Establishing a High-Performance and Productive Ecosystem for Distributed Execution of Python Functions Using Globus Compute

Rachana Ananthakrishnan*, Yadu Babuji*, Josh Bryan*, Kyle Chard*†, Ryan Chard†, Ben Clifford*, Ian Foster†*, Lev Gorenstein*, Kevin Hunter Kesling*, Chris Janidlo*, Daniel S. Katz‡, Reid Mello*, J. Gregory Pauloski*, Lei Wang*
*University of Chicago †Argonne National Laboratory ‡University of Illinois Urbana-Champaign

***Abstract*—The research computing ecosystem is increasingly heterogeneous and diverse. Democratizing access to these essential resources is critical for accelerating research progress. However, the gap between a high-level workload, such as Python in a Jupyter notebook, and the resources and interfaces exposed by HPC systems is significant. Users must securely authenticate, manage network connections, deploy and manage software, provision and configure nodes, and manage workload execution. Globus Compute reduces these barriers by providing a managed, fire-and-forget model that enables execution of Python functions across any resource to which a user has access. However, while Globus Compute has relieved users from many of the challenges of remote computing, we have observed some inefficiencies that remain in terms of use. For example, many users wrap external applications, such as C/C++, Fortran, and even MPI applications, in Python functions and users must deploy many endpoints on a single computer to exploit different configurations. In this paper we describe enhancements to Globus Compute to address these barriers: an asynchronous, future-based executor interface for submitting and monitoring tasks, shell and MPI-based function types, and a multi-user endpoint that can be deployed by administrators and used by authorized users.**

I. Introduction

High-performance computing (HPC) systems are powerful tools for tackling complex scientific and engineering problems. However, the expertise required to use HPC systems places barriers to adoption. These challenges are magnified when users need to deploy applications and workflows across multiple HPC systems. For example, users must securely authenticate with each resource, establish and maintain network connections, deploy and manage workflow software, provision nodes from heterogeneous resources, configure environments for execution, transfer input data, and monitor and manage execution of their tasks. Here we describe how Globus Compute, previously known as funcX [1], and its recent advancements, reduce these barriers and simplify use of advanced, distributed computing resources.

Globus Compute is a federated Function-as-a-Service (FaaS) platform that enables managed execution of Python functions across distributed computing *endpoints*. It implements a hybrid model, much like other Globus services [2], in which users or administrators first deploy Globus Compute Agent software on their resources, effectively exposing these resources to the Globus ecosystem, before they, or others, can execute functions on those endpoints. Users interact via a cloud service that provides a managed, fire-and-forget interface, i.e., a single highly-available user interface, while also buffering submission of tasks and retrieval of results.

Unlike other FaaS platforms, Globus Compute layers a federated model across existing computing resources, from laptops to supercomputers. Thus, it is distinct from other work that federates only Cloud FaaS providers (e.g., AWS Lambda and Google Cloud Functions) [3]. Globus Compute presents a unique, highly usable and productive platform for researchers to make use of distributed computing resources, significantly lowering barriers to adoption. It removes the need to manage and maintain network connections, embraces the web-standard OAuth 2 protocol for authentication and authorization, caches function submission and results in the cloud, and manages execution of functions on remote resources, optionally using containers. Building on the Parsl library [4], Globus Compute supports dynamic provisioning of nodes from batch schedulers (e.g., Slurm, PBS, and Flux) and Kubernetes.

Over the past several years, Globus Compute adoption has grown rapidly. More than 1000 users have executed functions and more than 4000 have accessed the web interface. Users have deployed over 12,000 endpoints and used these endpoints to run 44 million tasks. However, analysis of usage [5] and discussion with users highlighted several potential opportunities to improve the platform to further reduce barriers to use. For example, many users use Python functions to wrap execution of external applications and codes, such as C/C++ or Fortran codes, and in some cases to invoke MPI programs. While not antithetical to the Globus Compute model per se, these usages present opportunities for greater efficiency if Globus Compute was aware of the resources and task constraints. Second, users are deploying many endpoints on a single computer to exploit different configurations (e.g., queues, node types, number of nodes) and that for well-used resources, there are many endpoints running concurrently on behalf of many users. This presents challenges in terms of overheads and for administrators managing the many users on their machines.

In this paper, we describe enhancements to Globus Compute to support these use cases and further simplify adoption of

HPC resources. The Globus Compute SDK and Endpoint software are available on GitHub [6] and PyPI [7], [8]. Our contributions are:

- A future-based executor interface for asynchronous workload execution.
- A programmatic interface to run Shell commands and MPI applications as functions in a FaaS platform.
- A new runtime engine that supports dynamic partitioning of a batch job to run MPI applications concurrently.
- Multi-user endpoints that can be deployed by system administrators and then accessed by authorized users.
- Discussion of use of the proxy pass-by-reference model for out-of-band data transfer.

The remainder of the paper is as follows. Section II describes the Globus Compute model. Section III presents the Python SDK and the executor interface, ShellFunctions, and MPIFunctions. Section IV presents the multi-user endpoint. Section V describes how Globus Transfer and ProxyStore can be used to deal with large datasets. Section VI discusses use of Globus Compute in applications. Finally, Section VII presents related work and Section VIII summarizes our contributions.

II. Background

Globus Compute exposes a FaaS interface to users. Users define *functions* that they can then execute on remote *endpoints*. The cloud-hosted Globus Compute *web service* brokers user-endpoint communications to transmit functions and input arguments, and return results, reliably and securely.

