The Globus Compute Dataset: An Open Function-as-a-Service Dataset From the Edge to the Cloud

André Bauer[a,b], Haochen Pan[a], Ryan Chard[b], Yadu Babuji[a], Josh Bryan[a], Devesh Tiwari[c], Ian Foster[b,a], Kyle Chard[a,b]

[a] *University of Chicago, United States*
[b] *Argonne National Laboratory, United States*
[c] *Northeastern University, United States*

Abstract

We present a unique function-as-a-service (FaaS) dataset capturing the use of the Globus Compute (previously funcX) platform. Globus Compute implements a federated model via which users may deploy endpoints on arbitrary remote computers, from the edge to high performance computing (HPC) cluster, and they may then invoke Python functions on those endpoints via a reliable cloud-hosted service. The dataset covers 31 weeks and includes 2 121 472 task submissions from 252 users executed on 580 remote computing endpoints. It includes 277 386 registered functions. We describe the dataset and various observations, some that are similar to other FaaS datasets, for example, that 74% of tasks run for less than 1 second, and some that are unique to Globus Compute, for example, that endpoints are used in different ways and that the majority of functions are related to scientific computing and machine learning. To the best of our knowledge, this dataset represents the first federated FaaS dataset that includes user workloads, distributed computing endpoints, and analysis of registered function bodies. We expect the dataset to be useful for researching FaaS architectures, workload modeling, container warming, and other distributed computing architectures.

Keywords: Serverless computing, Function-as-a-Service, Globus Compute, FAIR dataset, Computing continuum

1. Introduction

Serverless computing has emerged as a productive and scalable approach to developing and deploying applications. Serverless computing is a cloud-based paradigm that enables developers to write code without worrying about infrastructure management, thereby reducing barriers and potentially increasing efficiency and scalability. The most common serverless paradigm, Function-as-a-Service (FaaS), popularized by Amazon Lambda, Google Functions, and Azure Functions, allows users to register programming functions with a cloud service and then invoke those functions with arbitrary input arguments. The cloud provider transparently provisions the virtual infrastructure (typically a container), executes the function, and returns results.

In this paper, we describe and analyze an open FaaS dataset from Globus Compute (previously known as funcX [1, 2]). Globus Compute implements a unique federated FaaS model that is primarily aimed at scientific computing scenarios. Unlike cloud FaaS platforms, Globus Compute's federated model is based on a hybrid architecture via which users deploy computing endpoints on arbitrary remote resources (from the edge to the data center). These endpoints are responsible for managing the local computing infrastructure, including provisioning resources via different interfaces (e.g., batch schedulers, Kubernetes, local threads). Globus Compute endpoints are deployed on various high performance computing (HPC) systems, cloud platforms, and edge devices (e.g., Raspberry Pi, Jetson Nano, and personal laptops). Like other FaaS platforms, Globus Compute exposes a REST API that enables users to define their functions, specify input parameters, and submit tasks for execution. However, unlike cloud FaaS platforms, users must also specify the endpoint on which to execute the function. Globus Compute handles the distribution of task execution across available resources, ensuring scalability and efficient utilization of computational power. It also provides features like function versioning, result caching, and performance monitoring.

We present a new open-source dataset that encompasses 31 weeks of data, with 2.1 million function invocations from 252 users executed across 580 distributed endpoints. We explore a wide range of dimensions, such as usage patterns, function invocation patterns, endpoint configurations, function bodies, and geographical locations. Our analysis aims to uncover valuable insights that can contribute to a deeper understanding of serverless architectures and their performance characteristics. For instance, the extracted arrival rates and run time distribution can be used for simulating scientific workloads and exploring scheduling challenges across the computing continuum.

To promote FAIR (findability, accessibility, interoperability, and reusability) research, we have publicly made the dataset and analysis scripts available.[1] We intend to update the dataset as the adoption of the platform grows.

In contrast to other public FaaS datasets, such as the Azure dataset [3], our dataset is unique in the following ways.

1. We capture the function execution lifecycle in detail, with six timestamps as functions flow through the platform

[1] The dataset is publicly available: `https://doi.org/10.5281/zenodo.10044780`

and endpoints. These timestamps allow us to explore time taken to acquire resources from batch schedule for example.
2. We consider a federated model that provides inform tion about how 580 endpoints are configured (e.g., HI cloud, and edge) and where they are deployed across computing continuum. This inherently differs from a s gle hosted service that users use from a data center.
3. We include data from 252 users spanning a wide vari of mostly scientific applications.
4. We analyze the function source code to classify the diff ent "types" of functions and compare how they are us
5. Our dataset spans 31 weeks of usage, which allows us to explore temporal patterns.

Like the Azure dataset, which has generated new research in workload performance prediction [4], serverless function scheduling [5], load-balancing policies [6], and cold-starts [7], we believe that our dataset and the insights presented in this paper can advance research for FaaS architectures, workload modeling, container warming, and other distributed computing architectures. This is particularly noteworthy due to the level of detail offered by our dataset.

The remainder of this article is structured as follows: Section 2 introduces the Globus Compute platform and briefly outlines three example use cases. Section 3 describes the dataset and how it was obtained. Section 4 presents a high-level statistical overview of the dataset. Section 5 provides a deeper analysis of various topics of interest. Section 6 reviews related work. Finally, Section 7 summarizes our contributions.

2. Globus Compute

Globus Compute, previously known as funcX [1, 2], is a unique federated function-as-a-service platform primarily used for scientific computing. We describe here the architecture of Globus Compute and some representative applications.

2.1. Globus Compute Architecture

The Globus Compute architecture is shown in Figure 1. Unlike traditional cloud-hosted FaaS platforms (e.g., Amazon Lambda) and open-source platforms (e.g., OpenWhisk [8] or OpenFaaS [9]), Globus Compute combines a single cloud-hosted web service with an ecosystem of user deployed endpoints. The cloud service exposes a REST API to users and a message queue interface to remote computing endpoints.

2.1.1. User interface

Task execution is similar to other FaaS platforms. First, users register Python functions with the cloud service by passing a serialized function body. The Cloud service stores the serialized function body in the database and assigns a unique UUID for subsequent use. Users may then invoke the function by supplying the function UUID, target endpoint UUID, and

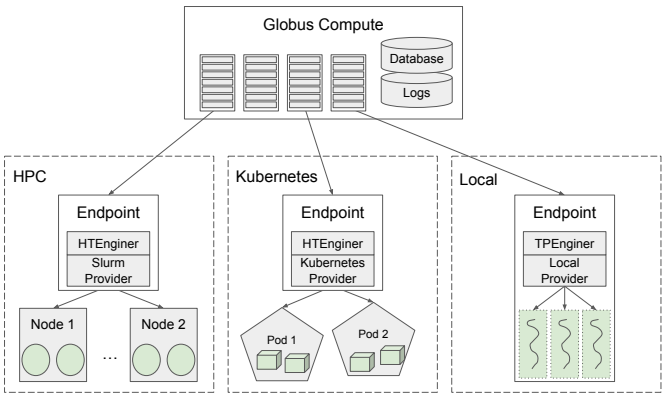

Figure 1: Globus Compute Architecture. The cloud service maintains state (e.g., functions, endpoints, and tasks). It uses message queues to communicate with remote endpoints. Endpoints provision resources from different sources (e.g., HPC, Kubernetes, Local) and execute tasks on those resources.

```
1  from globus_compute_sdk import Executor
2
3  # function to be executed
4  def science_function(input_args):
5      # do work
6      return result
7
8  # create the executor for a specific endpoint
9  gce = Executor(endpoint_id=<endpoint_id>)
10
11 # submit the function to be executed
12 future = gce.submit(science_function,
13                      input_args=<args>)
14
15 # wait for the task to complete
16 print(future.result())
```

Listing 1: Using Globus Compute to execute a function: First, a simple function is defined (Line 4). Then, an Executor is created specifying the endpoint to use (Line 9). The Executor is used to submit the function and input arguments (Line 12). Finally, the future is used to access the result (Line 16).

optionally any input arguments. The Globus Compute SDK offers various interfaces to simplify use, including a batch submission interface and an implementation of Python's concurrent futures executor interface. In Listing 1, we show an example of using the executor interface to execute a task.

