funcX: Federated Function as a Service for Science

Zhuozhao Li, Ryan Chard, Yadu Babuji, Ben Galewsky, Tyler Skluzacek, Kirill Nagaitsev, Anna Woodard, Ben Blaiszik, Josh Bryan, Daniel S. Katz, *Senior Member, IEEE,* Ian Foster, *Fellow, IEEE,* and Kyle Chard

Abstract—*funcX* is a distributed function as a service (FaaS) platform that enables flexible, scalable, and high performance remote function execution. Unlike centralized FaaS systems, *funcX* decouples the cloud-hosted management functionality from the edge-hosted execution functionality. *funcX*'s endpoint software can be deployed, by users or administrators, on arbitrary laptops, clouds, clusters, and supercomputers, in effect turning them into function serving systems. *funcX*'s cloud-hosted service provides a single location for registering, sharing, and managing both functions and endpoints. It allows for transparent, secure, and reliable function execution across the federated ecosystem of endpoints—enabling users to route functions to endpoints based on specific needs. *funcX* uses containers (e.g., Docker, Singularity, and Shifter) to provide common execution environments across endpoints. *funcX* implements various container management strategies to execute functions with high performance and efficiency on diverse *funcX* endpoints. *funcX* also integrates with an in-memory data store and Globus for managing data that may span endpoints. We motivate the need for *funcX*, present our prototype design and implementation, and demonstrate, via experiments on two supercomputers, that *funcX* can scale to more than 130 000 concurrent workers. We show that *funcX*'s container warming-aware routing algorithm can reduce the completion time for 3000 functions by up to 61% compared to a randomized algorithm and the in-memory data store can speed up data transfers by up to 3x compared to a shared file system.

Index Terms—Function-as-a-Service, cyberinfrastructure, distributed computing

1 Introduction

The exponential growth of data and increasing hardware diversity is driving the need for computation to occur wherever it makes the most sense, for example, on a suitable computer, where particular software is available, or near data. Prior research, in grid [39] and peer-to-peer [60] computing, has studied and explored the foundations for remote computing. However, with the exception of cloud platforms, general-purpose remote computation has remained elusive due to, for example, slow and unreliable network communications, security challenges, and dependencies between software and heterogeneous computer architectures.

Commercial cloud providers have been at the forefront of recent advances in networks, hardware, and distributed computing, leveraging widespread virtualization, universal trust fabrics, and high-speed networks to deliver serverless computing services such as function-as-a-service (FaaS) [27], [40], [77]. FaaS enables developers to register a high-level programming function and to then invoke that function many times by passing input arguments. The user needs not concern themselves with provisioning infrastructure or configuring execution environments. FaaS systems have quickly become integral to a wide range of applications, particularly for event-based and dev-ops applications.

The FaaS model is particularly attractive in science as a way of decomposing monolithic science applications into a collection of modular, performant, and extensible functions [38], [41], [48], [58], [72]. However, existing FaaS systems are typically centralized and specific to a particular cloud, rather than being designed to be deployed on heterogeneous research cyberinfrastructure (CI) or to use federated resources. Typically research CI uses batch scheduling interfaces and inflexible authentication and authorization models, which does not lend itself to the fine-grain and sporadic function workloads. In response, we propose a federated FaaS model for general-purpose remote computing at scale across diverse CIs, both centrally and at the edge.

In this paper, we present *funcX*, a federated, scalable, and high-performance function execution platform. *funcX* leverages a *distributed endpoint model* to support remote function execution across distributed and heterogeneous research CI. Users can transform many computing resources, such as laptops, clouds, clusters, supercomputers, or Raspberry Pis they are authorized to access, into function serving systems by deploying *funcX*'s endpoint software. Users then use the cloud-hosted *funcX* service to register Python functions and invoke those functions on their deployed endpoints. *funcX* manages the reliable and secure execution of functions, staging function code and inputs, provisioning resources, managing safe and secure execution (optionally in containers), monitoring execution, and returning outputs to users.

- *Z. Li is with Department of Computer Science and Engineering and Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, Shenzhen, China. Email: lizz@sustech.edu.cn*
- *Y. Babuji, T. J. Skluzacek, A. Woodard, B. Blaiszik, J. Bryan, I. Foster, and K. Chard are with the University of Chicago, Chicago, IL, 60637. E-mail: {yadunand, skluzacek, annawoodard, blaiszik, jbryan, foster, chard}@uchicago.edu*
- *R. Chard and I. Foster are with Argonne National Laboratory, Lemont, IL 60439. E-mail: {rchard, foster}@anl.gov*
- *D. S. Katz is with NCSA, CS, ECE, and the iSchool, University of Illinois, Urbana, IL, 61801. E-mail: d.katz@ieee.org*
- *B. Galewsky is with NCSA, University of Illinois, Urbana, IL, 61801. E-mail: bengal1@illinois.edu*
- *K. Nagaitsev is with Northwestern University, Evanston, IL, 60208. E-mail: knagaitsev@u.northwestern.edu*

Thus, users benefit from the convenience and reliability of a cloud-hosted service combined with the flexibility and performance of a federated ecosystem of endpoints.

We extend our previous work [32] to support complex data dependencies between scientific functions. Specifically, we focus on enabling data transfer between functions that are executing on the same (*intra-endpoint*) or different (*inter-endpoint*) endpoints. For intra-endpoint communication we use an in-memory data store, for inter-endpoint communication we use Globus [37]. We also present new heuristic-based container management and function routing schemes that reduce container warming overhead and efficiently route functions to appropriately configured containers.

The primary novelty of our work is in the adaptation of the FaaS paradigm to a federated research ecosystem, combining a distributed endpoint model with a hosted FaaS platform to support remote function execution across distributed and heterogeneous research CI. We demonstrate the viability of our approach with a highly modularized and extensible design as well as a scalable and performant implementation. We also show that it is beneficial to decompose scientific applications into monolithic functions that may be executed on different remote resources. The contributions of our work are as follows:

- *funcX*, a distributed and federated FaaS platform that can: be deployed on research CI, dynamically provision and manage resources, leverage various container technologies, and facilitate secure, scalable, and distributed function execution.
- Automated data movement between functions using widely-used in-memory data stores and high-performance data transfer technology to transparently support data dependencies between functions.
- Design and evaluation of performance enhancements for function serving on distributed research CI, including function warming, batching, and function routing.
- Experimental studies showing that *funcX* delivers execution latencies comparable to those of commercial FaaS platforms and scales to 1M+ functions across 130K active workers on supercomputers.

The rest of this paper is as follows. §2 describes an example use case and presents general requirements for FaaS in science. §3 presents a conceptual model of *funcX*. §4 describes the *funcX* system architecture. §5 discusses how data is managed in *funcX*. §6 presents *funcX*'s container management model. §7 evaluates *funcX* performance. §8 reviews *funcX*'s use in scientific case studies. §9 discusses related work. Finally, §10 summarizes our contributions.

2 Motivations and Requirements

Over the last two years the scientific community has been working to understand SARS-CoV-2 and develop effective tests, therapeutics, and vaccines. However, progress in these areas is dependent on our ability to understand SARS-CoV-2 protein structures. At Argonne's Advanced Photon Source [49], scientists use an emerging method called fixed-target serial synchrotron crystallography (SSX) to collect physiological temperature data from thousands of protein crystals.

Listing 1: Three functions used in the SSX pipeline and an example of how the funcX SDK is used to register and invoke the `process_stills` function.

```
def process_stills(data):
  inputs = data['inputs']
  phil = data['phil']
  cmd = f'dials.stills_process {phil} {inputs}'
  res = subprocess.run(cmd)
  return res.stdout

def solve(data):
  from gladier.tools import template_prime
  pdata = template_prime.substitute(data['template'])
  cmd = f"prime.run {pdata} > prime.log"
  res = subprocess.run(cmd)
  return res.stdout

def extract_metadata(data):
  from gladier.tools import (get_dims,
    get_lattice_counts, plot_lattice_counts)
  lc = get_lattice_counts(xdim, ydim, int_files)
  plot_name = f'1int-sinc.png'
  plot_lattice_counts(xdim, ydim, lc, plot_name)
  return plot_name

fc = FuncXClient()
func_id = fc.register_function(process_stills)
endpoint_id = '863d-...-d820d'

input_data = {'inputs': '...', 'phil': '...'}
task_id = fc.run(func_id, endpoint_id,
                 data=input_data)
res = fc.get_result(task_id)
```

Data are generated at unprecedented rates with tens of thousands of images captured each hour. Keeping pace with the experiment requires rapid data processing across multiple, heterogeneous computing resources to efficiently analyze, refine, solve, and curate structures.

To meet these data processing and publication needs, SSX scientists have adopted an automated data management framework [81] that can manage data acquisition, analysis, curation, and visualization. Throughout this workflow, there are needs for computation both at the edge to detect and pre-process data rapidly, as well as on HPC resources to perform computationally expensive analysis tasks and produce structures. *Each of these steps relies on different packages and functions, has different processing durations, occurs at different times, and requires different types and amounts of computing resources. Thus, it is essential that the scientists be able to decompose the entire processing pipeline into a series of individual functions to perform on data as they are moved and transformed.* These functions, shown in Listing 1, analyze individual images, refine and solve the crystal structure, and extract metadata and create plots before publishing results.

This typical science use case, with parallels in many other domains described in our previous work [32], highlights the benefits of FaaS approaches (e.g., decomposition, abstraction, flexibility, scalability, reliability), and also elucidates several requirements for FaaS approaches.