Functions: Globus Compute decouples the two tasks of *defining* and *executing* functions. Users write Python functions that encapsulate the desired computation and can then be invoked one or more times from different locations and on different resources. These functions can perform data processing, simulations, or any other logic. While the functions themselves are written in Python, they may act as an interface to call programs written in other languages, applications, or scripts. However, this requires using standard Python libraries to fork processes, and leaves management of those processes to users.

Function execution is asynchronous. Functions are sent to the Globus Compute service, which buffers them until the requested endpoint is online. At that point, tasks are sent to the endpoint for execution. When the task is complete, the endpoint returns the results or exception back to the Globus Compute service. Users can retrieve results asynchronously as they are stored in the cloud for up to two weeks.

Endpoints: An endpoint is the logical representation of a remote computing resource. Endpoints are created by deploying the Globus Compute Agent—a pip installable Python code—on a specific computational resource. When a user invokes a function through Globus Compute, the system directs the request to the specified endpoint. The Agent listens for incoming tasks, executes the task on the local resource, monitors execution, captures errors, and returns results or exceptions back to the cloud service.

The Globus Compute Agent communicates with cloud-hosted Globus Compute RabbitMQ message queues via the AMQPS protocol (TLS/SSL encrypted Advanced Message Queuing Protocol). The Agent uses Parsl [4] to provision computing resources and to execute functions. Specifically, it relies on two abstractions: the *Provider* to provision resources, and the *Engine* to execute functions on those resource.

The Provider abstracts different computing resources, enabling Globus Compute endpoints to provision resources from different resource managers. The abstraction exposes an interface to obtain resources, check the status of requests, and to release resources. Globus Compute includes Provider implementations for many batch schedulers (e.g., Slurm, PBS, Flux), Kubernetes, and for use of local processes.

The `GlobusComputeEngine` wraps Parsl's `HighThroughputExecutor`. It uses a pilot job model to execute tasks on provisioned resources. Specifically, when started it creates an *interchange* locally to manage execution of functions, and deploys a *manager* on each provisioned resource. For each manager, it will deploy a set of *worker* processes, following the configuration supplied by the user (e.g., one worker per node, one worker per GPU, or one worker per core). When a task is ready to be executed, it is sent by the interchange to an available manager (one that is online and with available capacity). The workers then retrieve these tasks, execute them on the provisioned resources, and return results back to the interchange via the manager. All communication is with ZeroMQ, optionally using ZMQ Curve for authentication and encryption. Communication with nodes is multiplexed via managers to reduce the number of ports and connections.

Web service: The Globus Compute web service is operated as a hosted service and provides a single, highly-available interface for managing endpoints, functions, and tasks. The service is responsible for buffering tasks and results, ensuring they are not lost and are transmitted when the appropriate resources become available. Deployed in Amazon's Elastic Container Service (ECS), and leveraging various AWS services (e.g., Relational Database Service, Simple Storage Service, and Amazon MQ) the architecture is both reliable and highly scalable through automated scaling, replication, and geographic distribution. The service is also monitored such that administrators are notified when there are failures.

The web service is implemented as a FastAPI REST service. The REST API is deployed in containers on ECS. A relational database manages state (e.g., registered functions, endpoints, and tasks). When an endpoint is connected, the web service creates a pair of RabbitMQ queues for tasks and results. When a task is submitted, the web service places the task in the task queue for the specified endpoint. Large task inputs are stored in S3. The endpoint retrieves tasks from the *task queue*, executes them, and returns results to the *result queue* for that endpoint. A *result processor*, a Python application deployed in containers, monitors the queues and processes results. Processed results are stored in S3 until they are retrieved by the user.

Security model: The Globus Compute security model is designed to ensure that all interactions between users, the Globus

Compute service, and endpoints are secure and controlled. Central to this model is the use of the OAuth2-based Globus Auth [9] for authentication and authorization, which provides a secure and flexible way to manage user identities and access rights. This allows resource owners to control access based on user roles, ensuring that sensitive resources are protected from unauthorized access. Additionally, all communications between the Globus Compute service, users, and endpoints are encrypted using industry-standard protocols. Furthermore, Globus Compute employs robust auditing and logging mechanisms. Every action performed within the system, such as function invocation or resource allocation, is logged with detailed metadata. This provides traceability and accountability, allowing administrators to monitor usage patterns and ensure compliance with organizational policies.

III. Python SDK

The Globus Compute Python SDK provides a programmatic interface to Globus Compute. It wraps the Globus Compute REST API with a Pythonic interface supporting function registration, function execution, task management (e.g., result retrieval), and endpoint management.

A. Executor Interface

To better integrate with the Python ecosystem, and provide a simpler experience for users, we have developed a new asynchronous, future-based interface. Specifically, we extend Python's `concurrent.futures.Executor` interface by defining a subclass, `GlobusComputeExecutor`. The executor interface provides a `submit` method that takes a user-defined python function and its arguments and returns a `future` for subsequent monitoring and retrieval of results.

The benefits of this interface include the Pythonic programming model for executing tasks and also efficiency when compared with the traditional method requiring repeated polling for task status and to retrieve results. The Globus Compute Executor abstracts interactions with the Globus Compute REST API, including registering functions "on-the-fly" and batching of requests within a time period to avoid many individual REST requests to run tasks. The executor also instantiates an AMQPS connection with the Globus Compute web service that streams results directly and immediately as they arrive at the server back to the client. This is a far more efficient paradigm in terms of bytes over the wire, time spent waiting for results, and boilerplate code to check for results.