Globus Compute uses Globus Auth for authentication and authorization. Users may authenticate using one of hundreds of supported identity providers. The Globus Compute service is represented as an OAuth 2 resource server with associated OAuth 2 scopes. Users may obtain OAuth 2 access tokens to access the service and register, manage, and execute functions. Upon creation, endpoints are securely paired with the Globus Compute service (by following an OAuth 2 flow). Globus Compute allows endpoints to be shared among a group of users under restrictive conditions: it is an admin operation (not exposed to users) that must be done by the Globus Compute team. We only enable sharing when endpoint owners and resource owners approve. In such cases, the endpoint is associated with a

user-managed Globus Group. Further, an endpoint manager can specify an allowlist of function UUIDs that may be executed at the endpoint by authorized users.

2.1.2. Endpoints

Globus Compute endpoints are designed to abstract remote computing resources. They are assigned to retrieve tasks from the cloud service and then execute them in a local environment. Globus Compute endpoints use Parsl [10] to dynamically provision compute resources and execute tasks. The model relies on two key constructs: *Providers* and *Engines*. Providers abstract the interface used to provision resources from different parallel and distributed computing systems. For example, Globus Compute includes providers for local processes, batch schedulers (e.g., Slurm, portable batch system (PBS), and load sharing facility (LSF)), and Kubernetes. In each case, the provider communicates with the target system to request resources (e.g., processes, nodes, or containers), deploy required software on those resources (e.g., the endpoint worker), and monitor usage.

An Engine manages how tasks are executed on provisioned resources. We leverage Parsl's pilot job model, which deploys a manager process on a provisioned node to manage communication and deploys one or more Python worker processes to execute tasks. The worker receives tasks from the manager, executes them, and returns the results. Like other FaaS systems, Globus Compute enables workers to be deployed in containers to create customized and isolated execution environments. It is also able to scale resources to meet workload needs dynamically; however, unlike cloud FaaS systems, Globus Compute layers such scaling behavior over various resource types, including Kubernetes and HPC schedulers.

2.2. Example Use Cases

Globus Compute is used for a range of use cases. We describe briefly three common examples: bag of tasks, federated workflows, and computation as a service.

2.2.1. Bag of Tasks

At the height of the COVID-19 pandemic, the US Department of Energy formed the National Virtual Biotechnology Laboratory (NVBL) to pool resources to address key challenges in response to the pandemic [11, 12]. As part of the NVBL, a vast amount of computing resources were made available from DOE, NSF, industry, and internationally. The Globus Compute platform played a key role in running an enormous bag of tasks computations across contributed resources. For example, it was used to extract features from 4.2 billion molecules for use in machine-learning workflows to screen therapeutics. The search space was broken into smaller batches of roughly 20 000 molecules per batch, and then the batches were executed across several supercomputers to create a canonical simplified molecular-input line-entry system (SMILES), compute molecular descriptors, render 2D figures, and compute molecular fingerprints. The processing produced a 60TB dataset that has been subsequently used for various analyses [13].

Such workloads are typified by a large number of tasks run for a short period of time (perhaps hours to days) with long periods of inactivity between work.

2.2.2. Federated Workflows

Scientific instruments produce massive volumes of data at very high velocities. Effectively utilizing such data requires online processing, often requiring extreme-scale resources, to provide rapid feedback to the experiment. Globus Compute has been used as a key component of automated workflows connecting various experiments at the Advanced Photon Source and the Argonne Leadership Computing Facility (ALCF). In prior work [14], we reported on experiences implementing production workflows for five different scientific instruments, each of which engaged powerful computers for data inversion, model training, or other purposes. The workflows were constructed using Globus Flows [15], enabling, for example, data to be transferred from an instrument to an HPC system when it was acquired, for various Globus Compute functions to be executed both at the edge and HPC for purposes such as model training, quality control, reconstruction, and metadata extraction; and for results to be shared with scientists via catalogs.

These workloads are typified by high arrival rates while experiments are progressing and long periods of downtime without experiments. Functions are used for many purposes; thus, they may run from seconds to hours.

2.2.3. Computation as a Service

The complexity of modern machine learning (ML) models, both in terms of computational resources needed for training as well as the complex environments required for inference, have made it intractable for many researchers. The inference-as-a-service paradigm aims to democratize access to ML models by deploying a model as a service and enabling on-demand inference. Globus Compute provides an ideal platform on which to build such services. Endpoints can be deployed on specialized and scalable computers, pre-configured with suitable environments (e.g., via Conda or containers). ML models can then be registered as functions and called by users to run inference against input arguments. The Data and Learning Hub (DLHub) [16], which enables the publication of ML models and on-demand inference relies on Globus Compute endpoints deployed on a Kubernetes cluster. Similarly, researchers at ALCF deployed AlphaFold [17], a deep learning system that predicts protein structures, as a service on the Polaris supercomputer [15].

These workloads are typified by sporadic usage. Model inference tasks typically run for short periods of time and are often involved across a wide range of input parameters.

3. The Globus Compute Dataset

The dataset[2] was gathered from the Globus Compute platform starting from November 28th, 2022, 00:00:00 UTC (inclusive) until July 3rd, 2023, 00:00:00 UTC (exclusive). It

[2]The dataset is publicly available: `https://doi.org/10.5281/zenodo.10044780`

contains 2 121 472 task submissions from 252 users executed across 580 geographically distributed endpoints. Moreover, the dataset includes 277 386 registered functions, among which we found 3026 unique serialized function bodies. Due to challenges with the serialization method (and changes over the period of the dataset), we were able to deserialize 2473 function bodies, of which 1847 are unique after deserialization. Please note that we do not differentiate between failed and successful tasks. We also do not include function bodies in the publicly released dataset for privacy reasons.

To avoid confusion in the following sections, we describe the relationships between different terms as depicted in Figure 2. We use **function** to refer to the registered Python code (including the input signature and **function body**), **task** to refer to an instance of a running function (an invocation), and **endpoint** to refer to the location on which a task is executed.

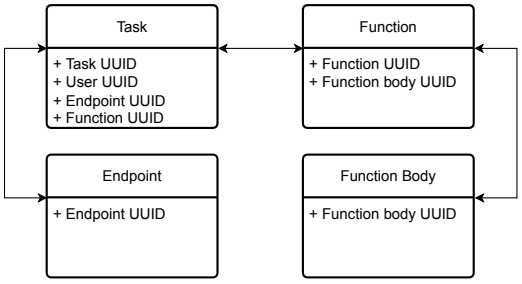

Figure 2: Overview of the relationship between terms.

3.1. Data Collection

We collected data from the cloud-hosted Globus Compute platform and by reconciling logs. Specifically, Endpoint and function data are stored in a relational cloud-hosted database. We extracted a subset of the data related to endpoints and functions used during the recording period.

Globus Compute captures and records timestamps as tasks transition through the system. The complete set of transitions is depicted in Figure 3 and further described in Table 1. The initial time that a task is submitted is recorded when the request is first submitted to the Globus Compute web service. Subsequent task lifecycle transitions, such as when the task is delivered to an endpoint and when execution is performed by a worker process, are captured and returned back to the web service to be recorded. Logs are stored in Amazon CloudWatch and are exported in the format described in this paper.

3.1.1. Data Processing

We describe how we deserialized the function bodies and obtained geo-locations for endpoints.

