductions in application latency.
- Transmit data between sites directly for data larger than 10 kB. If messages are smaller than 100 MB and direct connection between resources is feasible, Redis is ideal, otherwise Globus is the best choice.
- Employ pass-by-reference with a conventional workflow system (e.g., Parsl) if data are larger than 10 kB, especially if data are reused between tasks.

VI. Conclusions

We have presented experiences deploying two scientific applications across multiple heterogeneous resources. The applications share a need to run simulation and AI tasks on separate resources, but differ in the amount of data transfer

---

[1]The large submit time was fixed by Globus, but the fix will not be in production by the paper deadline. We anticipate the transfer request time to be 10x faster with the new version.
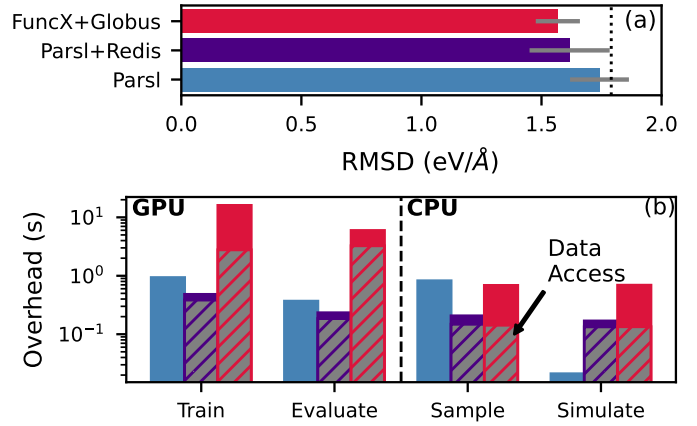

Fig. 7. Comparison of (a) Root Mean Squared Deviation (RMSD) in predicted forces on a test set and (b) median overheads per task type for three different workflow systems. (a) Error bars are the standard error of the mean over three tests. (b) The time spent waiting for result data is shown in gray for the systems where ProxyStore is used to transfer data separately from task instructions (Redis store with Parsl, Globus store with FuncX).

by an order of magnitude and in the frequency of AI tasks being required. Our goal was to do so by leveraging cloud-hosted function-as-a-service (FaaS) and data transfer services to simplify deployment and ensure security, but without giving up performance. We employed a combination of FuncX and Globus Transfer to pass task instructions and data between resources, ProxyStore to allow data transfer via Globus without modifying application code, and Colmena to express steering policies that hide data transfer latencies. We show that our implementation achieves scientific outputs that are indistinguishable from those produced by implementations that require direct connections between resources.

References

[1] R. Vescovi *et al.*, "Linking scientific instruments and computation: Patterns, technologies, and experiences," *Patterns*, vol. 3, no. 10, 2022.

[2] A. Brace *et al.*, "Stream-AI-MD: Streaming AI-driven adaptive molecular simulations for heterogeneous computing platforms," in *Platform for Advanced Scientific Computing Conference*, 2021, pp. 1–13.

[3] L. Smarr and C. E. Catlett, "Metacomputing," *Communications of the ACM*, vol. 35, no. 6, pp. 44–52, 1992.

[4] S. Brunett *et al.*, "Application experiences with the Globus toolkit," in *7th Intl Symp on High Performance Dist Computing*, 1998, pp. 81–88.

[5] K. Chard, S. Tuecke, and I. Foster, "Efficient and secure transfer, synchronization, and sharing of big data," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 46–55, 2014.

[6] R. Chard *et al.*, "funcX: A federated function serving fabric for science," in *29th Intl Symp on High-Performance Parallel Dist Computing*, 2020.

[7] Y. Babuji *et al.*, "Parsl: Pervasive parallel programming in Python," in *Intl Symp on High-Performance Parallel and Dist Computing*, 2019.

[8] M. Feller, I. Foster, and S. Martin, "GT4 GRAM: A functionality and performance study," in *TeraGrid Conference*, 2007.

[9] S. Cholia, D. Skinner, and J. Boverhof, "Newt: A restful service for building high performance computing web applications," in *Gateway Computing Environments Workshop*, 2010, pp. 1–11.

[10] J. Stubbs *et al.*, "Tapis: An API platform for reproducible, distributed computational research," in *Advances in Information and Communication*, K. Arai, Ed., 2021, pp. 878–900.