Listing 1 shows an example of using the executor to launch a task and retrieve results.

```
1 from globus_compute_sdk import Executor
2
3 def some_task(*a, **k):
4       return 1
5
6 with Executor(endpoint_id="...") as ex:
7     fut = ex.submit(some_task)
8     print("Result:", fut.result())
```

Listing 1: Using the Globus Compute Executor interface to execute a task.

B. Shell Functions

Many Globus Compute users employ Python functions as an interface to execute applications, programs in other languages, or other scripts and binaries. To better support these use cases we implemented an abstraction for representing a new type of function: *ShellFunction*. The ShellFunction allows for the specification of a command line string, along with runtime details such as run directory, per-task sandboxing, and task walltime. A ShellFunction returns a *ShellResult* that encapsulates the output from executing the command line string, wrapping the return code and snippets from the standard out and error streams.

Listing 2 presents an example ShellFunction that wraps the Linux "echo" command, accepts a message as input, and writes output to stdout. The command line string is formatted at invocation time with the message argument.

```
1 from globus_compute_sdk import ShellFunction,
      Executor
2
3 # Command is formatted with kwargs when invoked
4 sf = ShellFunction("echo '{message}'")
5
6 with Executor(endpoint_id="...") as ex:
7     for msg in ["hello", "hola", "bonjour"]:
8         future = ex.submit(sf, message=msg)
9         shell_result = future.result()
10        print(shell_result.stdout)
```

Listing 2: ShellFunction used to execute the `echo` command.

1) Shell results: The output from a ShellFunction is encapsulated in a ShellResult with the following fields: the return code from the execution of the command line supplied, the last N lines of the stdout stream and stderr stream, and the formatted command line string that was executed. Globus Compute will monitor and record the stdout and stderr streams. By default, it will capture the last 1000 lines of these streams; however, the number of lines can be configured.

2) Working directory: ShellFunctions executed on a remote system may operate on files local to the remote system. By default, the working directory of a ShellFunction is the Globus Compute endpoint path. Thus, there is potential for ShellFunctions to interfere with one another, for example, by overwriting files. To mitigate function contention, ShellFunctions can be configured to execute in a *sandbox*. When sandbox is enabled (i.e., specified in the endpoint configuration), Globus Compute will create a unique directory for each ShellFunction to execute using the task's UUID.

3) Walltime: A common requirement when executing a function remotely is to ensure that resources are not wasted if a function fails during execution. Because a function may run forever, oversight is required to potentially kill a process. We define a `walltime` keyword argument to ShellFunction that can be used to specify the maximum duration (in seconds) after which execution should be terminated. If the execution is terminated due to reaching the walltime, the return code will be set to 124: the shell return code when a timeout is exceeded.

Listing 3 shows an example ShellFunction in which the walltime is set to one second. In this case, the ShellFunction wraps the Linux "sleep" command and passes an argument of two seconds. Globus Compute will interrupt execution and the return code will be 124.

```
bf = ShellFunction("sleep 2", walltime=1)
future = executor.submit(bf)
print(future.returncode)
```

Listing 3: ShellFunction calls `sleep` with specified walltime.

C. MPI Functions

To better support HPC users, we seek to combine the power of MPI with the simple FaaS interface for remote computing. To achieve this goal, we define a new function type: *MPIFunction*. MPIFunction is an extension to ShellFunction. It supports the same interface for specifying the command to invoke on the endpoint and leverages the same monitoring support to capture output streams. However, rather than run a shell command, it executes an MPI application using a specified MPI launcher.

Given that MPI functions can be configured to use multiple cores across multiple nodes, we define a resource specification to describe the particular resources in a machine agnostic manner using the same representation as Parsl [4]. The resource specification is represented as a Python dictionary and can be configured with the number of nodes, number of ranks per node, and number of ranks. The specification is set when creating the Globus Compute Executor on the client-side. The resource specification can be configured as shown in Listing 4. This specification is translated to a machine-specific MPI launch command by the MPIFunction at runtime, for better portability across HPC systems.

```
executor.resource_specification = {
    # Nodes required for the application instance
    'num_nodes': <int>,
    # Ranks / app elements to launch per node
    'ranks_per_node': <int>,
    # Number of ranks in total
    'num_ranks': <int>,
}
```

Listing 4: Resource specification template for common MPI parameters.

1) MPIEngine: MPIFunctions must be executed in an environment that supports MPI execution. Unlike Python functions that are expected to run on a single node with some subset of the on-node compute resources (CPU/memory), MPI applications have more complex requirements. Generally, MPI applications require multiple MPI ranks (processes) launched across multiple nodes, along with complex affinity needs due to GPUs and NUMA environments. In a many-task paradigm, as is the case with Globus Compute, the runtime backend must be capable of executing multiple MPI applications with varied requirements concurrently within a single batch job.

To address these requirements we implement a new endpoint engine type: *GlobusMPIEngine*. GlobusMPIEngine implements advanced functionality to partition a batch job dynamically based on user-defined function requirements. It can automatically discover the resource available within a batch job on the Slurm and PBSPro batch systems.

When executing an MPIFunction, Globus Compute automatically prefixes the supplied command with $PARSL_MPI_PREFIX which resolves to an appropriate MPI launcher prefix (e.g., `mpiexec -n 4 -host <NODE1, NODE2>`). Listing 5 shows a configuration for a GlobusMPIEngine using Slurm. In this case, MPI tasks will be run over four nodes.

```
# Configuration for a Slurm based HPC system
display_name: SlurmHPC
engine:
  type: GlobusMPIEngine
  mpi_launcher: srun

  provider:
    type: SlurmProvider

    launcher:
      type: SimpleLauncher

    # Specify # of nodes per batch job that
    # will be shared by multiple MPIFunctions
    nodes_per_block: 4
```

Listing 5: Configuration of a Globus Compute Endpoint to support MPIFunction execution.