Preprocessing. We preprocessed the dataset to remove functions run as part of automated continuous integration testing by the Globus Compute team. We filtered the endpoint and function data by the IDs reported in the task trace so as only to report endpoints and functions used during this period.

Function deserialization. Globus Copmute records serialized function bodies in a central database. However, we found that

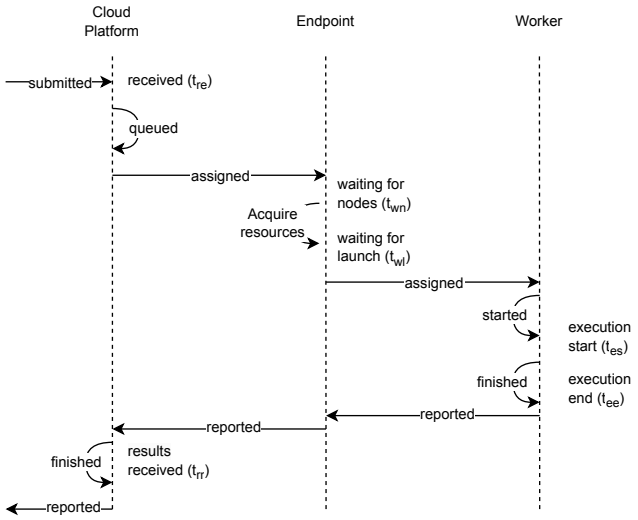

Figure 3: Sequence diagram of the flow of a submitted task.

the way functions are serialized differs based on the environment in which it was defined. For functions defined in local Python interpreters, we can obtain a string that represents the raw Python file. We were able to obtain unique source code for 1554 of 1578 functions defined in this way.

For functions defined in Jupyter notebooks, we have only the function object and the `__code__` attribute representing the compiled function body. In this case, we used the metadata to create a pyc file. We were able to recover 1208 of 1448 pyc. The most common reason for failures was when the function imported private libraries that were not accessible to us. We then decompiled the byte code to source code and discarded files with syntax errors.

We were able to recover 919 source files from the byte code. Following this process, many import statements are included on a single line. We used the *isort* Python utility to separate the import statements. After deserialization of both function types, we remove duplicates (using *cloc*) to obtain 1847 unique function bodies.

Geo-locations. Globus Compute records IP addresses for endpoints. Of the 580 endpoints included in the dataset, we obtained IP addresses for 428. We used the GeoLite2 city database [18] to map IP addresses to locations. While the mapping is accurate in many cases, for some IP addresses without city-level information, the location is stated as the center of the country.

Anonymization. To preserve anonymity, we obscure the data by hashing all identifiers using HMAC-SHA256 with secret salts. We apply a different salt to each column but ensure that identifiers (function, endpoint, user) are consistent across the dataset. This allows us to maintain the relationships between the data while removing any identifiable data.

Summary. The dataset comprises information about (i) the function invocations (see Table 1), (ii) the functions (see Table 2), and (iii) the endpoints (see Table 3). The UTC timestamps in the function invocation dataset are recorded in nanoseconds allowing fine-grained insights.

Characteristic	Description
Task uuid	The id of the submitted task.
User uuid	The id of the user who submitted the task.
Endpoint uuid	The id of the endpoint where the task is executed.
Function uuid	The id of the invoked function.
Argument size	The Byte size of the (black-box) input.
Received (t_{re})	The UTC timestamp when the task was received by the cloud platform.
Waiting for nodes (t_{wn})	The UTC timestamp the task was received by an endpoint and is queued for execution.
Waiting for launch (t_{wl})	The UTC timestamp the task was assigned and was waiting to be started.
Execution start (t_{es})	The UTC timestamp when the task execution started.
Execution end (t_{ee})	The UTC timestamp when the task execution finished.
Result received (t_{rr})	The UTC timestamp when the results were returned to the web service.

Table 1: Description of tasks (function invocations) dataset (N = 2 121 472).

Characteristic	Description
Function uuid	The id of the function.
# Lines of code	The number of source code lines.
Cyclomatic complexity	The calculated complexity.
# Imported libraries	The number of imported libraries.

Table 2: Description of functions dataset (N = 277 386).

Characteristic	Description
Endpoint uuid	The id of the endpoint.
Type	The type of provider of the endpoint (e.g., HPC scheduler).
Endpoint version	The version of the endpoint.

Table 3: Description of endpoints dataset (N = 580).

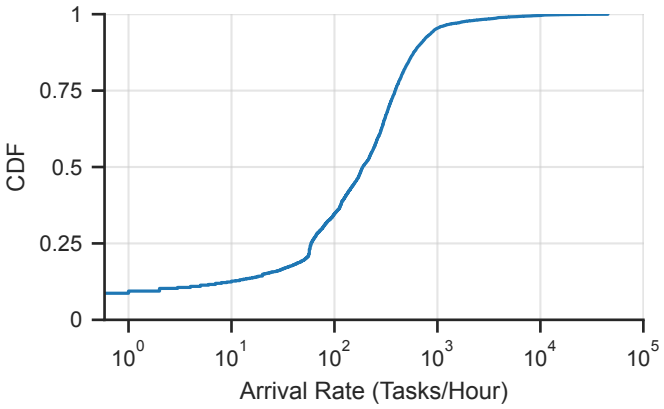

Figure 4: Distribution of the task arrival rate per hour with the x-axis in log-scale.

4. Statistical Analysis

To provide a high-level overview of the dataset, we report descriptive statistical measures for the system performance, interarrival times, task invocations, function bodies, and user behavior in Table 4. As central tendency, we report the mean and the median. As measures of variability, we report the standard deviation and the range.

4.1. System Performance

Figure 4 shows the cumulative distribution function (CDF) of submitted tasks per hour. The average task submission rate per hour was 404.31. In 25%, 50%, and 75% of the recorded time, the Globus Compute platform received at most 59, 179, and 381 tasks per hour, respectively. The highest rate observed was 4500 tasks per hour, which occurred on April 25th, 2023, coinciding with the annual Globus World conference in which the transition from funcX to Globus Compute was announced and Globus Compute was added to the Globus website. In contrast, during 8% of the recorded time, the Globus Compute platform did not receive any task submissions. The time-dependent behavior of the task submissions is discussed in Section 5.1.

Changing the perspective from the overall system to the individual endpoints, we investigate each endpoint's average task submission rate. The endpoints had to handle an average submission rate of 110.75 tasks per hour. 25%, 50%, 75%, and 90% of the endpoints received an average of 0.80, 2.03, 10.24, and 66.85 tasks per hour, respectively. The most used endpoint had to serve an average invocation rate of 8110 tasks per hour. Please note, unlike the Globus Compute platform, an endpoint can be started and shut down arbitrarily by users and thus can have long periods of unavailability. Therefore, an endpoint's minimum average tasks per hour was just 0.33.