- **Research CI**: functions may require HPC-scale and/or specialized and heterogeneous resources. Many resources expose batch scheduler interfaces (with long delays, periodic downtimes, proprietary interfaces) and specialized container technology (e.g., Singularity,

Shifter) that make it challenging to provide common execution interfaces, on-demand and elastic capacity, and fault tolerance.
- **Distribution**: different parts of an application may be most efficiently executed on different, often distributed, resources (e.g., near data, on a specialized computer).
- **Data**: functions analyze both small and large data, stored in various locations and formats, and accessible via different methods (e.g., Globus [31]).
- **Authentication**: institutional identities and specialized security models are used to access data, compute resources, and other cyberinfrastructure.
- **State**: functions may be connected and share state (e.g., files or database connections) to decrease overheads.

Existing FaaS solutions may satisfy these requirements partially, but not completely. For example, some FaaS systems (e.g., OpenWhisk [4], KNIX [14]) support on-premise deployments on specialized hardware (e.g, GPU), but not on distributed and federated computing resources. Some FaaS systems (e.g., DFaaS [34], ChainFaaS [42]) support function execution in distributed environments, but not on research CI. Here we present *funcX*, a federated and scalable FaaS platform that enables researchers to decompose applications into functions and execute them on arbitrary remote computers via the FaaS paradigm.

3 Conceptual Model

We first describe the conceptual model behind *funcX* to provide context to the implementation architecture. *funcX* allows users to register and then execute *functions* on arbitrary *endpoints*. All user interactions with *funcX* are performed via a REST API implemented by a cloud-hosted *funcX* service.

Functions: *funcX* is designed to execute *functions*: snippets of Python code that perform some activity. A *funcX* function explicitly defines a Python function and input signature. The function body must specify all imported modules. While *funcX* supports only Python functions, users can easily write Python functions to invoke tools written in other languages. Listing 1 shows several functions used in the SSX pipeline mentioned in §2. The `process_stills` function takes a single input dictionary as input, which includes the locations of the images and the *phil* file describing the analysis configuration. The function then makes use of the *DIALS* [82] tool to analyze the image.

Function registration: A function must be registered with *funcX* before it can be executed. The registration includes a name and the serialized function body. Optionally, it may also specify users, or groups of users, who may be authorized to invoke the function, and a container image to be used for execution. Containers allow the construction of environments with the dependencies (system packages and Python libraries) required to execute the function. *funcX* assigns a universally unique identifier (UUID) for management and invocation. Users may update functions they own.

Endpoints: A *funcX* endpoint is a logical entity that represents a compute resource. The corresponding *funcX* agent allows the *funcX* service to dispatch functions to that resource for execution. The agent handles authentication and authorization, provisioning of nodes on the compute resource, and monitoring and management. Administrators or users can deploy a *funcX* agent and register an endpoint for themselves and others, providing descriptive (e.g., name, description) metadata. Each endpoint is assigned a UUID for subsequent use.

Function execution: Authorized users may invoke a registered function on a selected endpoint. To do so, they issue a request via *funcX* that identifies the function and endpoint to be used as well as inputs to be passed to the function. Functions are executed asynchronously: each invocation returns an identifier via which progress may be monitored and results retrieved. *In this paper, we refer to an invocation of a function as a "task."* Importantly, following the FaaS model, while users must specify the specific endpoint for use, they do not manage the resources on which the function is executed (e.g., nodes, containers, or modules)

***funcX* service:** Users interact with *funcX* via a cloud-hosted service that exposes a REST API for registering functions and endpoints, and for executing functions, monitoring their execution, and retrieving results. The REST API provides a uniform interface via which users can make asynchronous and stateless calls to manage endpoints and function executions. REST APIs are the the most common interface for FaaS platforms (e.g., AWS Lambda [1] and Google Cloud Functions [11]). The service is connected to accessible endpoints via the endpoint registration process.

User interface: *funcX* provides a Python SDK that wraps the REST API. Listing 1 shows an example of how the SDK can be used to register and invoke a function on a specific endpoint. The example first constructs a *client* and registers the `process_stills` function. It then invokes the registered function using the `run` command, passing the unique function identifier, the endpoint id on which to execute the function, and inputs (in this case `data`). Finally, the example shows that the results can be asynchronously retrieved using `get_result`.

4 Architecture and Implementation

funcX combines a cloud-hosted management service with software agents deployed on remote resources: see Fig. 1.

4.1 The *funcX* Service

The *funcX* service maintains a registry of *funcX* endpoints, functions, and users in a persistent AWS Relational Database Service (RDS) database. To facilitate rapid function dispatch, *funcX* stores serialized function codes and tasks (including inputs and task metadata) in an AWS ElastiCache Redis hashset. The service also manages a Redis queue for each endpoint that stores task IDs for tasks to be dispatched to that endpoint. The service exposes a REST API to register and manage endpoints, register functions, execute and monitor functions, and retrieve the output from tasks. The *funcX* service is secured using Globus Auth [75], which allows users, programs and applications, and *funcX* endpoints to securely make API calls. When an endpoint registers with the *funcX* service, a unique *forwarder* process is created for each endpoint. Endpoints establish secured ZeroMQ connections with their forwarder to receive tasks, return results, and perform heartbeats.

funcX implements a hierarchical task queuing architecture consisting of queues at the *funcX* service, endpoint,

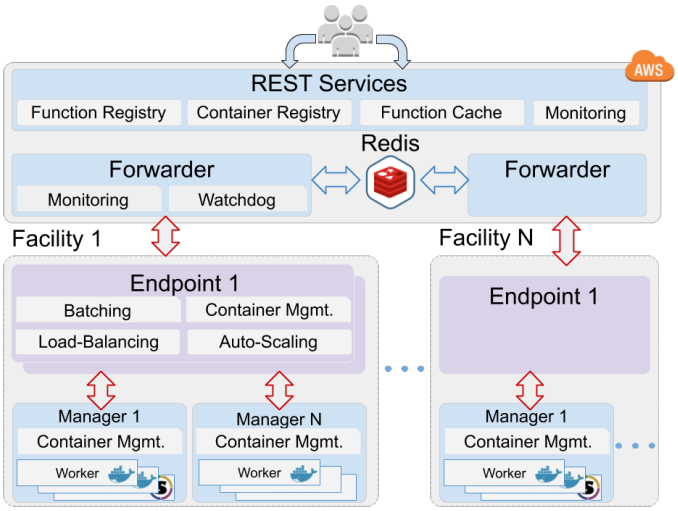

Fig. 1: *funcX* architecture, showing the *funcX* service (top) consisting of a REST interface, Redis store, and Forwarders. *funcX* endpoints (bottom) provision resources and coordinate the execution of functions.

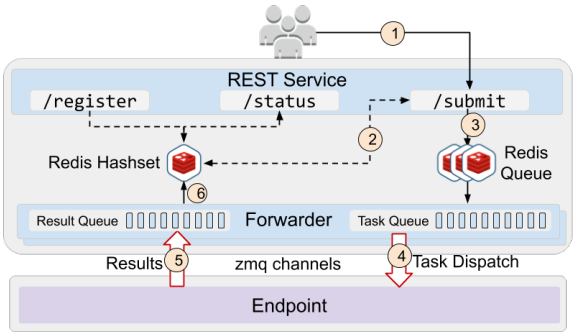

Fig. 2: *funcX* task execution path. A task submitted to *funcX* (1) is stored in Redis (2), queued for execution (3), and dispatched via a Forwarder to an endpoint (4); results are returned (5), then stored in Redis for users to retrieve (6).

and worker. These queues support reliable fire-and-forget function execution that is resilient to failure and intermittent endpoint connectivity. At the first level, each registered endpoint is allocated a unique Redis *task queue* and *result queue* to store and track tasks, which are implemented using Redis `Lists` structure. We use Redis as it provides a simple yet performant system for brokering tasks. Redis is offered as a hosted Amazon service and can be elastically scaled as workload increases. *funcX* serves primarily as a broker to manage and distribute tasks. Redis provides high throughput queuing via an in-memory store with little overhead on the tasks and results passed through the queue—an important requirement for providing low latency execution. One limitation of this approach is that we must implement message acknowledgments to ensure that tasks and results are communicated reliably between clients, endpoints, and the *funcX* service. We note that as use cases expand, we may need to consider other message queues, such as Kafka [3], Pulsar [5], or AMQP-based systems (e.g., RabbitMQ [20]).

Fig. 2 shows the *funcX* task lifecycle. At function submission, the *funcX* service routes the task to the specified endpoint's task queue. The forwarder listens to the queue for tasks and then dispatches the task to the corresponding agent. *funcX* agents internally queue tasks at both the manager and worker. These queues ensure that tasks are not lost once they have been delivered to the endpoint. Results are returned to the *funcX* service and stored in the endpoint's result queue until they are retrieved by the user.

funcX relies on AWS-hosted databases, caches, and Web serving infrastructure to reduce operational overhead, elastically scale resources, and provide high availability. While these services provide significant benefits to *funcX*, they have associated costs. To minimize these costs we apply several techniques, such as using small cloud instances with responsive scaling to minimize the steady state cost and restricting the size of input and output data passed through the *funcX* service to reduce storage (e.g., in Redis store) and data egress costs. Further, we periodically purge results from the Redis store once they have been retrieved by the client or after a period of time.

The *funcX* service is designed to provide robustness and fault tolerance using several techniques. First, the *funcX* service implements health checks, including a liveness check, CPU utilization, and response time monitoring, etc. The service is automatically restarted when these health checks indicate failures. Second, the RDS database and Redis task queue are replicated to ensure that any data (e.g., users, functions, endpoints, and tasks) are not lost. Third, the forwarder uses configurable periodic heartbeats (30 seconds by default) to detect if an agent is disconnected. A task is sent to an agent only when it is connected. When an agent is disconnected, all the tasks dispatched to the agent are returned back into the task queue and the tasks are forwarded to that agent when it reconnects. Fourth, tasks are cached at each layer and only removed when downstream layers have acknowledged receipt. Finally, to serve the distributed endpoints in *funcX*, the *funcX* service is deployed on cloud-hosted services, which internally provide high reliability and robustness.