2) Running an MPIFunction: Listing 6 shows an example MPIFunction that calls the Linux "hostname" command on every rank it runs on. We supply a resource_specification requesting 2 nodes with a variable number of ranks per node (from 1 to 2). MPIFunctions return the same ShellResult described above and capture output streams in the same way. The output from this example is shown in Listing 7.

```
from globus_compute_sdk import MPIFunction

func = MPIFunction("hostname")
for n in range(1, 2):
    print(f'n={n}')
    executor.resource_specification = {
        "num_nodes": 2,
        "ranks_per_node": n,
    }
    future = executor.submit(func)
    mpi_result = future.result()
    print(mpi_result.stdout)
```

Listing 6: Executing several MPIFunctions with different resource specifications.

```
n=1
exp-14-08
exp-14-20
n=2
exp-14-08
exp-14-20
exp-14-08
exp-14-20
```

Listing 7: Results from running the example code in Listing 6

IV. Multi-user Endpoints

Globus Compute initially supported only single-user endpoints. Endpoints were installed in user space and could be

used exclusively by the user who installed them. Single-user endpoints are statically configured, and must be restarted to change their configuration, for example, to increase the number of nodes allocated. As a result, it is common for users to run several endpoints on an HPC resource. Further, administrators have no visibility into the use of their resources and are unable to easily help debug user problems (e.g., with configurations for their resources). To overcome these challenges we have developed a new multi-user endpoint that can be installed by administrators and that enables remote use and dynamic endpoint configuration by users [10].

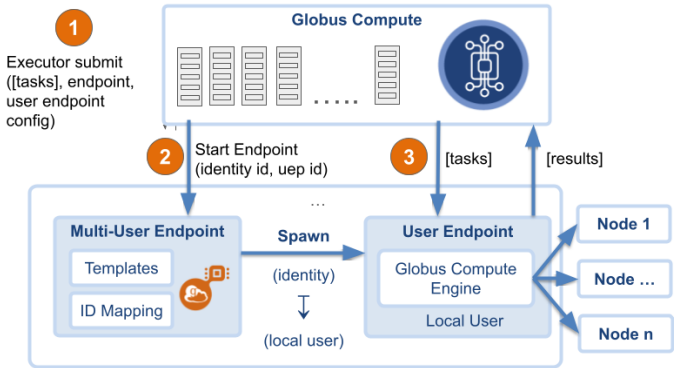

Fig. 1: Multi-user endpoint architecture. (1) Users specify a user endpoint configuration when submitting a task to a multi-user endpoint. (2) The Globus Compute service issues a *Start Endpoint* request to the multi-user endpoint. The multi-user endpoint determines whether an appropriate user endpoint is currently operating; if not, it spawns a user endpoint on behalf of the user to meet the task requirements. (3) The user endpoint connects to the service and receives tasks to perform.

At its core, the multi-user endpoint is a process manager: it starts user endpoint agents upon request from the Globus Compute service. Importantly, a multi-user endpoint does not run tasks for users. It starts child processes (fork()) on the host (becoming the appropriate local user and dropping privileges), and lets the user compute endpoint agent (exec()) process tasks as normal.

We describe below the multi-user endpoint model from the administrator and user perspective.

A. Administrator Perspective

Administrators can deploy a multi-user endpoint on a shared resource. They can configure the endpoint with policies regarding how users are mapped to local accounts and define templates controlling how endpoints are used by users.

1) Installation and configuration: Administrators can install the Globus Compute Agent for supported Linux Distributions from RPM and Deb repositories, or through pip. Once installed, they can configure a multi-user endpoint configuration using the same `globus-compute-endpoint configure` subcommand as single user endpoints, with an additional `multi-user` flag. The administrator must authenticate using Globus Auth to register the multi-user endpoint with Globus Compute. They can also define metadata used to describe the endpoint for search and display on the Globus web application.

2) Identity mapping: When the multi-user endpoint is asked to start a local user endpoint, an important question is how to map it to a valid local user and UID. To do so, we use the identity mapping logic from Globus Connect Server, the agent software used by Globus Transfer.

Every request from the Globus Compute service to start a user endpoint includes the identity information of the user who submitted the request. The multi-user endpoint retrieves the identity information and compares it against the mapping file to a) determine if the user is authorized to access the endpoint; and b) determine the local user account in which to spawn the user endpoint.

The identity mapping process supports several options: a default mapping for cases where there is only one allowed domain, pattern-based mappings, and callouts to external programs for custom mapping algorithms. With expression-based mapping, administrators can write rules that extract data from fields in the Globus identity document to form local usernames. This works well when there is a common relationship between user identity information and local account names. Listing 8 shows a simple example where identities from the "uchicago.edu" domain are mapped to the same local username. To simplify mapping, we support a simple regular expression matching language and provide functions for common transformations (e.g., ignoring case).

To support more complicated mapping scenarios, we allow administrators to use external programs (e.g., a Python or Bash script) to perform the mapping. This allows administrators to implement more complex mapping logic or consult external sources of information such as databases or LDAP servers for mappings. The Globus Compute Agent will make a call to the specified program for each request.

```
[
  {
    "DATA_TYPE": "expression_identity_mapping
    #1.0.0",
    "mappings": [
      {
        "source": "{username}",
        "match": "(.*)@uchicago\\.edu",
        "output": "{0}"
      }
    ]
  }
]
```

Listing 8: Identity mapping configuration that will convert any user with an @uchicago.edu identity to the same username on the local system

3) Template configuration: One goal of the multi-user endpoint is to simplify use. Many endpoint configuration options remain static for a single resource (e.g., scheduler type) while others must adhere to site or HPC resource policies (e.g., node limits, walltimes). To address this need to simplify use, we use a template-based approach via which administrators can define a template for configurable options that are exposed to users.