The CDF of end-to-end times (from task *received* to *results received*) is displayed in Figure 5. The average end-to-end time for a submitted task was 23 minutes, with a median time of 0.34 seconds. Additionally, 59% of all submitted tasks were completed in less than 1 second, while 80% finished in less than 10 seconds. Moreover, 85% of all tasks were completed in less than 1 minute. The peak time of 1.17 million seconds is due to one task waiting to be delivered to an endpoint that was offline at the time. The user must have started the endpoint approximately 13 days and 14 hours after the task was submitted. In the subsequent subsection, we examine the different states a task goes through to understand the breakdown of the end-to-

Characteristic	Central Tendency		Measure of Variability	
	Mean	Median	SD	Range
System performance				
Arrival rate [req/h]	404.31	179.00	1.46e+03	[0e+00; 4.53e+04]
Avg. arrival rate per endpoint [req/h]	110.75	2.03	634.45	[0.33; 8.11e+03]
End-to-end time [s]	1.36e+03	0.34	1.66e+04	[1.57e-03; 1.17e+06]
Interarrival times				
Received (t_{re}) → Wait for node (t_{wn}) [s]	414.83	0.10	1.48e+04	[1.02e-06; 1.17e+06]
Wait for node (t_{wn}) → Wait for launch (t_{wl}) [s]	260.58	7.23e-03	1.88e+03	[1.77e-04; 1.31e+05]
Wait for launch (t_{wl}) → Execution starts (t_{es}) [s]	298.89	9.02e-03	2.08e+03	[4.91e-04; 1.31e+05]
Execution starts (t_{es}) → Execution ends (t_{ee}) [s]	49.04	0.03	300.37	[7.6e-05; 1.04e+05]
Execution ends (t_{ee}) → Results received (t_{rr}) [s]	5.42	0.13	51.13	[3.74e-05; 4.9e+04]
Tasks				
Avg. function idle time [s]	2.13e+03	61.38	5.55e+04	[5.12e-06; 5.44e+06]
Argument size [Bytes]	1.73e+04	62.00	2.14e+05	[30.00; 1.03e+07]
Function Bodies				
# Lines of code	35.68	48.00	29.47	[1.00; 467.00]
Cyclomatic complexity	5.91	6.00	3.96	[1.00; 20.00]
Imported libraries	1.50	1.00	1.52	[0.00; 18]
Users				
Avg. task submission interval [s]	1.89e+05	3.25e+03	8.1e+05	[2.67e-06; 7.48e+06]
# Tasks submitted	8.42e+03	22.50	5.12e+04	[1.00; 6.78e+05]
# Functions submitted	1.1e+03	7.50	1.07e+04	[1.00; 1.23e+05]
# Used endpoints	3.08	1.00	4.56	[1.00; 29.00]

Table 4: Descriptive statistical measures of the dataset.

end time.

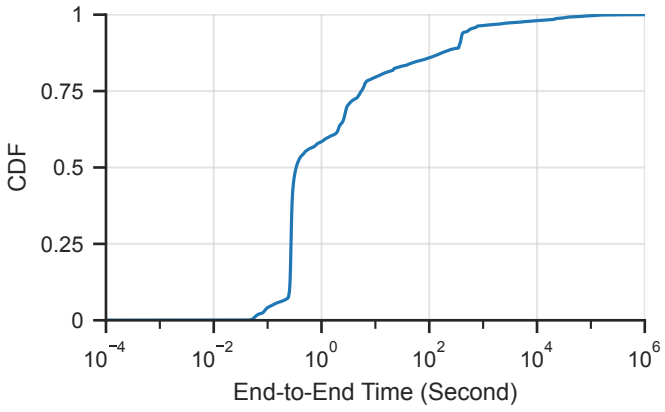

Figure 5: Distribution of the task end-to-end time of all submitted tasks in seconds with the x-axis in log-scale.

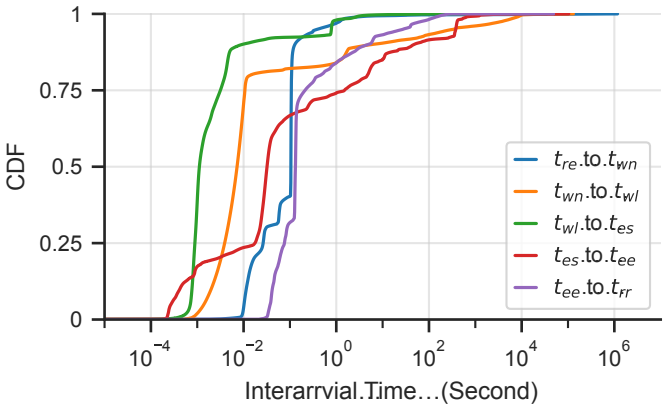

Figure 6: Distributions of the five interarrival times in seconds with the x-axis in log-scale.

4.2. *Interarrival times*

Figure 6 shows five CDFs representing interarrival times. As expected, we see significant differences between the different times matching the different stages of the task lifecycle. The blue curve illustrates the CDF of the time between the states $t_{re} \rightarrow t_{wn}$. On average, a submitted task had to wait 414.83 seconds to be queued for execution. However, for 40% of the submitted tasks, the waiting time was less than 0.1 seconds. Further, 97% and 99% of submitted tasks experienced a waiting time of less than 1 second and 1 minute, respectively. Similar to the end-to-end time (see Section 4.1), the longest waiting time observed was 1.17 million seconds (13.5 days).

Times between the states $t_{wn} \rightarrow t_{wl}$ and $t_{wl} \rightarrow t_{es}$ are shown as CDFs by the orange and green curves in Figure 6, respectively. The transition time from the queue to being assigned and waiting for execution was 260.58 seconds on average. In 70%

of cases, it took less than 0.01 seconds for the transition. Similarly, for 84% of tasks, this transition occurred within 1 second, and for 92% of tasks, it happened within 1 minute. Afterward, a task was assigned for, on average, 298.89 seconds until the execution started. For 39% of the submitted tasks, this time was less than 0.001 seconds. Moreover, 92% and 98% of tasks were started within 0.1 seconds and 1 second, respectively.

In Figure 6, the CDF of the execution time of a submitted task, $t_{es} \rightarrow t_{ee}$, is represented as red curve. The average execution time for a task was 49.04 seconds, with a median value of 0.03 seconds. Additionally, 74% of submitted tasks had an execution time of less than 1 second. Our findings align with similar studies. For instance, the median of the average execution time in the Azure dataset is below one second [3], and another investigation reported a median execution time of 0.06 seconds [19]. Among the submitted tasks that ran for less than 1 second, 43% were submitted by a single user. Notably, nearly all of this user's tasks were interacting with the batch scheduler, involving actions such as job submissions to the cluster or monitoring the status of submitted jobs. Additionally, 90% and 99% of submitted tasks were completed within 1 minute and 1 hour, respectively. Tasks running for more than 1 hour were submitted by only 4.4% of the total users. Among these users, two were responsible for 90% of these long-running tasks.

The purple curve in Figure 6 depicts the CDF of the time between the states $t_{ee} \rightarrow t_{rr}$. On average, the time taken to return the results to the web service was 5.42 seconds. However, for 32% of the submitted tasks, the time taken to return the results to the web service was less than 0.1 seconds. Furthermore, 84% and 93% of tasks received their results within 1 second and 10 seconds, respectively.

4.3. Task Invocations

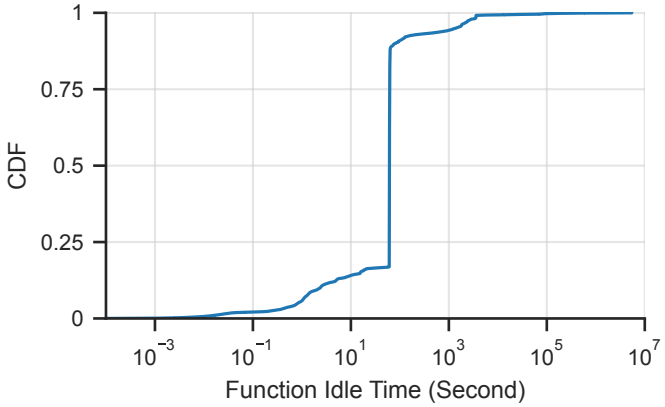

Figure 7: Distribution of the average function idle time in seconds with the x-axis in log-scale.

As noted in Section 3, a single function may be executed many times. To this end, we are interested in the average function idle time, which represents the time between the end of one function's execution and its next call. The average function idle time is displayed as CDF in Figure 7. On average, a function was called approximately every 35 minutes after its execution was finished. In contrast, 50% of functions were called again in less than 61.38 seconds. Furthermore, 6%, 17%, and 99% of functions were repeatedly called after execution less than every second, minute, and hour, respectively. Notably, by implementing a keep-alive or warming time of 5 minutes, the cold start for 93% of functions can be avoided.