4.2 Function Containers

funcX uses containers to package function code and dependencies that are to be deployed on a compute resource. Our review of container technologies, including Docker [59], LXC [17], Singularity [54], Shifter [46], and CharlieCloud [63], led us to adopt Docker, Singularity, and Shifter. Docker works well for local and cloud deployments, whereas Singularity and Shifter are designed for use in HPC environments and are supported at large-scale computing facilities (e.g., Singularity at ALCF and Shifter at NERSC). Singularity and Shifter implement similar models and thus it is easy to convert from a common representation (e.g., a Dockerfile) to both formats.

funcX requires that each container includes a base set of software, including Python 3 and *funcX* worker software. Other system libraries or Python modules needed for function execution must also be included. When registering a function, users may optionally specify a container to be used for execution; if no container is specified, *funcX* executes functions using the worker's Python environment.

In future work, we intend to make this process dynamic, using repo2docker [36] to build Docker images and convert them to site-specific container formats as needed.

4.3 The *funcX* Endpoint

The *funcX* endpoint represents a remote resource and delivers high-performance remote execution of functions in a secure, scalable, and reliable manner.

The endpoint architecture, depicted in the lower portion of Fig. 1, is comprised of three components, which are discussed below:

- *funcX agent*: a persistent process that queues and forwards tasks and results, interacts with resource schedulers, and load balances tasks.
- *Manager*: manages the resources for a single compute node on an endpoint by deploying and managing a set of workers.
- *Worker*: executes tasks (optionally within a container).

The *funcX agent* is a software agent that is deployed by a user on a compute resource (e.g., an HPC login node, cloud instance, or a laptop). It registers with the *funcX* service and acts as a conduit for routing tasks and results between the service and workers. The *funcX* agent manages resources on its system by working with the local scheduler or cloud API to deploy *managers* on compute nodes. The *funcX* agent uses a pilot job model [76] to provision and communicate with resources in a uniform manner, irrespective of the resource type (cloud or cluster) or local resource manager (e.g., Slurm, PBS, Cobalt). As each manager is launched on a compute node, it connects to and registers with the *funcX* agent. The *funcX* agent then uses ZeroMQ sockets to communicate with its managers. To minimize blocking, all communication is performed by threads using asynchronous communication patterns. The *funcX* agent uses a randomized scheduling algorithm to allocate tasks to suitable managers with available capacity. The *funcX* agent can be configured to provide access to specialized hardware or accelerators. When deploying the agent, users can specify how worker containers should be launched, enabling them to mount specialized hardware and execute functions on that hardware. In future work, we will extend the agent configuration to specify custom hardware and software capabilities and report this information to the *funcX* agent and service for scheduling.

To provide fault tolerance and robustness, for example with respect to node failures, the *funcX* agent relies on periodic heartbeat messages and a process to detect lost managers. The *funcX* agent tracks tasks that have been distributed to managers so that when failures do occur, lost tasks can be re-executed (if permitted). *funcX* agents communicate with the *funcX* service's forwarder via a ZeroMQ channel. Loss of a *funcX* agent is detected by the forwarder and when the *funcX* agent recovers, it repeats the registration process to acquire a new forwarder and continue receiving tasks. To reduce overheads, the *funcX* agent can shut down managers to release resources when they are not needed, suspend managers to prevent further tasks from being scheduled to them, and monitor resource capacity to aid scaling decisions.

Managers represent, and communicate on behalf of, the collective capacity of the workers on *a single node*, using just two sockets per node. Managers determine the available CPU and memory resources on a node, and partition the node among the workers. Managers advertise deployed container types and available capacity to the endpoint.

Workers persist on a node (optionally within containers) and each executes one task at a time. Since workers have a single responsibility, they use blocking communication to wait for tasks from the manager. Once a task is received, it is deserialized, executed, and the serialized results are returned via the manager.

4.4 Managing Compute Infrastructure

funcX is designed to support a range of computational resources, from embedded computers to clusters, clouds, and supercomputers, each with distinct access modes. As *funcX* workloads are often sporadic, resources must be provisioned and deprovisioned as needed to reduce costs due to idle resources. *funcX* uses Parsl's provider interface [26] to interact with various resources, specify resource-specific requirements (e.g., allocations, queues, limits, cloud instance types), and define rules for automatic scaling (i.e., limits and scaling aggressiveness). This interface allows *funcX* to be deployed on batch schedulers such as Slurm, PBS, Cobalt, SGE, and Condor; major cloud systems such as AWS, Azure, and Google Cloud; and Kubernetes.

4.5 Serialization

funcX supports the registration of arbitrary Python functions and the passing of data (e.g., primitive types and complex objects) to/from those functions. *funcX* uses a Facade interface with several serialization libraries (including pickle, dill, and JSON) as some Python object types cannot be serialized with some serialization libraries, and no single serialization library can serialize all objects. The *funcX* serializer sorts the serialization libraries by speed and applies them in order successively until the object is successfully serialized. This approach combines the strengths of various libraries, including support for complex objects (e.g., machine learning models) and traceback objects in a fast and transparent fashion. Once objects are serialized, they are packed into buffers with headers that include routing tags and the serialization method, such that only the buffers need to be unpacked and deserialized at the destination.

4.6 Batching

funcX supports two batching to amortize costs across many function requests: internal batching enables managers to request many tasks on behalf of their workers, minimizing network communication costs; and, user-facing *batching* that enables users to define batches of function inputs, allowing users to trade off efficient execution and increased per-function latency by creating fewer, larger requests. The SDK includes a matching batch interface for retrieving the results of many tasks concurrently.

4.7 Security Model

funcX requires a security model to ensure that functions are executed on endpoints by authenticated and authorized users and that one function cannot interfere with another.

Authentication and authorization: Since *funcX* endpoints may be deployed across arbitrary resources, we first summarize authentication and authorization requirements.

- Different research CI may rely on diverse identity management systems and authentication models (e.g., two-factor authentication). To ease the deployment of *funcX* agent on any resources, *funcX* needs a general model that provides a uniform API, rather than maintaining a set of APIs for the diverse identity providers.
- Users may have to use different accounts (e.g., institution accounts, national CI credentials, or national laboratory accounts) to access different resources. Users would like to use one account to authenticate *funcX* endpoints infrequently.
- One frequent use case in scientific computing is that resources are shared among a group of scientists. Ideally, the authorization model should enable users to grant access to others while enforcing secure delegation.

funcX uses Globus Auth [75] for identity and access management (IAM), and protection of all APIs. We use Globus Auth as it satisfies the above requirements, is widely adopted in scientific community, implements standard protocols (e.g., OAuth 2), enables simple delegation (e.g., such that a user may allow the *funcX* service or another user to access their endpoint), and offers a flexible OAuth client model for developing the *funcX* SDK. Although Globus Auth is used as the primary implementation, other IAM services that provide similar capabilities and interfaces could be integrated with *funcX*.

The *funcX* service is registered as a Globus Auth *resource server*, allowing users to authenticate using a supported identity (e.g., institution, Google, ORCID) and enabling various OAuth-based authentication flows (e.g., native client) for different scenarios. *funcX* has associated Globus Auth scopes (e.g., "urn:globus:auth:scope:funcx:register_function") via which other clients (e.g., applications and services) may obtain authorizations for programmatic access. *funcX* endpoints are themselves Globus Auth native clients, each dependent on the *funcX* scopes, which are used to securely connect to the *funcX* service. Endpoints require the administrator to authenticate prior to registration in order to acquire access tokens used for constructing API requests. The connection between the *funcX* service and endpoints is established using ZeroMQ. Communication addresses are sent as part of the registration process. Inbound traffic from endpoints to the cloud-hosted service is limited to known IP addresses.

Isolation: *funcX* function execution can be isolated in containers to ensure that functions cannot access data or devices outside their context. To enable fine-grained tracking of execution, we store execution request histories in the *funcX* service and in logs on *funcX* endpoints.

5 Data Management

Data management is essential for many applications: functions may interact with large and/or remote datasets, and tasks may use the outputs of other tasks as inputs. This section describes how data can be staged and managed between different *funcX* endpoints (*inter-endpoint*) and between different functions within an endpoint (*intra-endpoint*).

Listing 2: Inter-endpoint data transfer with Globus.

```
from funcx import GlobusFile

data = GlobusFile(globus_endpoint_id='65e...',
                  file_path='/~/file.txt')
task_id = fc.run(func_id, endpoint_id,
                 remote_data=data)
```

5.1 Inter-endpoint Data Transfers

To minimize operational costs and performance overheads we limit the size of data that can be passed through the *funcX* service to 10 MB. To enable functions to be seamlessly invoked with large data that may be located on remote computers, we require an out-of-band data transfer mechanism. We summarize the primary requirements as follows.

- Transfers can be managed programmatically by *funcX*.
- The transfer mechanism should be natively supported and approved by the administrations of research CI.
- Transfers should be optimized and provide high performance, endpoint-to-endpoint movement.
- The transfer mechanism should be interoperable with *funcX*'s authentication and authorization model (i.e., Globus Auth) to secure data transfers on behalf of users.
- The transfer mechanism should allow a user's functions to fetch data that is shared among a group.

We focus on wide area data management, rather than cloud storage, as data may be stored or generated in different locations (e.g., instruments, campus clusters, supercomputers) in many scientific use cases. Based on the requirements above, we integrate Globus transfer [31] to streamline inter-endpoint data transfers. Globus has several advantages that lead us to this choice: i) it is a research data management service that provides high-performance data transfers between arbitrary storage resources, such as supercomputers, laptops, and clouds; ii) it is widely deployed on research CI and used in scientific research; iii) data are transferred directly between the source and destination systems via the GridFTP [24] protocol; iv) it provides a Python SDK that allows a user's functions to fetch shared data.

To use Globus, the Globus Connect software must be installed on the storage system, this is often done by administrators installing Globus Connect Server on large clusters or it can be done individually in user-space using Globus Connect Personal. Storage systems are registered as a Globus endpoint with associated authentication mechanism in the Globus service. Each endpoint is given a unique endpoint identifier that is used when transferring data.