Administrators can optionally also define a schema for the template configuration properties to protect against injections and also (in the future) to help guide users when specifying their configuration.

We adopt Jinja2 templates as they are commonly used in Python programs. In the multi-user endpoint configuration, the administrator can specify a template to use for that endpoint. Most administrator-installed multi-user endpoints will likely need at least one templatable field (e.g., account id), but beyond that, this file can be configurable as required. Listing 9 shows an example template for a resource using Slurm. In this case, common configurations for the resource are specified, such as using GlobusComputeEngine, SlurmProvider, and SrunLauncher. The administrator also restricts users of the multi-user endpoint to use the "cpu" partition. The configuration defines three configurable properties: nodes_per_block, account, and walltime. Each property is mapped to a Jinja template option, denoted with double braces. Other Jinja syntax is supported with the use of a default property.

```
engine:
  type: GlobusComputeEngine
  nodes_per_block: {{ NODES_PER_BLOCK }}

  provider:
    type: SlurmProvider
    partition: cpu
    account: {{ ACCOUNT_ID }}
    walltime: {{ WALLTIME|default("00:30:00") }}

    launcher:
      type: SrunLauncher
```

Listing 9: Multi-user configuration template specifying fixed provider type and partition, while enabling users to configure account and walltime.

When a user passes a configuration to the multi-user endpoint, the endpoint first validates that the configuration document meets the specified schema. It then forks and drops privileges, passing the user-provided configuration data to the administrator-written template via a Jinja processor. This configuration is then used to start the user endpoint.

Note that while we focus here on benefits to administrators, non-administrators also benefit from this feature. Rather than having to manage multiple endpoint configurations (for example, charging different HPC accounts, changing provisioned cluster size, or choosing different walltime limits), a user can write a template to allow all of these items to be specified at task submission time.

4) Allowed functions: Administrators may want to restrict the functions that can be executed on the endpoint, for example, when deploying portals or science gateways providing compute capabilities for communities. The multi-user endpoint can be configured with permitted functions by specifying a list of function UUIDs. In early use of this feature, administrators have implemented an out-of-band process in which they manually review function code before adding the UUIDs to the allowed function list. This feature relies on the fact that all registered Globus Compute functions are immutable.

5) Authentication policies: The policies described above are implemented at the compute endpoint. To provide more flexible authentication and authorization policies, we also support cloud-enforcement of particular policies. In these cases, the Globus Compute service validates policies before submitting a request to the endpoint. We support a small set of policies, defined via Globus Auth and shareable with other Globus services. These policies can express required authentication domains or excluded domains, require that users must have authenticated within the given session with a particular identity provider, or have authenticated within a particular period of time.

B. User Perspective

Users can submit tasks to a multi-user endpoint in the same way as they currently do with single-user endpoints. The general workflow is as follows. They first discover the ID of a multi-user endpoint, for example via resource-specific documentation or via search in the Globus Compute web application or API. They then configure their client code to submit tasks to that endpoint via the executor API. When they submit the tasks, the multi-user endpoint spawns the user endpoint, and subsequent communication is directly with the single user endpoint. Note that users do not know that a user endpoint process is spawned; nor do they need a new ID to submit to the spawned endpoint. Once the submitted tasks are completed, the user endpoint is destroyed.

One difference from a user's perspective, in addition to no longer needing to install or maintain their own endpoints, is that users can now specify a resource configuration when defining their executor. Listing 10 shows an example using the executor interface with the resource configuration. The resource configuration is defined as a Python dictionary and passed to the executor before use. The user configuration must specify all of the required attributes from the template file.

```
from globus_compute_sdk import Executor

uep_conf = {
  "NODES_PER_BLOCK": 64
  "ACCOUNT_ID": "314159265",
  "WALLTIME": "00:20:00"
}

mep_ep_id = "..."

with Executor(endpoint_id=mep_ep_id) as gce:
   gce.user_endpoint_config = uep_conf
   fut = gce.submit(hello_world)
   res = fut.result()
```

Listing 10: User configuration to make use of the multi-user configuration template from Listing 9.

Listing 10 shows how users refer only to a single multi-user endpoint ID when creating the executor and submitting tasks. However, it is possible for users to define different configurations that lead to creation of new user endpoints. To handle this mapping, Globus Compute maintains a mapping between a hash of the configuration and the user endpoint that is spawned. Thus, creation of executors with the same

user configurations will direct tasks to the same user endpoint. Users can therefore force use of different user endpoints by modifying the configuration such that the hash is different.

C. Discussion

The multi-user endpoint provides a number of important benefits to both users and administrators.

Lowering Barriers of Use: HPC systems often have unique configurations and tools for running tasks. As a result, configuration of a Globus Compute endpoint can be complicated for end users. Shifting the burden of configuration to HPC administrators (experts in their own systems) allows end users to submit functions without managing unnecessary-to-their-research boilerplate such as SSH configuration details, a specific cluster's firewall policy, or how to specify the resource's scheduler, options, and so forth.

Improved Access Control: With multi-user endpoints, administrators have granular control over user access permissions and resource usage. User access is controlled by the identity mappings to local user accounts, and can be augmented at the web service layer through authentication policies. All limits placed on users through scheduler controls as well as standard Unix limits are respected by the multi-user endpoint. Consequently, limited access can be granted to users without the need for SSH access to the machine.