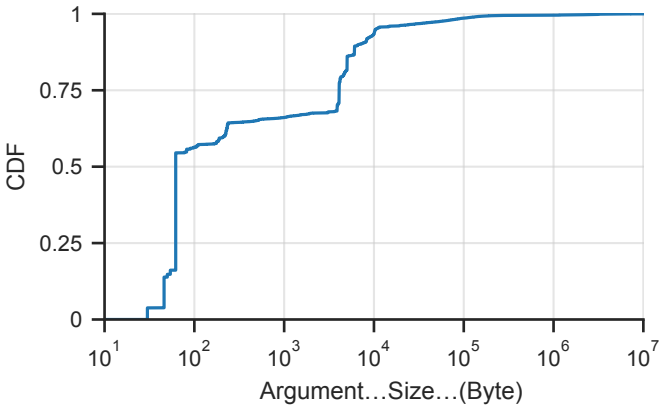

Figure 8: Distribution of the task argument size of all submitted functions in Bytes with the x-axis in log-scale.

The CDF of the size of arguments for submitted tasks is depicted in Figure 8. The average input size for a submitted task was 17 Kilobytes. Moreover, 50%, 75%, and 99% of submitted tasks had an input less than 62 Bytes, 4 Kilobytes, and 132 Kilobytes, respectively. Tasks with argument sizes exceeding 5 Megabytes were submitted by 4.7% of the users and accounted for 0.03% of all submitted tasks. Of these 12 users, two were responsible for 86% of these submissions. Please note that these two users are different from the two users with long-running tasks discussed previously.

4.4. Function Bodies

In this section, we report statistics about the function bodies. As we were not able to deserialize all function bodies, we focused on a subset of the data that maps to a deserialized function. More specifically, the subset includes 1 854 104 tasks, 268 476 functions, and 1847 function bodies.

The length of function bodies in our dataset varies, ranging from 1 to 467 lines of code (LoC). On average, a submitted task had 35.68 lines, and 50% of these tasks contained 48 LoC or fewer. Moreover, 96% of the tasks had a maximum of 54 LoC. Interestingly, 44% of submitted tasks had exactly 54 LoC. This is due to the repeated calls of a specific function. When focusing only on unique functions (268 476) instead of all submitted functions (1 854 104), the average function length reduced to 19.90 LoC, with 50% of functions having 15 LoC or fewer. When considering only unique function bodies (1847), the average LoC was 32.30, with a median of 13. The CDFs of LoC for all tasks, functions, and function bodies are displayed as the blue, orange, and green curves, respectively, in Figure 9.

To understand the workload of the Globus Compute platform, we analyze function complexity. The cyclomatic complexity of the functions, which represents the number of linearly

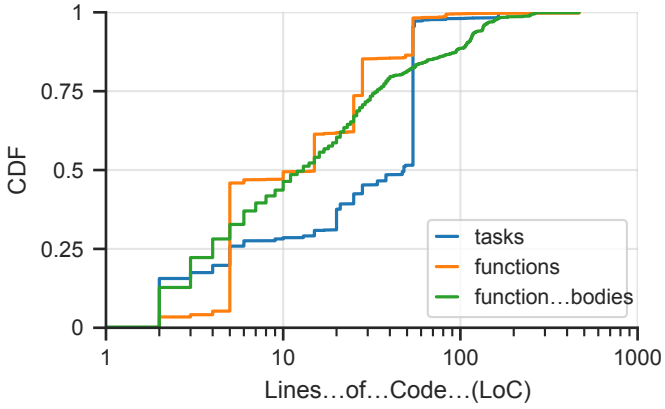

Figure 9: Distributions of lines of code with the x-axis in log-scale. The blue curve represents data from 1 854 104 submitted tasks, the orange curve from 268 476 functions, and the green curve from 1847 unique functions.

independent paths within the code, in our dataset ranges from simple (1-10) to more complex (11-20). Consequently, none of the functions are classified as complex (21-50) or untestable (>50) according to the given categorization [20]. On average, the complexity is 5.91, with 99% of submitted tasks (1 854 104) having a complexity of 10 or lower. The average complexity reduces to 3.32 and 2.97, with a median of 2 for unique functions (268 476) and 1 for unique function bodies (1847). The CDFs of the cyclomatic complexity for all submitted tasks, all unique functions, and all unique function bodies are shown as the blue, orange, and green curves, respectively, in Figure 10. Remarkably, 99% of the submitted tasks and 94% of function bodies tend to be more simple than complex, respectively. These results are corroborated by previous studies [21, 22].

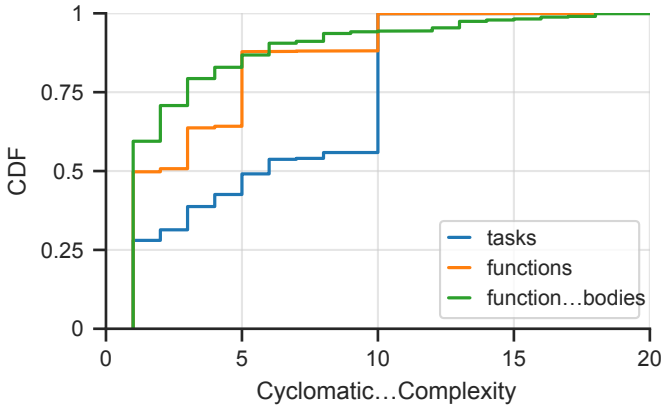

Figure 10: Distributions of the cyclomatic complexity. The blue curve represents data from 1 854 104 submitted tasks, the orange curve from 268 476 functions, and the green from 1847 unique functions.

Lastly, we are interested in the number of libraries used within a function. In our dataset, functions had between 0 and 18 imported libraries. A detailed investigation of the libraries is conducted in Section 5.8. On average, 1.50 libraries were imported, with 70% and 92% of submitted tasks (1 854 104) having 1 or fewer and 3 or fewer imported libraries, respectively. For unique functions (268 476) and unique function bodies (1847), the average number of imported libraries increases to 1.83 and 2.36, while the median is 2 for both cases. The CDFs of the number of imported libraries for all submitted tasks, all unique registered functions, and all unique function bodies are depicted as the blue, orange, and green curves, respectively, in Figure 11.

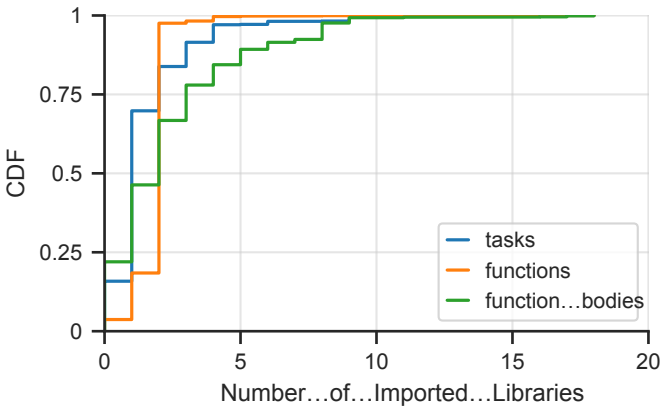

Figure 11: Distributions of the number of imported libraries. The blue curve represents data from 1 854 104 submitted tasks, the orange curve from 268 476 functions, and the green curve from 1847 unique functions.

4.5. User Behavior

After investigating the function idle time (see Section 4.3), we switch our focus to the user perspective. Here, we are interested in the user's "task submission interval," which represents the time between two task submissions by the same user. To this end, we calculated the average user task submission interval per user, and the corresponding CDF is displayed in Figure 12. On average, a user submitted a task approximately every 2 days and 4 hours. Further, 22%, 51%, 81%, and 96% of users submitted functions in less than a minute, hour, day, and week, respectively. Notably, eight users submitted functions at a rate exceeding 1 Hz.