In this paper, we extend *funcX* to allow for references to Globus-accessible files to be passed as input/output to/from a function. Specifically, users must specify the Globus endpoint and the path to the file on that endpoint. When Globus-accessible files are passed to/from a *funcX*

function, *funcX* can automatically stage data either prior to, or after invocation of the function. An example of using Globus for inter-endpoint data transfer is shown in Listing 2.

We have found that Globus is well suited for our current use cases; however, other mechanisms (e.g., HTTP, FTP, and rsync) could also be used for inter-endpoint data transfers by augmenting functions to make direct data downloads or uploads. In future work we will extend the inter-endpoint transfer model in *funcX* to transparently support these mechanisms as we have done in Parsl.

5.2 Intra-endpoint Data Transfers

Modern applications may involve frequent fine-grained communications among functions executed on an individual endpoint (i.e., intra-endpoint data transfers). For example, distributed machine learning (ML) training may require that state be coordinated among all worker nodes; and MapReduce-style applications often involve a shuffle phase where every map task sends data to every reduce task.

Here we describe the advantages and disadvantages of potential intra-endpoint data management approaches.

- **A shared file system** that can be accessed by every worker on an endpoint. The effort to attach such storage to a *funcX* endpoint is minimal, as many clusters, clouds, and supercomputers provide built-in shared file system (sharedFS) or object storage. However, they often have high access cost, limited IO performance, and high latency when writing and reading many files.
- **MPI** is a message passing fabric that is highly scalable and optimized for data communications on supercomputers with specialized interconnects; however, MPI libraries are not natively available or optimized for many computers (e.g., clouds and private clusters). More importantly, the synchronous nature of MPI's collective communication is not well-suited for the asynchronous task-based model in *funcX*, as it blocks tasks from making progress even when partial results are ready, which is important for many performance-driven asynchronous applications (e.g., distributed machine learning training); HPC containers often must be adapted to make use of local MPI libraries; and a failure of one MPI process may cause other MPI processes to block, which stops other tasks from continuing. We note that fault tolerance has improved in the recent release of MPI 4.0; however, this is not commonly deployed at the time of writing.
- **Socket and socket-like connections** (e.g., ZeroMQ) between workers can provide high throughput and low latency direct data transfers. However, creating pairwise connections between workers is expensive and in some cases workers (e.g., in containers) may not be network addressable or may not have sufficient open ports to support connections between all workers.
- **In-memory data stores** (e.g., MemCache [61] and Redis [21]) provide higher IOPS and lower latency than shared file systems and support more data types than socket connections (e.g., serialized data). However, they require that storage be provisioned explicitly, that additional services be hosted, and they cannot match the raw throughput or latency of direct socket connection [55].

Listing 3: Intra-endpoint data transfer with Redis.

```
def example(key, data):
  from funcx_endpoint import get_redis_client
  rc = get_redis_client()
  rc.set(key, data)
  rc.get(key)
```

The aforementioned advantages and disadvantages lead us to select the shared file system and in-memory data store (Redis) approaches to support intra-endpoint data transfers in *funcX*, as these approaches are both general and are readily available (or can be deployed with minimal effort) on most target resources. We present a preliminary performance study of these four approaches in §7.3 and the results show that the performance of shared file system and Redis is similar to the other approaches, especially when transferring large volumes of data. We have extended the *funcX* agent such that users may specify a requirement for a Redis cluster to be deployed alongside their endpoint. The *funcX* SDK provides a general interface to retrieve the Redis client which users can interact with, as shown in Listing 3.

6 Container Management

funcX uses containers to provide customized execution environments for functions irrespective of the endpoint's host environment. In this section, we discuss how the *funcX* agent spawns containers to serve functions, retains warm containers, routes functions to containers, and scales resources based on function requirements.

6.1 Container Warming

Commercial FaaS platforms [79] keep function containers *warm* by leaving them running for a short period of time (e.g., 5-15 minutes) following the execution of a function. Warm containers remove the need to instantiate a new container to execute a function, significantly reducing latency.

We argue that this need is especially important in HPC environments for several reasons. First, containers and Python environments (e.g., conda) are generally stored on shared file systems of HPC systems. Therefore, starting many containers and Python environments concurrently for the workers at the HPC scale may impose significant stress on the shared file systems. Second, many HPC centers implement their own methods for instantiating containers that place limitations on the number of concurrent requests. Third, individual cores are often slower in many-core architectures like Xeon Phis. As a result, the start time for containers can be much larger than what would be seen on a PC, as shown in TABLE 3 in §7.4.

In *funcX*, container warming is implemented by the *funcX* agent. To reduce the number of container cold starts, the *funcX* agent keeps a container warm until there are insufficient resources available to process pending workloads or the container has been idle for a configurable period of time (e.g., 10 minutes). The *funcX* agent is extensible to support other container-warming strategies, such as releasing the least recently used container and application-agnostic container warming [68] if necessary.

6.2 Warming-aware Function Routing

Ideally we aim to minimize the number of container cold starts due to the cost of starting a container in HPC environments. To do so, the *funcX* agent needs to know which computing nodes have warm containers and what types of warm containers, so that it can route the function tasks to the appropriate warm containers.

The *funcX* agent employs a hierarchical, warming-aware scheduling algorithm to route function tasks to workers to optimize throughput. The *funcX* agent determines which functions to route to a given manager, and each *manager* determines how to launch and spawn containers to satisfy the arriving workload. Thus, warming-aware routing involves coordination between managers and *funcX* agent. Each manager advertises its deployed container types and its available resources to the *funcX* agent. Based on the advertised information of each manager, the *funcX* agent implements a warming-aware scheduling algorithm to route tasks to managers. Specifically, when receiving a task with requirement for a specific container type, the scheduler attempts to send the task to a manager that has a suitable warm container. When there are multiple available managers with the required container type warm, priority is given to the one with the most available container workers to balance load across managers. If there are not any warmed containers on any connected managers, the *funcX* agent chooses one manager at random to execute the task. While we use random scheduling in our implementation, other scheduling policies, such as bin-packing and round-robin, could also be used. To amortize network latency during manager advertising and task dispatching, the *funcX* agent also supports *prefetching*, which allows a manager to prefetch a configurable number of additional tasks beyond its current availability.

Upon receiving a set of tasks, the manager determines the required container types and dynamically starts (and stops) containers to serve tasks in a fair manner: we set the number of deployed containers for a function type to be proportional to the number of received tasks of this function type. For instance, if 30% of the tasks a manager receives are of type A and the manager can spawn at most 10 containers, the manager will spawn 3 containers of type A.

It is worth mentioning that the function routing is different when an endpoint is deployed on a Kubernetes cluster. Both the manager and its workers are deployed within a container pod that can only serve one type of function. Hence, in this case, each manager is deployed with a specific container image and the agent simply needs to route tasks to corresponding managers.

We apply relatively simple scheduling algorithms here to demonstrate the benefits of warming aware routing; however, the *funcX* agent implements modular scheduling interfaces for function routing (at *funcX* agents) and container deployment (by managers) which enabling different algorithms (e.g., priority-aware or deadline-driven scheduling) to be implemented by users. We note that when task duration is much larger than the container cold start time, the benefits of warming-aware routing are limited.

6.3 Elastic Resource Provisioning

One of the main benefits of the FaaS computing model is elasticity. To provide elasticity on a *funcX* endpoint, a *funcX* agent dynamically provisions resources via an extensible provisioning *strategy* interface.

The strategy interface consists of a monitoring and a scaling component within the *funcX* agent. The monitoring component interacts periodically (e.g., every second) with the provider interface (introduced in §4.4) and the *funcX* agent to fetch the current endpoint load, including the active and idle resources (i.e., the number of container workers) and the number of pending function requests. Based on the monitoring information, the scaling component automatically provisions more resources when the number of function requests is greater than the number of idle resources, and releases resources that have been idle for some period of time, via the provider interface. The maximum idle time is set to two minutes by default, but is user-configurable for each endpoint.

Similar to commercial FaaS platforms such as AWS Lambda and Azure Functions, the *funcX* strategy allows users to configure the minimum and maximum resources to be used, as well as how aggressively a *funcX* agent scales those resources (e.g., request one more resource when there are ten waiting requests). However, elasticity may be subject to resource request delays, such as the time to request a new instance on a cloud or to provision a resource via an HPC scheduler.

7 Evaluation

We evaluate the performance of *funcX* in terms of latency, scalability, and throughput. We also study the effects of batching, function routing, and data transfer approaches.

7.1 Latency

We explore *funcX* latency by instrumenting the system. Figure 3 shows latencies for a warm container as follows: t_s: Web service latency to authenticate, store the task in Redis, and append the task to an endpoint's queue; t_f: forwarder latency to read task from the Redis store, forward the task to an endpoint, and write the result to the Redis store; t_e: endpoint latency to receive tasks and send results to the forwarder, and to send tasks and receive results from the worker; and t_w: function execution time. The endpoint was deployed on ANL's Cooley cluster for this test and had an 18 ms latency on average to the forwarder. We observe that t_w is fast relative to the overall system latency. The network latency between service and forwarder includes minimal communication time due to internal AWS networks (measured at $<$1 ms). Most *funcX* overhead is found in t_s due to authentication, and in t_e due to internal queuing and dispatching. We note here that the aim of *funcX* is not to build yet another low-latency FaaS platform, but instead to provide a new federated model in which functions can be executed on arbitrary remote machines. Nevertheless, in our previous work we showed that the latency of *funcX* is comparable to commercial FaaS platforms, such as AWS Lambda, Google Cloud Functions, and Azure Functions [32].

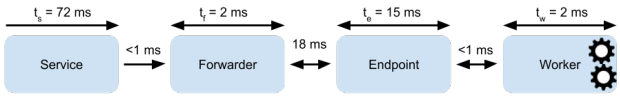

Fig. 3: *funcX* latency breakdown for a container.