Efficient Resource Utilization: Multi-user compute endpoints allow administrators to optimize resource allocation and utilization by creating predefined configurations for users or groups of users. For example, an administrator may provide full access to a select group, while providing only limited access to a wider set of users.

Improved user experience: The multi-user model removes the need for users to log in deploy and manage endpoints. It also removes the need to maintain multiple endpoints for different configurations. Instead, all configuration can be done remotely via Globus Compute APIs.

V. Data Movement

Globus Compute limits the amount of data that can be passed to, or returned from a task to 10 MB. For data sizes beyond 10 MB, such as large data frames, files, or machine learning models, external transfer methods can be used, such as Globus Transfer and ProxyStore.

A. Globus Transfer

A simple solution for data movement is to write data to a local file system, replacing task arguments or results with file paths to the corresponding objects. However, if applications require access to that data, it must be then moved between Globus Compute endpoints. Globus transfer, which offers a secure, fire-and-forget model for reliable and performant file transfer between Globus Connect endpoints [2], can be used for this purpose. Globus Connect software is widely deployed on research computing facilities (there are more than 60,000 active endpoints at the time of writing), and Globus Connect Personal endpoints can be configured as needed when leveraging personal or edge devices. As with Globus Compute, authentication is provided through Globus Auth and thus Globus Compute and Transfer can be easily used together.

B. ProxyStore

ProxyStore [11] streamlines data flow management in distributed Python applications. At its core is the transparent object proxy, a reference-like object that refers to an object in distributed storage. The proxy is "transparent" because it automatically resolves its target object when first used and then forwards all operations on itself to the target.

A proxy is initialized with a *factory*, a callable object that, when invoked, retrieves the target from remote storage. Thus, the complexity of interacting with low-level communication protocols and storage mediums is encapsulated within the factory. The proxy can be efficiently passed around without the consumer of the proxy needing to be aware of the communication mechanisms being used.

This approach effectively combines the advantages of pass-by-reference and pass-by-value patterns and yields many benefits for Globus Compute applications. Proxying task arguments and results avoids transfer of large objects through the cloud service which improves task latency and circumvents the 10 MB payload limit. Task code does not need to be modified to work with proxies due to their transparent behavior. Objects reused by many tasks can be cached in the worker process. Proxies can leverage many communication channels and storage systems to fit the specific deployment. For example, TCP, RDMA, objects stores, and shared file systems can be used when the client and workers are located within the same site, and peer-to-peer methods (Globus Transfer and UDP hole punching) are provided for wide-area deployments.

ProxyStore can be easily integrated into Globus Compute applications. Initializing the Store interface and creating a proxy of an object requires only a few lines of code. More sophisticated applications can use the `Executor` wrapper provided by ProxyStore to wrap their Globus Compute `Executor`. This wrapper automatically proxies task arguments and results based on a user-defined policy (e.g., object size or type) and will clean up proxied objects based on the lifetimes of the tasks with which the proxies are associated [12].

VI. Discussion

Since November 2022, almost 17 million tasks have been executed with Globus Compute. We see in Fig. 2, which shows tasks per day, increasing and more consistent use over time.

Multi-user endpoints were released in April, 2024. By August 2024, 87 multi-user endpoints had been deployed and used to spawn 1718 user endpoints: more than 13% of the then total 12,418 Globus Compute endpoints. We outline in the following some of the purposes to which multi-user endpoints are being applied.

Resource scheduling: Delta [13] builds on Globus Compute to provide a single interface for task submission to many endpoints. Delta profiles the execution of functions on

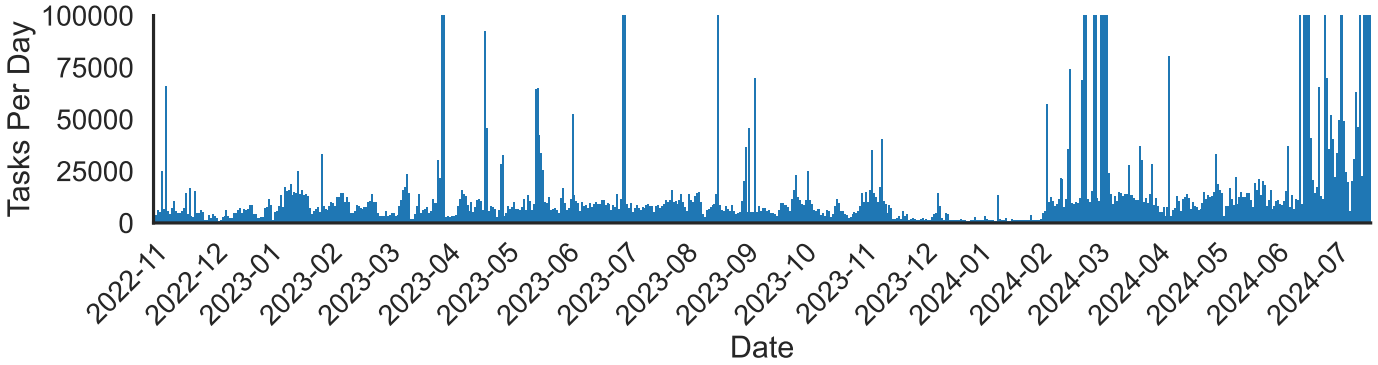

Fig. 2: Task invocations per day, truncated at 100,000 tasks from November 28, 2022 to August 14, 2024.

different endpoints, constructing a predictive model that can estimate runtime based on the specific capabilities of each resource. With this model, Delta can make informed decisions about where to schedule workloads in the most effective and timely manner, maximizing the overall efficiency of the system. GreenFaaS [14] uses a similar model, but focused on improving the sustainability and energy efficiency of FaaS workloads. GreenFaaS deploys an energy monitor alongside the Globus Compute Endpoint and captures energy use as tasks are executed. It uses this information to then schedule tasks to specific resources based on predicted energy use.