[11] E. Deelman *et al.*, "The evolution of the Pegasus workflow management software," *Computing in Science & Engineering*, vol. 21, no. 4, 2019.

[12] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *14th Python in Science Conference*, 2015.

[13] Y. Zhao *et al.*, "Swift: Fast, reliable, loosely coupled parallel computation," in *IEEE Congress on Services*, 2007, pp. 199–206.

[14] E. Afgan *et al.*, "The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2022 update," *Nucleic Acids Research*, vol. 50, no. W1, pp. W345–W351, Apr. 2022.

[15] V. Balasubramanian *et al.*, "RADICAL-Cybertools: Middleware building blocks for scalable science," 2019.

[16] Amazon Lambda. https://aws.amazon.com/lambda. Accessed Jan 2023.

[17] Google Cloud Functions. https://cloud.google.com/functions/. Accessed Jan 2023.

[18] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed Jan 2023.

[19] I. Foster and D. B. Gannon, *Cloud Computing for Science and Engineering*. MIT Press, 2017.

[20] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: The prospect of serverless scientific computing and HPC," in *Latin American High Performance Computing Conference*, 2017, pp. 154–168.

[21] M. Malawski, "Towards serverless execution of scientific workflows–HyperFlow case ssudy," in *Workshop on Workflows in Support of Large-Scale Science*, 2016, pp. 25–33.

[22] G. Fox and S. Jha, "Conceptualizing a computing platform for science beyond 2020," in *10th Intl Conference on Cloud Computing*, 2017.

[23] G. Kiar *et al.*, "A serverless tool for platform agnostic computational experiment management," *Frontiers in Neuroinformatics*, vol. 13, 2019.

[24] Apache OpenWhisk. http://openwhisk.apache.org/. Accessed Jan 2023.

[25] Fn project. https://fnproject.io. Accessed Jan 2023.

[26] KNIX MicroFunctions. https://github.com/knix-microfunctions/knix. Accessed Jan 2023.

[27] C. Garcia *et al.*, "Abaco–a modern platform for highthroughput parallel scientific computations," in *Intl Workshop on Science Gateways*, 2020.

[28] S. Ghaemi, H. Khazaei, and P. Musilek, "ChainFaaS: An open blockchain-based serverless platform," *IEEE Access*, vol. 8, 2020.

[29] M. Ciavotta *et al.*, "DFaaS: Decentralized function-as-a-service for federated edge computing," in *10th Intl Conf Cloud Networking*, 2021.

[30] N. J. Carriero *et al.*, "The Linda alternative to message-passing systems," *Parallel Computing*, vol. 20, no. 4, pp. 633–655, 1994.

[31] M. F. Aktas *et al.*, "WA-Dataspaces: Exploring the data staging abstractions for wide-area distributed scientific workflows," in *46th Intl Conference on Parallel Processing*, 2017, pp. 251–260.

[32] J. Soumagne *et al.*, "Mercury: Enabling remote procedure call for high-performance computing," in *Intl Conf Cluster Computing*, 2013, pp. 1–8.

[33] R. Ross *et al.*, "Mochi: Composing data services for high-performance computing environments," *J Computer Science and Technology*, 2020.

[34] M. F. Aktas *et al.*, "WA-DataSpaces: Exploring the data staging abstractions for wide-area distributed scientific workflows," in *46th Intl Conference on Parallel Processing*, 2017, pp. 251–260.

[35] E. Dart *et al.*, "The Science DMZ: A Network Design Pattern for Data-Intensive Science," ser. SC'13, 2013.

[36] J. Chung *et al.*, "SciStream: Architecture and toolkit for data streaming between federated science instruments," in *31st Intl Symp on High-Performance Parallel and Distributed Computing*, 2022, p. 185–198.

[37] S. Jha, V. R. Pascuzzi, and M. Turilli, "AI-coupled HPC workflows," 2022. [Online]. Available: https://arxiv.org/abs/2208.11745

[38] J. Ejarque *et al.*, "Enabling dynamic and intelligent workflows for HPC, data analytics, and AI convergence," *Future Generation Computer Systems*, vol. 134, pp. 414–429, Sep. 2022.

[39] ATLAS Collaboration, "AtlFast3: The next generation of fast simulation in ATLAS," *Computing and Software for Big Science*, vol. 6, p. 7, 2022.