7.2 Scalability and Throughput

We study the strong and weak scaling of the *funcX* agent on ANL's Theta and NERSC's Cori supercomputers. Theta is a 11.69-petaflop system based on the second-generation Intel Xeon Phi "Knights Landing" (KNL) processor. Its 4392 nodes each have a 64-core processor with 16 GB MCDRAM, 192 GB of DDR4 RAM, and are interconnected with high speed InfiniBand. Cori is a 30-petaflop system with an Intel Xeon "Haswell" partition and an Intel Xeon Phi KNL partition. We ran our tests on the KNL partition, which has 9688 nodes, each with a 68-core processor (with 272 hardware threads) with six 16-GB DIMMs, 96 GB DDR4 RAM, interconnected in a Dragonfly topology. We perform experiments using 64 Singularity containers on each Theta node and 256 Shifter containers on each Cori node. Due to a limited allocation on Cori we use the four hardware threads per core to deploy more containers than cores.

Strong scaling evaluates performance when the total number of function invocations is fixed; weak scaling evaluates performance when the average number of functions executed on each container is fixed. To measure scalability we created functions of various durations: a 0-second "no-op" function that exits immediately, a 1-second "sleep" function, and a 1-minute CPU "stress" function that keeps a CPU core at 100% utilization. For each case, we measured completion time of a batch of functions as we increased the total number of containers. Notice that the completion time of running M "no-op" functions on N workers indicates the overhead of *funcX* to distribute the M functions to N containers. Due to limited allocations we did not execute sleep or stress functions on Cori, nor did we execute stress functions for strong scaling on Theta. We pre-warmed all containers in these experiments.

7.2.1 Strong scaling

Figure 4(a) shows the completion time of 100 000 *concurrent* function requests with an increasing number of containers. On both Theta and Cori, the completion time decreases as the number of containers increases, until we reach 256 containers for "no-op" and 2048 containers for "sleep" on Theta. As reported by Wang et al. [79] and Microsoft [19], Amazon Lambda achieves good scalability for a single function to more than 200 containers, Microsoft Azure Functions can scale up to 200 containers, and Google Cloud Functions does not scale well beyond 100 containers. While these results do not indicate the maximum number of containers that can be used for a single function, and likely include per-user limits imposed by the platform, our results show that *funcX* scales similarly to commercial platforms.

7.2.2 Weak scaling

We performed *concurrent* function requests such that each container receives, on average, 10 requests. Figure 4(b) shows weak scaling for "no-op," "sleep," and "stress." For "no-op," the completion time increases with more containers on both Theta and Cori. This reflects the time required to distribute requests to all of the containers. On Cori, *funcX* scales to 131 072 concurrent containers and executes more than 1.3 million "no-op" functions. Again, we see that the completion time for "sleep" remains close to constant up to 2048 containers, and the completion time for "stress" remains close to constant up to 16 384 containers. Thus, we expect a function with a several-minute duration would scale well to many more containers.

7.2.3 Throughput

We observe a maximum throughput for a *funcX* agent (computed as number of function requests divided by completion time) of 1694 and 1466 requests per second on Theta and Cori, respectively.

7.2.4 Summary

Our results show that *funcX* agents i) scale to 130 000+ containers for a single function; ii) exhibit good scaling performance up to approximately 2048 containers for a 1-second function and 16 384 containers for a 1-minute function; and iii) provide similar scalability and throughput using both Singularity and Shifter containers on Theta and Cori. It is important to note that these experiments study the *funcX* agent, and not the end-to-end throughput of *funcX*. While the *funcX* web service can elastically scale to meet demand, communication overhead may limit throughput. To address this challenge and amortize communication overheads, we enable batch submission of tasks. These optimizations are discussed in §7.5.

7.3 Data Management

We evaluate four potential approaches for intra-endpoint data transfers (described in §5.2) on Theta. We use mpi4py for MPI, ZeroMQ for direct socket connections, Redis for the in-memory store, and Theta's Lustre shared file system. We note that we use mpi4py because it supports direct Python object transfers and previous work [65] has shown that mpi4py does not add significant overhead when compared with OpenMPI in terms of throughput and latency for data transfers. We emulate different communication patterns (i.e., point-to-point, broadcast, and all-to-all) and vary data transfer size. Fig. 5 shows the performance of these four approaches with different communication patterns. As expected, MPI performs the best, and sharedFS the worst. However, ZeroMQ and Redis achieve similar performance to MPI. As data volume increases, the performance difference between the four approaches diminishes, as transfer time is mainly determined by available network bandwidth (which is the same for all approaches).

While sharedFS and Redis perform slightly worse than MPI for small data sizes, we adopt them in *funcX* because of their generality, ease-of-use, and the challenges of using mpi4py (as well as MPI compiled in C) and ZeroMQ described in §5.2.

We now evaluate intra-endpoint data management in the context of MapReduce applications and a real-world science application deployed on funcX.

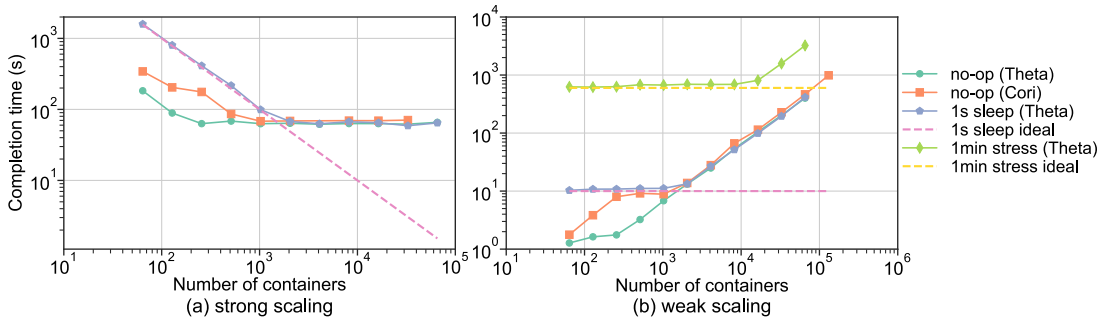

Fig. 4: Strong and weak scaling of the *funcX* agent.

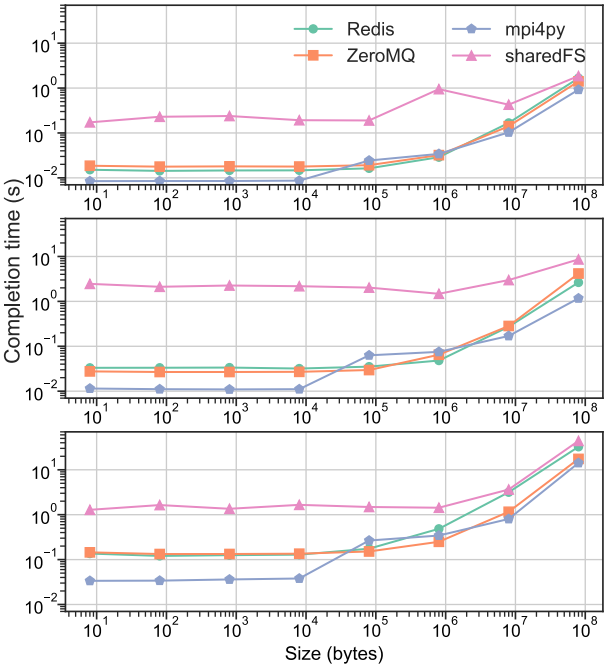

Fig. 5: Performance of the four intra-endpoint transfer approaches. Top: point-to-point; middle: broadcast to 20 nodes; bottom: all-to-all on 20 nodes.

7.3.1 MapReduce

To demonstrate how Redis and sharedFS can facilitate intra-endpoint data transfers for real applications, we deployed a *funcX* endpoint with a three-node Redis cluster. We also used the shared Lustre file system on Theta. We deployed two MapReduce applications: WordCount and Sort. These applications involve an all-to-all communication pattern between map and reduce tasks (i.e., data shuffling).

Each application processes 30 GB of Wikipedia text data, and has 300 map and 300 reduce tasks, requiring communication of 90000 data chunks in total. TABLE 1 shows the average completion time of each task spent in each phase of the MapReduce application: input read, map process, intermediate write, intermediate read, reduce process, output write, when using Redis and sharedFS approaches for data shuffling. WordCount benefits less from Redis than Sort as WordCount shuffles just one tenth of the data. The table shows that Redis can speed up the data shuffling phase of the workload (i.e., intermediate write and read) by up to 3x.

Note that TABLE 1 shows the average task completion time. The benefits of Redis over sharedFS on the total completion time of a MapReduce application may depend on the amount of parallelism and the portion of the shuffle phase over the other phases. For example, the total completion time of Sort with Redis and sharedFS are 220 and 520 seconds, respectively, yielding a 55.7% improvement. The WordCount application runs in 1800 seconds and 2200 seconds, respectively, yielding a 18.2% improvement. This is because Sort has a heavier shuffle phase than WordCount.

TABLE 1: Average completion time of the transfer phases in WordCount and Sort when using Redis and shared file system for intra-endpoint data management.

	WordCount (s)		Sort (s)	
	Redis	**SharedFS**	**Redis**	**SharedFS**
Intermediate write	3.55	8.15	3.27	5.32
Intermediate read	33.39	43.40	11.37	41.77

7.3.2 Colmena

Finally, we evaluate intra-endpoint data management in the context of a real-world scientific application to demonstrate the benefits of Redis over sharedFS. Colmena [80] is a framework that manages large-scale, AI-directed steering of computational campaigns (e.g., to efficiently explore large molecular spaces when designing new materials). A Colmena application consists of a *Thinker* that implements the decision-making policy to generate new tasks (e.g., new simulation, new model training, or model inference), a *Task Server* that dispatches task requests to resources and manages task results, and *Workers* that are deployed on compute resources to execute tasks. These components exchange data (e.g., task requests and results) with Redis used to facilitate transfers. We implement a Colmena benchmark with 1000 tasks, each with 1 MB input and 1 MB output data. Table 2 shows the average completion time of the communication stages in Colmena. Redis yields a lower completion time for all communication stages compared to sharedFS. Such a benefit has been demonstrated to be particularly important when running Colmena at scale with thousands of tasks.