Delta and GreenFaaS benefit from the dynamic configuration of endpoints provided by multi-user endpoints. They can remotely create user endpoints with specific configurations that best match submitted workload. They can also dynamically resize configured resources to reduce overprovisioning.

Real-time analysis: The Advanced Photon Source (APS) at Argonne National Laboratory uses Globus Flows [15] with Globus Compute to perform near-real-time analysis of synchrotron workloads by leveraging resources at the Argonne Leadership Computing Facility (ALCF) [16]. Globus Flows orchestrates data transfer, processing, and publication tasks using Globus Compute to execute functions to perform analysis, create visualizations, extract metadata, and perform other tasks such as training machine learning models [17], [18]. This approach enables dynamic, on-demand allocation of computing resources necessary to meet the demands of the beamlines. Multi-user endpoints enable the real-time computation to be adapted to the needs of the acquired data. Thus, the flow can provision resources suitable to handle incoming data, and deprovision resources when the task is complete.

Science Gateways: OpenCosmo and Earth Science Grid Federation (ESGF) [19] build on Globus Compute to deliver analysis functionality directly to users via a web portal. Through these portals, users can submit analysis tasks, which are subsequently routed to Globus Compute and executed by an ALCF community service account. To adhere to the strict security requirements of the ALCF, these projects rely on Globus Compute's ability to restrict execution exclusively to specific, pre-approved functions. This ensures that only authorized operations can be performed.

VII. Related Work

Cloud FaaS systems, such as Amazon Lambda [20], Azure Functions [21], and Google Cloud Functions [22], enable users to run code in response to events without provisioning or managing servers. These platforms allow users to deploy small, modular functions that automatically scale with demand, making them ideal for handling variable workloads. However, unlike Globus Compute, which can support workloads of any size, including MPI workloads, that may be distributed across diverse resources, cloud FaaS services are primarily confined to their respective platforms.

Open source FaaS systems such as Apache OpenWhisk [23], Fn [24], Kubeless [25], Abaco [26], ChainFaaS [27], and DFaaS [28] enable FaaS platforms to be deployed on local resources. These systems primarily use Kubernetes for deployment and are limited to use on a single computing resource. ChainFaaS uses a blockchain-based approach for leveraging idle personal computers in a volunteer computing model. DFaaS implements a federated and decentralized model for edge devices via use of a peer-to-peer model. These platforms do not provide the same managed computation nor do they support HPC resources.

rFaaS [29] is a FaaS system designed specifically for HPC. It extends the traditional FaaS model with RDMA to accelerate the execution of functions HPC systems. In rFaaS each function is invoked by directly writing the input data into the memory of the designated worker, providing efficient and low-latency execution across distributed systems. Like the open source systems above, rFaaS is designed for single deployment on a resource, rather than federated deployment across HPC resources. It may be possible to leverage rFaaS concepts in the Globus Compute Agent to improve performance.

Many tools and libraries have developed to abstract the challenges of using different batch schedulers, enabling job submission and monitoring without requiring users to navigate the specific nuances of each scheduler. For instance, SAGA [30], DRMAA [31], and PSI/J [32] offer unified interfaces that allow users to submit batch jobs seamlessly across different schedulers. Some workflow systems, such as Parsl, also facilitate remote computing. These tools are responsible for orchestrating the execution of tasks on a

designated resource and rely on SSH connections to remotely act on resources. Globus Compute builds on the Parsl library to both abstract different batch schedulers and orchestrate task execution within an endpoint.

Open OnDemand [33] provides a web-based interface for remote access to HPC systems, allowing users to log in with their institutional credentials, manage data, and create and manage computational jobs. Open OnDemand employs templates for batch jobs that are then mapped to the underlying scheduler's batch submission files. Open OnDemand operates as a web application and uses Apache-supported authentication methods. Identity mapping is handled similarly to Globus Compute, using static mappings, regex patterns, and custom scripts. Globus Compute provides a higher-level interface designed specifically for executing functions across multiple connected endpoints.

VIII. Summary

Globus Compute's hybrid cloud-edge model can significantly reduce barriers for adopting heterogeneous, specialized, and high-performance remote computing infrastructure. Early use of Globus Compute has shown its suitability for a range of use cases, but also highlighted areas for enhancement that could simplify use. In this paper, we presented new features designed to better support these use cases: an executor API for simple invocation, multi-user endpoints to separate configuration and management responsibilities between users and administrators, ShellFunctions and MPIFunctions to better support HPC users, and pass-by-reference data transfer methods that avoid moving data through the cloud service. We described how these features have been designed and the advantages they offer to different user communities.

References

[1] 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 *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2020, p. 65–76.

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

[3] S. Ristov, S. Pedratscher, and T. Fahringer, "AFCL: An abstract function choreography language for serverless workflow specification," *Future Generation Computer Systems*, vol. 114, pp. 368–382, 2021.

[4] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in Python," in *28th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2019.

[5] A. Bauer, H. Pan, R. Chard, Y. Babuji, J. Bryan, D. Tiwari, I. Foster, and K. Chard, "The Globus Compute dataset: An open function-as-a-service dataset from the edge to the cloud," *Future Generation Computer Systems*, vol. 153, pp. 558–574, 2024.