[40] A. Dunn, J. Brenneck, and A. Jain, "Rocketsled: A software library for optimizing high-throughput computational searches," *J Physics: Materials*, vol. 2, no. 3, p. 034002, Apr. 2019.

[41] J. H. Montoya *et al.*, "Autonomous intelligent agents for accelerated materials discovery," *Chemical Science*, vol. 11, no. 32, 2020.

[42] H. Lee *et al.*, "DeepDriveMD: Deep-learning driven adaptive molecular simulations for protein folding," in *3rd Workshop on Deep Learning on Supercomputers*, Nov. 2019.

[43] J. M. Wozniak *et al.*, "CANDLE/Supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinformatics*, vol. 19, no. S18, Dec. 2018.

[44] P. Balaprakash *et al.*, "DeepHyper: Asynchronous hyperparameter search for deep neural networks," in *25th Intl Conference on HPC*, 2018.

[45] S. Hudson *et al.*, "libEnsemble users manual," Argonne National Laboratory, Tech. Rep. Revision 0.7.1, 2020.

[46] P. Moritz *et al.*, "Ray: A distributed framework for emerging AI applications," in *13th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2018, pp. 561–577.

[47] O. Yildiz *et al.*, "Dynamic heterogeneous task specification and execution for in situ workflows," in *Workshop on Workflows in Support of Large-Scale Science*, Nov. 2021.

[48] L. Ward *et al.*, "Dataset for cloud services enable efficient ai-guided simulation workflows across heterogeneous resources," 2023. [Online]. Available: https://petreldata.net/mdf/detail/multiresource_ai_v1.1

[49] D. Polykovskiy *et al.*, "Molecular Sets (MOSES): A benchmarking platform for molecular generation models," *Frontiers in Pharmacology*, 2020.

[50] Y. Babuji *et al.*, "Targeting SARS-CoV-2 with AI- and HPC-enabled lead generation: A first data release," 2020, arXiv:2006.02431.

[51] D. A. Cohn, Z. Ghahramani, and M. I. Jordan, "Active learning with statistical models," *J Artificial Intelligence Research*, vol. 4, 1996.

[52] M. Li *et al.*, "Recent advancements in rational design of non-aqueous organic redox flow batteries," *Sustainable Energy & Fuels*, vol. 4, no. 9, pp. 4370–4389, 2020.

[53] G. Landrum, "RDKit: Open-source cheminformatics," https://www.rdkit.org. Visited May 1, 2021.

[54] L.-P. Wang and C. Song, "Geometry optimization made simple with translation and rotation coordinates," *J Chemical Physics*, vol. 144, no. 21, p. 214108, 2016.

[55] D. G. A. Smith *et al.*, "The MolSSI QCArchive project: An open-source platform to compute, organize, and share quantum chemistry data," *WIREs Computational Molecular Science*, vol. 11, no. 2, 2020.

[56] C. Bannwarth *et al.*, "Extended tight-binding quantum chemistry methods," *WIREs Comput. Mol. Sci.*, vol. 11, p. e01493, 2020.

[57] P. St. John *et al.*, "Message-passing neural networks for high-throughput polymer screening," *J Chemical Physics*, vol. 150, no. 23, 2019.

[58] G. S. Fanourgakis and S. S. Xantheas, "The flexible, polarizable, thole-type interaction potential for water (TTM2-f) revisited," *J Physical Chemistry A*, vol. 110, no. 11, pp. 4100–4106, Mar. 2006.

[59] S. Choudhury *et al.*, "Hydronet: Benchmark tasks for preserving intermolecular interactions and structural motifs in predictive and generative models for molecular data," *Machine Learning and the Physical Sciences Workshop at NeurIPS*, 2020.

[60] K. T. Schütt *et al.*, "SchNet – a deep learning architecture for molecules and materials," *J Chemical Physics*, vol. 148, no. 24, p. 241722, 2018.

[61] D. Smith *et al.*, "Psi4 1.4: Open-source software for high-throughput quantum chemistry," *J Chemical Physics*, vol. 152, no. 18, 2020.

[62] A. H. Larsen *et al.*, "The atomic simulation environment—a Python library for working with atoms," *J Physics: Condensed Matter*, vol. 29, no. 27, p. 273002, Jun. 2017.

[63] S. Tuecke *et al.*, "Globus Auth: A research identity and access management platform," in *12th Intl Conf e-Science*, 2016, pp. 203–212.