TABLE 2: Average completion time of the transfer phases in the Colmena benchmark when using Redis and shared file system for intra-endpoint data management.

Stage	**Redis (ms)**	**SharedFS (ms)**
Input data write from Thinker	7.15	32.31
Input read on Workers	0.70	11.36
Result write from Workers	18.04	244.72
Result read from Task Server	0.11	3.50

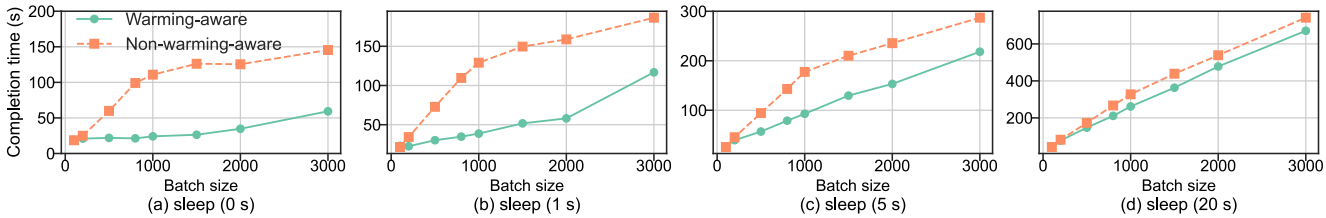

Fig. 6: Completion time of warming-aware and non-warming-aware routing.

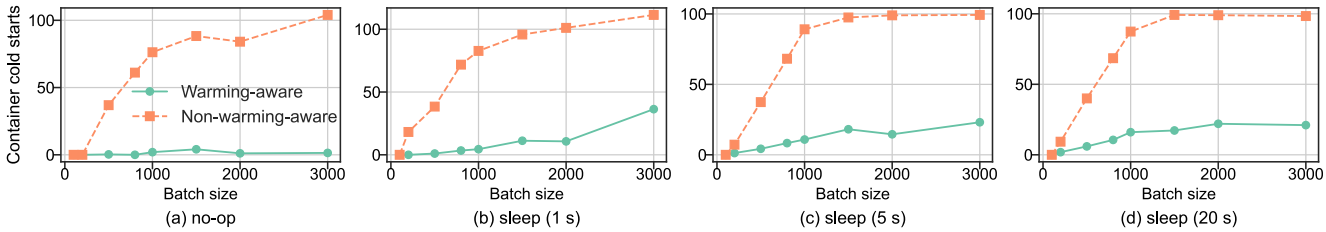

Fig. 7: Number of container cold starts of warming-aware and non-warming-aware routing.

7.4 Function Routing

Before exploring function routing performance, we first quantify the instantiation cost of various container technologies on different resources. Specifically, we measure the time taken to start a container and execute a Python command to import *funcX*'s worker module—a requirement prior to executing a *funcX* function. We deploy the containers on an AWS EC2 `m5.large` instance and on compute nodes on Theta and Cori following the facility's documented best practices. TABLE 3 shows the results. We speculate that the significant performance deterioration of container instantiation on HPC systems can be attributed to a combination of slower clock speed on KNL nodes and shared file system contention when fetching images. These results highlight the need to apply function warming approaches to reduce overheads on HPC systems.

TABLE 3: Cold container instantiation time for different container technologies on different resources.

System	Container	Min (s)	Max (s)	Mean (s)
Theta	Singularity	9.83	14.06	10.40
Cori	Shifter	7.25	31.26	8.49
EC2	Docker	1.74	1.88	1.79
EC2	Singularity	1.19	1.26	1.22

We evaluate *funcX*'s function routing strategy and show that it improves overall throughput as well as reducing the number of container cold starts. We deployed an endpoint on Theta and compared the performance of warming-aware routing and randomized (non-warming-aware) routing. The endpoint is allocated 10 nodes and each node can host 10 workers, each with its own container. We registered 10 functions, where each function requires a specific container (i.e., 10 different containers.) We submitted a batch of requests, each of which is chosen from one of the ten functions uniformly at random. Fig. 6 and Fig. 7 show the overall function completion time and the number of container cold starts for different batch sizes and for different function durations (0, 1, 5, and 20 seconds). We note that the number of requests in a batch is much higher than the available resources (100 container workers) in this experiment, and thus a container worker is more likely to be killed to serve other request when using non-warming-aware routing. Thus, the warming-aware routing reduces completion time by up to 61% for a batch of requests (i.e., higher throughput) and reduces the number of container cold starts significantly (e.g., 22 cold starts for 3000 functions), compared to the randomized routing strategy. This is because the warming-aware algorithm attempts to reuse the warm containers as much as possible to reduce the overhead of container instantiation. As expected, the benefit of warming-aware routing gradually diminishes as the function duration increases, because the function runtime, rather than the cold container instantiation time, becomes dominant.

7.5 Batching

To evaluate the effect of executor-side batching, we submit 10 000 concurrent "no-op" function requests and measure the completion time when executors can request one function at a time (batching disabled) vs when they can request many functions at a time based on the number of idle containers (batching enabled). We use 4 nodes (64 containers each) on Theta. We observe that the completion time with batching enabled is 6.7 s (compared to 118s when disabled).

8 Experiences with *funcX*

As of August, 2022 *funcX* has been used by 413 users to perform over 19.8 million function invocations, 338 105 functions have been registered, and 4027 endpoints have been created. Here we describe our experiences applying *funcX* to various scientific case studies.

AI-enabled steering of computational campaigns: Colmena [80] is an open-source library that enables researchers to build complex, AI-directed HPC campaigns. Researchers can implement flexible decision-making policies to steer different tasks (e.g., simulation, model update, and model inferences) of computational campaigns. When tasks are generated, *funcX* serves as an execution backend to distribute and execute tasks. The FaaS model of *funcX* and the implementation of container management allows Colmena to flexibly dispatch tasks to arbitrary computing resources, enabling ML-enhanced tasks to be sent to GPU-accelerated devices and high throughput simulations to HPC clusters. The integration of data management mechanisms (e.g.,

Globus and Redis) in *funcX* enables data to move between Colmena entities transparently without requiring the user to manage movement; further, it can improve performance and simplify distributed, data-intensive campaigns.

Linking instruments and HPC: *funcX* has been used to combine several experimental instruments with HPC infrastructure [78]. This approach allows scientists to offload computationally-intensive analysis tasks to HPC resources, simplifies large-scale parallel processing for large data rates, and enables online analysis. Such experimental techniques, including serial synchrotron crystallography [70], X-ray photon correlation spectroscopy [62], ptychography [30], and scientific machine learning [57], depend on orchestration of various activities in various locations. For this purpose, these examples use Globus flows [33] to create complex sequences of actions. For example, when data are acquired from an experiment, run quality control at the edge, move data to an HPC center, run analysis and reconstruction algorithms, and index resulting images in a data catalog. *funcX* provides the compute substrate enabling many of these actions to be executed in various locations. Integration of Globus for data management simplifies dispatching tasks to different resources without requiring changes to broader workflows to transfer and retrieve inputs and results. Further, scientific analysis toolkits are often containerized to promote portability and exploit available resources. The container warming features presented in this paper enable these workloads to reduce cold starts, which can be costly on large, shared file systems and facilitate rapid computation—a necessary feature to support real-time computation and experiment steering. We report function execution and data transfer characteristics in prior work [78].

AlphaFold as a Service: AlphaFold [47] is a cutting edge machine learning model that can predict a protein's 3D structure from its amino acid sequence. AlphaFold has garnered significant interest in the bioinformatics community, with applications in the development of therapeutics and accelerating the practice of deriving crystal structures at light sources. However, AlphaFold relies on powerful GPUs and large reference datasets, restricting access for many researchers. To address these challenges, the Argonne Leadership Computing Facility deployed AlphaFold as a service using *funcX* to dynamically provision GPU resources on-demand. In this work, containers are dispatched to GPU nodes and managed by the *funcX* endpoint to serve inference requests. As AlphaFold tasks can take over an hour to process, the Globus integration with *funcX* provides the ability to asynchronously transfer results to users.

Distributed ML: Flox [51] is a federated learning (FL) framework that decouples model training and inference from infrastructure management. Flox uses *funcX* to enable users to train and deploy FL models on one or more remote computers, and in particular on edge devices. We are applying these techniques to Rural AI applications [25], using *funcX* to facilitate training and deployment of models in remote locations. Rural AI requires reliable task and result transmission as devices are deployed in rural settings where device and network outages are common, and the quality of wireless networks varies depending on location. *funcX*'s hierarchically designed queues support the necessary robustness to dispatch (and queue) tasks across rural devices. Containers enable execution of tasks on heterogeneous edge devices for training and on centralized cloud instances for model aggregation.

9 Related Work

Both commercial and open-source FaaS platforms have proved extremely successful in industry as a way to reduce costs and remove the need to manage infrastructure.

Hosted FaaS platforms: *Amazon Lambda* [1], *Google Cloud Functions* [11], and *Azure Functions* [8] are the most well-known FaaS platforms. They support various function languages and trigger sources, connect directly to other cloud services, and apply fine-grain billing models. Lambda uses Firecracker [22], a custom virtualization technology built on KVM, to create lightweight micro-virtual machines. To meet the needs of IoT use cases, some cloud-hosted platforms support local deployment (e.g., AWS Greengrass [7]); however, they support only single machines and require that functions be exported from the cloud platform.