[6] "Globus Compute GitHub," https://github.com/globus/globus-compute. Accessed Sep 2024.

[7] "Globus Compute SDK PyPI," https://pypi.org/project/globus-compute-sdk/. Accessed Sep 2024.

[8] "Globus Compute Endpoint PyPI," https://pypi.org/project/globus-compute-endpoint/. Accessed Sep 2024.

[9] 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 *IEEE 12th International Conference on e-Science*, 2016, pp. 203–212.

[10] R. Ananthakrishnan, Y. Babuji, M. Baughman, J. Bryan, K. Chard, R. Chard, B. Clifford, I. Foster, D. S. Katz, K. Hunter Kesling, C. Janidlo, R. Mello, and L. Wang, "Enabling remote management of FaaS endpoints with Globus Compute Multi-User endpoints," in *Practice and Experience in Advanced Research Computing*. ACM, 2024.

[11] J. G. Pauloski, V. Hayot-Sasson, L. Ward, N. Hudson, C. Sabino, M. Baughman, K. Chard, and I. Foster, "Accelerating communications in federated applications with transparent object proxies," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2023.

[12] J. G. Pauloski, V. Hayot-Sasson, L. Ward, A. Brace, A. Bauer, K. Chard, and I. Foster, "Object proxy patterns for accelerating distributed applications," 2024. [Online]. Available: https://arxiv.org/abs/2407.01764

[13] 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 *International Parallel and Distributed Processing Symposium Workshops*. IEEE, 2021, pp. 66–75.

[14] A. Kamatar, V. Hayot-Sasson, Y. Babuji, A. Bauer, G. Rattihalli, N. Hogade, D. Milojicic, K. Chard, and I. Foster, "GreenFaaS: Maximizing energy efficiency of HPC workloads with FaaS," *arXiv preprint arXiv:2406.17710*, 2024.

[15] R. Chard, J. Pruyne, K. McKee, J. Bryan, B. Raumann, R. Ananthakrishnan, K. Chard, and I. T. Foster, "Globus automation services: Research process automation across the space–time continuum," *Future Generation Computer Systems*, vol. 142, pp. 393–409, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23000183

[16] R. Vescovi, R. Chard, N. D. Saint, B. Blaiszik, J. Pruyne, T. Bicer, A. Lavens, Z. Liu, M. E. Papka, S. Narayanan *et al.*, "Linking scientific instruments and computation: Patterns, technologies, and experiences," *Patterns*, vol. 3, no. 10, 2022.

[17] Z. Liu, T. Bicer, R. Kettimuthu, D. Gursoy, F. De Carlo, and I. Foster, "TomoGAN: Low-dose synchrotron x-ray tomography with generative adversarial networks," *JOSA A*, vol. 37, no. 3, pp. 422–434, 2020.

[18] 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*, vol. 9, no. 1, pp. 104–113, 2022.

[19] D. N. Williams, R. Ananthakrishnan, D. Bernholdt, S. Bharathi, D. Brown, M. Chen, A. Chervenak, L. Cinquini, R. Drach, I. Foster *et al.*, "The Earth System Grid: Enabling access to multimodel climate simulation data," *Bulletin of the American Meteorological Society*, vol. 90, no. 2, pp. 195–206, 2009.

[20] Amazon Lambda. https://aws.amazon.com/lambda. Accessed Aug 2024.

[21] Microsoft Azure Functions Documentation. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale. Accessed Aug 2024.

[22] Google Cloud Functions. https://cloud.google.com/functions/. Accessed August 16, 2024.

[23] Apache OpenWhisk. http://openwhisk.apache.org/. Accessed May 2022.

[24] Fn project. https://fnproject.io. Accessed Aug 2024.

[25] Kubeless. https://kubeless.io. Accessed Aug 2024.

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

[27] S. Ghaemi, H. Khazaei, and P. Musilek, "ChainFaaS: An open blockchain-based serverless platform," *IEEE Access*, vol. 8, pp. 131 760–131 778, 2020.

[28] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo, "DFaaS: Decentralized function-as-a-service for federated edge computing," in *10th International Conference on Cloud Networking*. IEEE, 2021, pp. 1–4.

[29] M. Copik, K. Taranov, A. Calotoiu, and T. Hoefler, "rFaaS: Enabling high performance serverless with RDMA and leases," in *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2023, pp. 897–907.

[30] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf, "SAGA: A simple API for grid applications. High-level application programming on the grid," *Computational Methods in Science and Technology*, vol. 12, no. 1, pp. 7–20, 2006.

[31] P. Troger, H. Rajic, A. Haas, and P. Domagalski, "Standardization of an API for distributed resource management systems," in *7th IEEE International Symposium on Cluster Computing and the Grid*, 2007, pp. 619–626.

[32] M. Hategan-Marandiuc, A. Merzky, N. Collier, K. Maheshwari, J. Ozik, M. Turilli, A. Wilke, J. M. Wozniak, K. Chard, I. Foster, R. F. da Silva, S. Jha, and D. Laney, "PSI/J: A portable interface for submitting, monitoring, and managing jobs," in *19th International Conference on e-Science*, 2023, pp. 1–10.

[33] D. Hudak, D. Johnson, A. Chalker, J. Nicklas, E. Franz, T. Dockendorf, and B. L. McMichael, "Open OnDemand: A web-based client portal for HPC centers," *Journal of Open Source Software*, vol. 3, no. 25, p. 622, 2018.