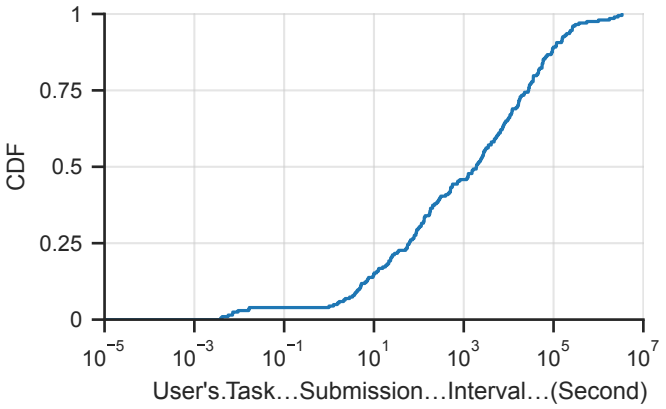

Figure 12: Distribution of user's task submission interval in seconds with the x-axis in log-scale.

Next, we are interested in the submission behavior of users. To this end, we investigate how many tasks (2 121 472) and

unique functions (277 386) were invoked by users. The CDFs of the number of tasks and unique registered functions submitted by each are depicted by the blue and orange curves in Figure 13, respectively. On average, a user submitted 8.42 thousand tasks. Among the users, 29% have submitted less than 10 tasks, while 65%, 85%, and 94% of the users have submitted fewer than 100, 1000, and 10 000 tasks, respectively. Only two users submitted more than 250 000 tasks. When considering the number of unique registered functions, a user submitted on average 1.1 thousand unique functions. While 15% of users have submitted only 1 unique function, 50% of users have submitted fewer than 7 unique functions. However, 65%, 85%, and 94% of the users have submitted 100, 1000, and 10 000 unique functions, respectively.

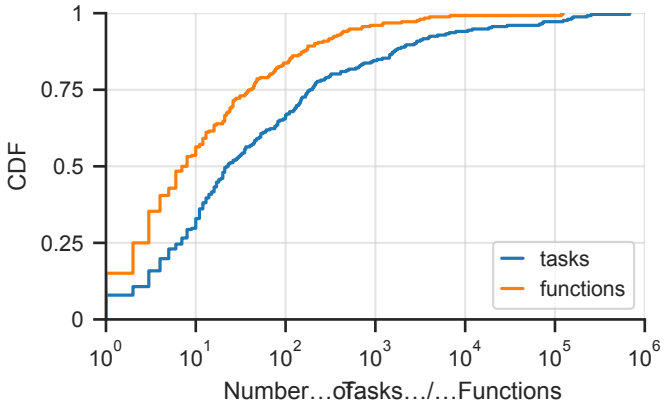

Figure 13: Distributions of the number of tasks per user (2 121 472) depicted as blue curve and the number of functions per user (277 386) as orange curve.

Finally, we analyze the number of endpoints used by each user. The corresponding CDF is shown in Figure 14. The average number of endpoints used per user is 3.08. More than half of users (54%) used only 1 endpoint for their computations. Moreover, 72%, 88%, and 94% of the users used 2, 5, and 10 endpoints or fewer, respectively. While 5 users used 24 endpoints or more, and 3 users used 25 endpoints or more.

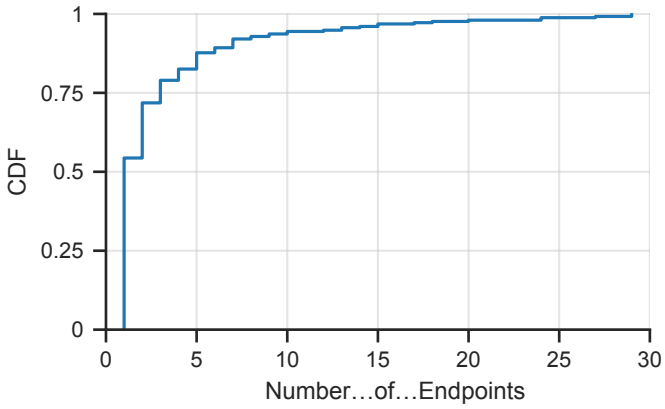

Figure 14: Distribution of the number of endpoints per user.

5. Detailed Analysis

We now delve deeper into specific topics of interest. Our focus in this section includes analyzing task submission patterns, investigating invocations per user and endpoint, examining the recurring functions and content of all functions and the libraries they import, as well as exploring the the type and location of endpoints.

5.1. Task Submission Behavior

To gain insight into the Globus Compute workload, we examine the submission patterns of tasks. While we have already discussed the statistical description in Section 4.1, we now focus on behavior over time. To this end, we present two visualizations: Figure 15 illustrates the task submissions per hour throughout the dataset's duration, while Figure 16 provides a clearer representation of the submitted task rates per day. Notably, we observe a consistent usage level across the dataset, with discernible declines around December 25th and January 1st, corresponding to Christmas Day and New Year's Day, respectively. We are unsure what caused the drop in usage around April 10th. Additionally, several peaks are noticeable, such as on April 24th, coinciding with the Globus World conference (see Section 4.1).

Examining the task submission behavior in more detail, we focus on the average task submission rate per hour and per weekday. Please note that we removed outliers beforehand (3-σ rule) so that events like Globus World do not distort the average. As shown in Figure 17, a distinct diurnal trend is evident in the average hourly invocation pattern. The task submission rate experiences fluctuations throughout the day, with notable peaks occurring during the morning and afternoon hours in the Central Daylight Time (CDT) zone. Particularly prominent peaks are observed at 10 am, 2 pm, and 4 pm CDT. After 10 pm, the invocation rate gradually decreases, reaching its minimum at 7 am CDT. This pattern suggests lower activity levels during the late-night and early-morning hours CDT. Figure 18 presents the average daily invocations. The data reveals a clear pattern where the task submission rates increase during the week, reaching its lowest point on Saturday. Subsequently, the invocation rate begins to increase again. This pattern is likely influenced by regular work schedules and typical usage patterns.

5.2. Task Invocations per User

As already touched on in Section 4.5, a small set of users contribute significantly to the overall number of task submissions. In order to delve deeper into this behavior, we conducted a more detailed investigation, and the results are presented in Figure 19. The CDFs for the most active users, displaying their contributions in terms of task submissions (blue curve), unique functions (green curve), and unique function bodies (green curve). Each dot on the lines represents a user, and for clarity, we plotted only the ten most active users.

Remarkably, the five most active users, or 1.9% of the total users, account for 69.35% of the task submissions. Expanding the scope to the top ten users, they are responsible for 91.70%

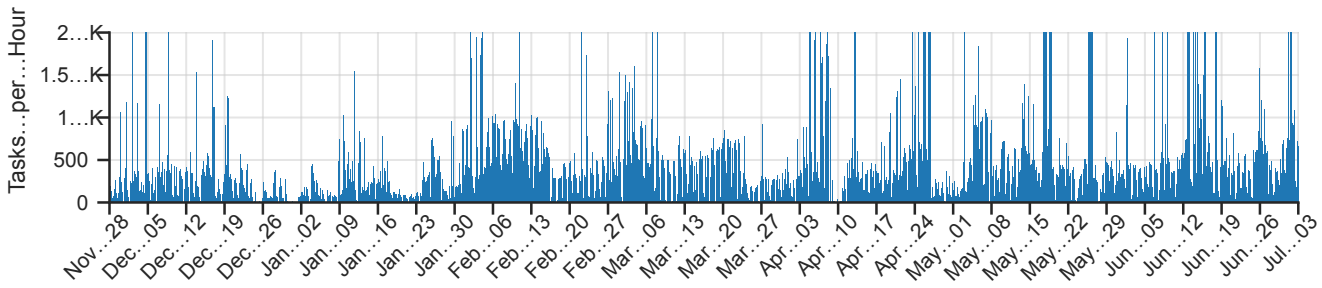

Figure 15: Task submissions per hour with values greater than 2000 truncated.