Open source platforms: Open FaaS platforms resolve two of the key challenges to using FaaS for scientific workloads: they can be deployed on-premise and can be customized to meet the requirements of data-intensive workloads without set pricing models.

Apache OpenWhisk [4], the basis of IBM Cloud Functions [12], defines an event-based programming model, consisting of *Actions* which are stateless, runnable functions, *Triggers* which are the types of events OpenWhisk may track, and *Rules* which associate one trigger with one action. OpenWhisk can be deployed locally as a service using a Kubernetes cluster.

Fn [10] is an event-driven FaaS system that executes functions in Docker containers. Fn allows users to logically group functions into applications. Fn can be deployed locally (on Windows, MacOS, or Linux) or on Kubernetes.

The *Kubeless* [15] FaaS platform builds upon Kubernetes. It uses Apache Kafka for messaging, provides a CLI that mirrors Amazon Lambda, and supports comprehensive monitoring. Like Fn, Kubeless allows users to define function groups that share resources.

SAND [23], which has been recently open-sourced as KNIX MicroFunctions [14], is a lightweight, low-latency FaaS platform from Nokia Bell Labs that provides application-level sandboxing and a light-weight process-based execution model. KNIX provides support for function chaining via user-submitted workflows. Recently, KNIX has been further extended to support GPU sharing among functions [64]. However, KNIX requires privileged access to nodes, which is generally not possible in research CI.

Abaco [74] implements the Actor model, where an *actor* is an Abaco runtime mapped to a specific Docker image. Each actor executes in response to messages posted to its *inbox*. It supports functions written in several programming languages and automatic scaling. Abaco also provides fine-grained monitoring of container, state, and execution events and statistics. Abaco is deployable via Docker Compose.

ChainFaaS [42] is a blockchain-based FaaS platform that makes use of idle personal computers. The platform allows users to submit functions that utilize contributed computing

power, or to be a provider who contributes the idle computing resources for potential profits. While ChainFaaS shares some similar goals with funcX, it focuses on deployment on personal computers, rather than large-scale research CI.

DFaaS [34] is a federated and decentralized FaaS platform for edge computing. It relies on a peer-to-peer network to share the states of edge nodes to balance loads among all the nodes.

Comparison with *funcX*: Hosted cloud providers implement high performance and reliable FaaS models that are used by an enormous number of users. However, they often have vendor lock-in, are not designed to support heterogeneous resources or research CI (e.g., schedulers, containers), do not integrate with the science ecosystem (e.g., in terms of data and authentication models), and can be costly.

Open source and academic frameworks support on-premise deployments and can be configured to address a range of use cases. However, most systems we surveyed are Docker-based and rely on Kubernetes (or other container orchestration platforms) for deployment. Some systems such as ChainFaaS and DFaaS support distributed function execution on personal computers and edge nodes. However, to the best of our knowledge, there are no systems that support remote execution over a federated ecosystem of endpoints on diverse research CI (from edge to HPC environments).

Other Related Approaches: FaaS has many predecessors, notably grid and cloud computing, container orchestration, and analysis systems. Grid computing [39] laid the foundation for remote, federated computations, often through federated batch submission [52]. GridRPC [66] defines an API for executing functions on remote servers requiring that developers implement the client and the server code. *funcX* extends these ideas to allow interpreted functions to be registered and then executed within sandboxed containers via standard cloud and endpoint APIs.

Container orchestration systems, such as Mesos [44], Kubernetes [43], KubeFed [16], MicroK8s [18], and K3s [13], allow users to scale deployment of containers while managing scheduling, fault tolerance, resource provisioning, and addressing other user requirements. Mesos and Kubernetes primarily rely on dedicated, cloud-native infrastructure. KubeFed extends Kubernetes to support multi-cluster deployments. MicroK8s and K3s are lightweight versions of Kubernetes and are designed for Edge and IoT use cases. These systems cannot be directly used with diverse research CI (e.g., HPC resources); however, these container orchestration systems serve as a basis for developing serverless platforms, such as Kubeless, and indeed play an increasingly important role in research CI. *funcX* focuses at the level of scheduling and managing functions, that are deployed across a pool of containers. We leverage both container orchestration systems (e.g., Kubernetes) as well as techniques from orchestration systems (e.g., warming) in funcX.

Data-parallel systems such as Hadoop [2] and Spark [6] enable map-reduce style analyses. Unlike *funcX*, these systems dictate a particular programming model on dedicated clusters. Python parallel computing libraries such as Parsl and Dask [9] support development of parallel programs, and parallel execution of selected functions within those scripts, on clusters and clouds. These systems could be extended to use *funcX* for remote execution of tasks.

LFM [67] provides advanced dependency management for Python functions by using transparent dependency detection and distribution, and dynamic provisioning and resource management at the granularity of a Python function. Azure Functions [68] proposed a policy that dynamically controls the pre-warming window for application containers to reduce the number of container cold starts, based on the characterization of applications. Researchers have proposed various methods to mitigate container cold start latency by leveraging various workflow-specific information, such as cascading starts and dependency graphs [29], [35], [69]. Anna [73] is an autoscaling key-value store that can be used to support stateful serverless computing. Delta [53] adds a shim layer on top of *funcX* that profiles the function performance on different endpoints and automatically schedules functions to appropriate endpoints. Several recent papers have aimed to model application performance and optimize performance on FaaS platforms [28], [45], [50], [56]. While *funcX* implements its own function routing, container management, data management schemes, and performance metrics, these systems are orthogonal to this paper and could be integrated with *funcX*.

Several frameworks have been implemented on top of *funcX* to create workflows for different scientific use cases. For instance, Xtract [71] uses *funcX* to enable workflow compositions for distributed bulk metadata extraction. Globus Automate [78] uses *funcX* to run arbitrary computations as part of automated and event-based workflows, it uses *funcX*'s APIs to automatically monitor the status of a *funcX* function and trigger the next step when it completes.

10 Conclusion

funcX is a distributed FaaS platform that is designed to support the unique needs of research computing. Unlike existing centralized FaaS platforms, *funcX* combines a reliable and easy-to-use cloud-hosted interface with the ability to securely execute functions on user-deployed *funcX* endpoints deployed on various remote computing resources. *funcX* supports many HPC systems and cloud platforms, can use three container technologies, and can expose access to heterogeneous and specialized computing resources. In this paper we extend *funcX* to support inter-endpoint and intra-endpoint data transfers between functions, and optimize function execution performance with advanced container management and warming-aware function routing mechanisms. We showed that *funcX* agents can scale to execute 1M tasks over 130 000 concurrent workers when deployed on the Cori supercomputer. We also showed that *funcX*'s data transfer mechanisms are comparable to alternative methods, and that they can significantly improve application performance. Finally, we showed that *funcX* can dynamically route functions to workers to reduce container warming overhead and that batching can significantly reduce overheads.

funcX demonstrates the advantages of adapting the FaaS model to create a federated computing ecosystem. Based on early experiences using *funcX* in scientific case studies [32], we have found that the approach provides several advantages, including abstraction, code simplification, portability, scalability, and sharing; however, we also identified several

limitations including suitability for some applications, conflict with current allocation models, and challenges decomposing applications into functions. We hope that *funcX* will serve as a flexible platform for research computing while also enabling new studies in function scheduling, dynamic container management, and data management.

In future work, we will continue our work to explore new scheduling approaches that can select appropriate endpoints for function execution and manage data dependencies between functions. We also plan to provide APIs that allow users to manage and discover functions and endpoints. We will extend *funcX*'s container management capabilities to create containers dynamically based on function requirements, and to stage containers to endpoints on-demand. We will also explore techniques for optimizing performance, for example by sharing containers among functions with similar dependencies and developing resource-aware scheduling algorithms.

Acknowledgment

This work was supported in part by NSF 2004894/2004932 and Laboratory Directed Research and Development funding from Argonne National Laboratory under U.S. Department of Energy under Contract DE-AC02-06CH11357. This work also used resources of the Argonne Leadership Computing Facility and Center for Computational Science and Engineering at Southern University of Science and Technology.

References

[1] Amazon Lambda. https://aws.amazon.com/lambda. Accessed May 1, 2022.
[2] Apache Hadoop. https://hadoop.apache.org/. Accessed May 1, 2022.
[3] Apache Kafka. https://kafka.apache.org/. Accessed May 1, 2022.
[4] Apache OpenWhisk. http://openwhisk.apache.org/. Accessed May 1, 2022.
[5] Apache Pulsar. https://pulsar.apache.org/. Accessed May 1, 2022.
[6] Apache Spark. https://spark.apache.org/. Accessed May 1, 2022.
[7] AWS Greengrass. https://aws.amazon.com/greengrass/. Accessed May 1, 2022.
[8] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed May 1, 2022.
[9] Dask. http://docs.dask.org/en/latest/. Accessed May 1, 2022.
[10] Fn project. https://fnproject.io. Accessed May 1, 2022.
[11] Google Cloud Functions. https://cloud.google.com/functions/. Accessed May 1, 2022.
[12] IBM Cloud Functions. https://www.ibm.com/cloud/functions. Accessed May 1, 2022.
[13] K3s. https://k3s.io/. Accessed May 1, 2022.
[14] KNIX MicroFunctions. https://github.com/knix-microfunctions/knix. Accessed May 1, 2022.
[15] Kubeless. https://kubeless.io. Accessed May 1, 2022.
[16] Kubernetes Cluster Federation. https://github.com/kubernetes-sigs/kubefed. Accessed May 1, 2022.
[17] Linux containers. https://linuxcontainers.org. Accessed May 1, 2022.
[18] MicroK8s. https://microk8s.io/. Accessed May 1, 2022.
[19] Microsoft Azure Functions Documentation. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale. Accessed May 1, 2022.
[20] Rabbitmq. https://www.rabbitmq.com/. Accessed May 1, 2022.
[21] Redis. https://redis.io/. Accessed May 1, 2022.
[22] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.
[23] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards high-performance serverless computing. In *USENIX Annual Technical Conference*, pages 923–935, 2018.
[24] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The globus striped gridftp framework and server. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 54, USA, 2005. IEEE Computer Society.
[25] O. Almurshed, P. Patros, V. Huang, M. Mayo, M. Ooi, R. Chard, K. Chard, O. Rana, H. Nagra, M. Baughman, and I. Foster. Adaptive edge-cloud environments for rural ai. In *2022 IEEE International Conference on Services Computing (SCC)*, pages 74–83, Los Alamitos, CA, USA, jul 2022. IEEE Computer Society.
[26] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, and et al. Parsl: Pervasive parallel programming in python. In *28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'19, pages 25–36. ACM, 2019.
[27] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.
[28] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen. Performance modeling and workflow scheduling of microservice-based applications in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2114–2129, 2019.
[29] D. Bermbach, A.-S. Karakaya, and S. Buchholz. Using application knowledge to reduce cold starts in faas services. In *Proceedings of the 35th annual ACM symposium on applied computing*, pages 134–143, 2020.
[30] T. Bicer, X. Yu, D. J. Ching, R. Chard, M. J. Cherukara, B. Nicolae, R. Kettimuthu, and I. T. Foster. High-performance ptychographic reconstruction with federated facilities. In *Smoky Mountains Computational Sciences and Engineering Conference*, pages 173–189. Springer, 2021.
[31] K. Chard, S. Tuecke, and I. Foster. Efficient and secure transfer, synchronization, and sharing of big data. *IEEE Cloud Computing*, 1(3):46–55, 2014.
[32] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, page 65–76, 2020.
[33] R. Chard, J. Pruyne, K. McKee, J. Bryan, B. Raumann, R. Ananthakrishnan, K. Chard, and I. Foster. Globus automation services: Research process automation across the space-time continuum. *arXiv preprint arXiv:2208.09513*, 2022.
[34] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo. Dfaas: Decentralized function-as-a-service for federated edge computing. In *IEEE 10th International Conference on Cloud Networking (CloudNet)*, pages 1–4. IEEE, 2021.
[35] N. Daw, U. Bellur, and P. Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, pages 356–370, 2020.
[36] J. Forde, T. Head, C. Holdgraf, Y. Panda, G. Nalvarete, B. Ragan-Kelley, and E. Sundell. Reproducible research environments with repo2docker. In *Workshop on Reproducibility in Machine Learning*, 2018.
[37] I. Foster. Globus Online: Accelerating and democratizing science through cloud-based services. *IEEE Internet Computing*, 15(3):70, 2011.
[38] I. Foster and D. B. Gannon. *Cloud Computing for Science and Engineering*. MIT Press, 2017.
[39] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Intl Journal of Supercomputer Applications*, 15(3):200–222, 2001.
[40] G. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. Status of serverless computing and function-as-a-service (FaaS) in industry and research. *arXiv preprint arXiv:1708.08028*, 2017.
[41] G. Fox and S. Jha. Conceptualizing a computing platform for science beyond 2020. In *IEEE 10th International Conference on Cloud Computing*, pages 808–810, 2017.
[42] S. Ghaemi, H. Khazaei, and P. Musilek. Chainfaas: An open blockchain-based serverless platform. *IEEE Access*, 8:131760–131778, 2020.
[43] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and running*

dive into the future of infrastructure. O'Reilly Media, Inc., 1st edition, 2017.

[44] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Conf. on Networked Sys. Design and Impl.*, pages 295–308, 2011.

[45] M. R. HoseinyFarahabady, A. Y. Zomaya, and Z. Tari. A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform. *IEEE Transactions on Parallel and Distributed Systems*, 29(7):1442–1455, 2018.

[46] D. M. Jacobsen and R. S. Canon. Contain this, unleashing Docker for HPC. *Cray User Group*, 2015.

[47] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, et al. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589, 2021.

[48] G. Kiar, S. T. Brown, T. Glatard, and A. C. Evans. A serverless tool for platform agnostic computational experiment management. *Frontiers in Neuroinformatics*, 13:12, 2019.

[49] Y. Kim, R. Jedrzejczak, N. I. Maltseva, M. Wilamowski, M. Endres, A. Godzik, K. Michalska, and A. Joachimiak. Crystal structure of Nsp15 endoribonuclease NendoU from SARS-CoV-2. *Protein Science*, 2020.

[50] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya. Automated fine-grained cpu cap control in serverless computing platform. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2289–2301, 2020.

[51] N. Kotsehub, M. Baughman, R. Chard, N. Hudson, P. Patros, O. Rana, I. Foster, and K. Chard. Flox: Federated learning with faas at the edge. In *IEEE International conference on eScience*, 2022.

[52] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[53] R. Kumar, M. Baughman, R. Chard, Z. Li, Y. Babuji, I. Foster, and K. Chard. Coding the computing continuum: Fluid function execution in heterogeneous computing environments. In *The Proceedings of Heterogeneity in Computing Workshop, in conjunction with IPDPS*, 2021.

[54] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459, 2017.

[55] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 90–103, 2019.

[56] C. Lin and H. Khazaei. Modeling and optimization of performance and cost of serverless applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):615–632, 2021.

[57] Z. Liu, H. Sharma, J.-S. Park, P. Kenesei, A. Miceli, J. Almer, R. Kettimuthu, and I. Foster. Braggnn: fast x-ray bragg peak analysis using deep learning. *IUCrJ*, 9(1), 2022.

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

[59] D. Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, (239):2, 2014.

[60] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical report, 2002.

[61] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[62] F. Perakis and C. Gutt. Towards molecular movies with x-ray photon correlation spectroscopy. *Physical Chemistry Chemical Physics*, 22(35):19443–19453, 2020.

[63] R. Priedhorsky and T. Randles. CharlieCloud: Unprivileged containers for user-defined software stacks in HPC. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, page 36. ACM, 2017.

[64] K. Satzke, I. E. Akkus, R. Chen, I. Rimac, M. Stein, A. Beck, P. Aditya, M. Vanga, and V. Hilt. Efficient gpu sharing for serverless workflows. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, pages 17–24, 2021.

[65] M. Saxena. Evaluation of mpi4py for natural language processing scenarios. Master's thesis, 2018.

[66] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of gridrpc: A remote procedure call api for grid computing. In *Grid Computing*, pages 274–278, 2002.

[67] T. Shaffer, Z. Li, B. Tovar, Y. Babuji, T. Dasso, Z. Surma, K. Chard, I. Foster, and D. Thain. Lightweight function monitors for fine-grained management in large scale python applications. In *35th IEEE International Parallel and Distributed Processing Symposium*, 2021.

[68] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX Annual Technical Conference (ATC 20)*, 2020.

[69] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu. Defuse: A dependency-guided function scheduler to mitigate cold starts on faas platforms. In *IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 194–204. IEEE, 2021.

[70] D. A. Sherrell, A. Lavens, M. Wilamowski, Y. Kim, R. Chard, K. Lazarski, G. Rosenbaum, R. Vescovi, J. L. Johnson, C. Akins, C. Chang, K. Michalska, G. Babnigg, I. Foster, and A. Joachimiak. Fixed-target serial crystallography at the Structural Biology Center. *Journal of Synchrotron Radiation*, 29(5), Sep 2022.

[71] T. J. Skluzacek, R. Wong, Z. Li, R. Chard, K. Chard, and I. Foster. A serverless framework for distributed bulk metadata extraction. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, pages 7–18, 2021.

[72] 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*, pages 154–168, 2017.

[73] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. 13(12):2438–2452, 2020.

[74] J. Stubbs, R. Dooley, and M. Vaughn. Containers-as-a-service via the actor model. In *11th Gateway Computing Environments Conference*, 2017.

[75] S. Tuecke, R. Ananthakrishnan, K. Chard, M. Lidman, B. McCollam, S. Rosen, and I. Foster. Globus Auth: A research identity and access management platform. In *12th IEEE International Conference on e-Science*, pages 203–212, Oct 2016.

[76] M. Turilli, M. Santcroos, and S. Jha. A comprehensive perspective on pilot-job systems. *ACM Computing Surveys*, 51(2):43, 2018.

[77] B. Varghese, P. Leitner, S. Ray, K. Chard, A. Barker, Y. Elkhatib, H. Herry, C. Hong, J. Singer, F. P. Tso, E. Yoneki, and M. F. Zhani. Cloud futurology. *Computer*, 52(9):68–77, 2019.

[78] R. Vescovi, R. Chard, N. Saint, B. Blaiszik, J. Pruyne, T. Bicer, A. Lavens, Z. Liu, M. E. Papka, S. Narayanan, et al. Linking scientific instruments and hpc: Patterns, technologies, experiences. *arXiv preprint arXiv:2204.05128*, 2022.

[79] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *USENIX Annual Technical Conference*, pages 133–146, 2018.

[80] L. Ward, G. Sivaraman, J. G. Pauloski, Y. Babuji, R. Chard, N. Dandu, P. C. Redfern, R. S. Assary, K. Chard, L. A. Curtiss, et al. Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing. In *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)*, pages 9–20. IEEE, 2021.

[81] M. Wilamowski, D. Sherrell, G. Minasov, Y. Kim, L. Shuvalova, A. Lavens, R. Chard, N. Maltseva, R. Jedrzejczak, M. Rosas-Lemus, N. Saint, I. Foster, K. Michalska, K. Satchell, and A. Joachimiak. Methylation of rna cap in sars-cov-2 captured by serial crystallography. *bioRxiv*, 2020.

[82] G. Winter, D. G. Waterman, J. M. Parkhurst, A. S. Brewster, R. J. Gildea, M. Gerstel, L. Fuentes-Montero, M. Vollmar, T. Michels-Clark, I. D. Young, et al. Dials: Implementation and evaluation of a new integration package. *Acta Crystallographica Section D*, 74(2):85–97, 2018.