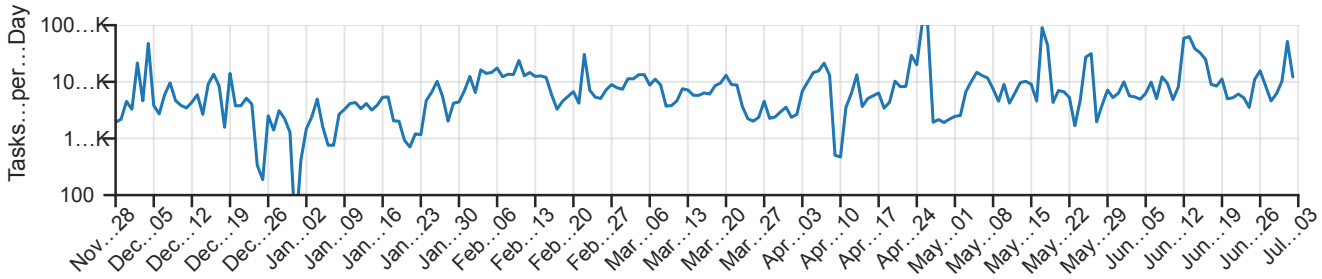

Figure 16: Task submissions per day with the maximum (107 890) and the minimum (20) truncated.

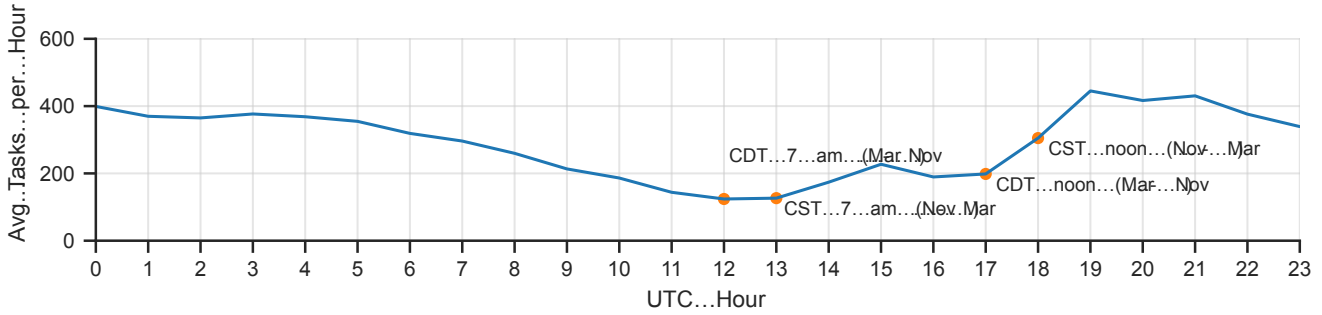

Figure 17: Average task submissions per UTC hour with outliers removed by 3-σ rule.

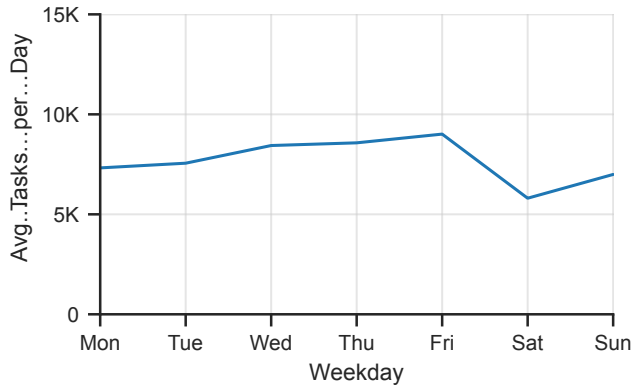

Figure 18: Average task submissions per weekday with outliers removed by 3-σ rule.

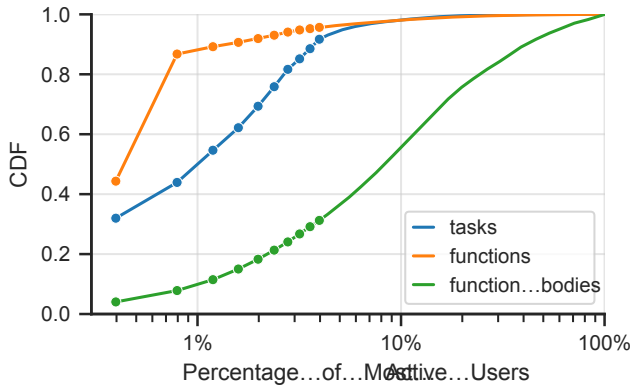

Figure 19: Distributions of task submissions per user in percentage. The blue curve represents data from 2 121 472 submitted tasks, the orange curve from 277 386 functions, and the green curve from 1847 unique function bodies.

of the submissions. These findings are consistent with previous studies [23, 3]. The trend of concentration becomes even more apparent when it comes to unique functions. The top five users contribute to 91.92% of all functions, while the top ten users are responsible for 95.67% of the total unique functions.

Examining the function bodies, we find that the top five users contribute 24.53% of all unique function bodies, and the top ten users contribute 41.91% of all unique function bodies. These figures underscore the significant impact of a few users. Inter-

estingly, the top ten users in the three categories (task submissions, functions, and function bodies) show some overlap but not complete consistency. This indicates different usage patterns among these users. While some users issue as many tasks as they have functions, others repeatedly invoke a few functions. Further, there are users who programmed many function bodies, but are not among the top users in terms of task submissions or functions. In summary, the user behavior can be broadly categorized into two groups: those who repeatedly execute the same or similar tasks and those who utilize various functions but less frequently.

5.3. Task Invocations per Endpoint

In Section 5.2, we observed that Globus Compute is predominantly used by a small group of power users. A similar trend can be seen in the usage of endpoints, as depicted in Figure 20. This figure displays a CDF of the most popular endpoints, with each dot representing an individual endpoint. For clarity, we have included only the top 10 popular endpoints. The most heavily used endpoint serves 32.01% of all submitted tasks. Expanding the scope to the top five endpoints, they collectively handle about 63.42% of all submitted tasks, while the top ten endpoints were used for 77.70% of all tasks.

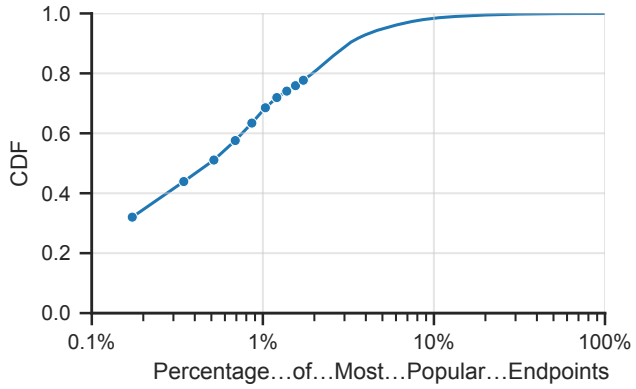

Figure 20: Distribution of most popular endpoints in percentage.

Although it is possible to share endpoints among users, it is uncommon as we carefully limit usage and require approval from administrators and users. As a result, only a small fraction, around 5%, of endpoints are shared with multiple users. Among these shared endpoints, Figure 21 presents the CDFs of the most frequent users accessing the most shared endpoints, where each dot corresponds to a user. For the sake of clarity, we show only the top 5 users each. The most widely shared endpoint (blue curve) is used by 92 different users, followed by 33 users (orange curve), 13 users (green curve), and 7 users (red curve). The most shared endpoint serves only 8214 tasks, equivalent to 0.39% of all submitted tasks. Within these invocations, the top-ranked user is responsible for 33.36% of all executions at this particular shared endpoint, while the top five users collectively account for 76.28% of these invocations. The second most shared endpoint receives 5895 invocations, with the most active user responsible for 45.99% of these tasks. The third most shared endpoint is used for 43 submitted tasks, and the fourth most shared endpoint handles ten submitted tasks.

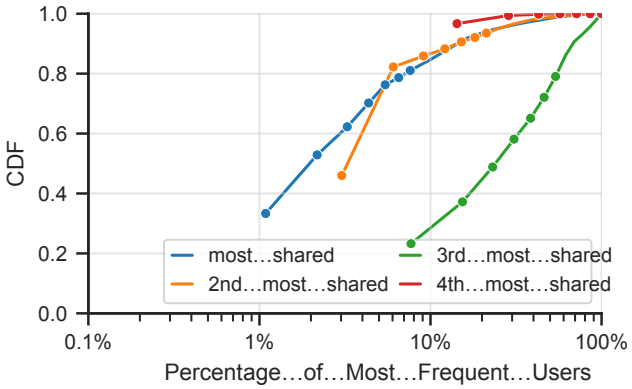

Figure 21: Distributions of the most frequent users in task submissions percentage, individually ranked in the four most shared endpoints, with the x-axis in log-scale. The most shared endpoint (blue curve) is used by 92 different users, the second most shared (orange curve) by 33, the third most shared (green curve) by 13, and the fourth most shared (red curve) by 7.

We reviewed the usage of all endpoints and identified several common endpoint "types." Figure 22 shows four endpoints that are representative of broader types in the data. The first (blue) shows *ephemeral* use, where the endpoint is used for a short period of time and never used again. The second (orange) shows *constant* use, likely supporting a regular workload with about the same number of task submissions per hour. The third (green) shows *sporadic* use with seemingly no pattern for usage every few days. The fourth shows a *campaign* style pattern in which an endpoint is used for a number of tasks over a few days and then not used again for several months.

5.4. Frequently Invoked Functions

We investigated the popularity of functions invoked by submitted tasks, taking into account that different tasks may invoke the same function, and different functions may have identical function bodies. Please note that we derived the results on the reduced dataset (1 854 104 tasks, 268 476, 1847 unique function bodies) as discussed in Section 4.3. Figure 23 provides the CDFs for the most frequently invoked functions based on unique function (orange curve) and unique function body (green curve). Each dot in the figure represents a function. For clarity, we include only the ten most popular functions.

The function with the most invocations (leftmost orange dot) was called 53 370 times, making up 2.5% of all task submissions (1 854 104 in total). The combined invocations of the top five most popular functions account for 11.31% of all submissions, while the top 10 most invoked functions (i.e., 0.04% of all functions) make up 18.65% of all task submissions. Consequently, the remaining 99.996% of functions constitute the remaining 81.34% of submissions. To be more specific, the top 1% of all 268 476 unique functions are responsible for 66.72% of all task submissions, while the top 10% account for 86.72%.

The unique function body that is most frequently invoked accounts for 816 027 task submissions, representing 44.01% of

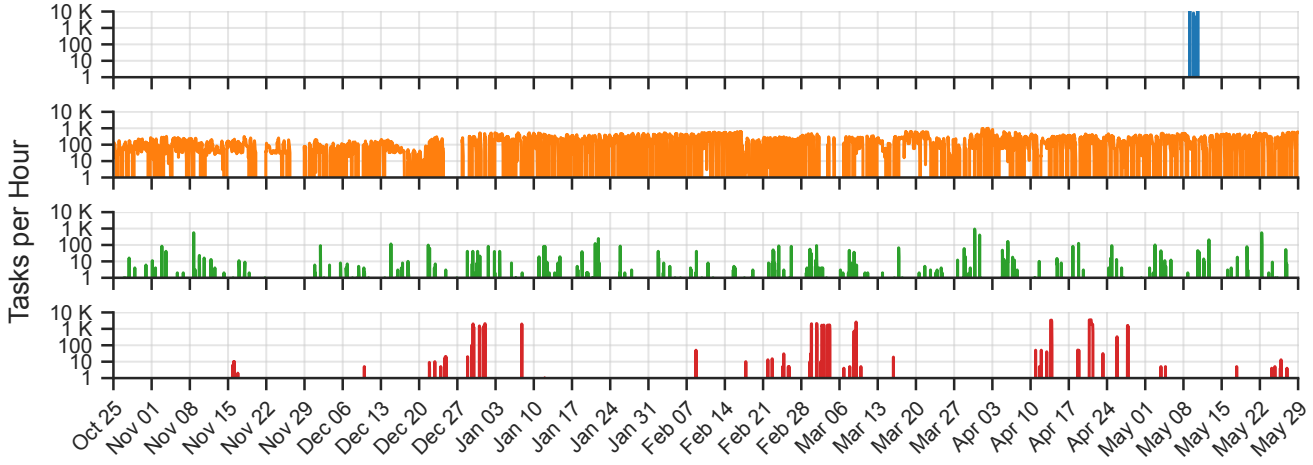

Figure 22: Task submissions per hour for four selected endpoints with each y-axis in log-scale.

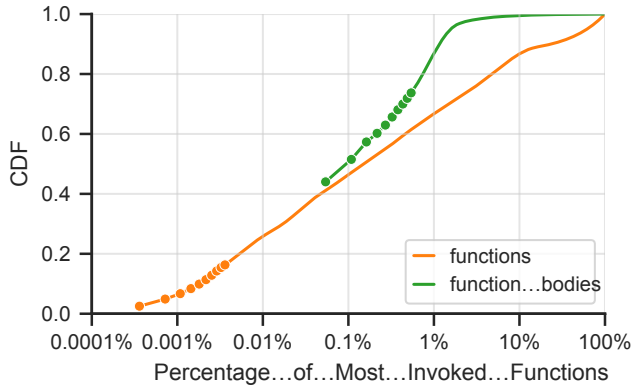

Figure 23: Distributions of task submissions in percentage, individually ranked by functions (268 476) and function bodies (1847).

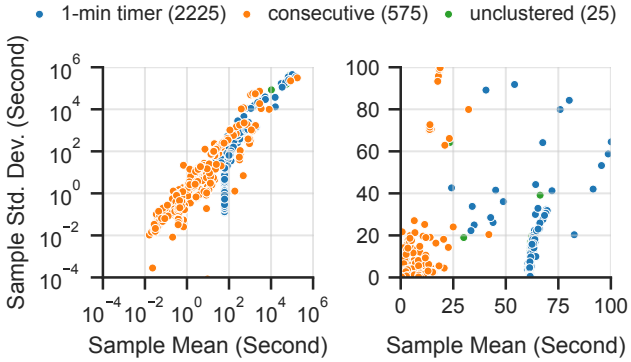

Figure 24: Characterization of the frequently-invoked functions with colors reflect their HDBSCAN clusters and legends.

all submissions. Cumulatively, the top five and top 10 function bodies are responsible for 62.94% and 73.70% of all invocations, respectively. Remarkably, 756 function bodies are invoked only once, which is depicted by the relatively flat portion of the green curve in Figure 23. Moreover, the top 10% of function bodies account for 99.51% of all submissions.

5.5. *Function Idle Time*

To explore similarities between functions in our dataset, we investigate the function idle time patterns of frequently invoked functions. Please recall that the function idle time is the time between the end of one function's execution and its next call. Moreover, we consider a function to be frequently invoked if it was called at least 50 times. 2825 of 277 386 functions meet this criteria. We then computed the Kolmogorov-Smirnov statistic to compare the function idle time distributions for each pair of functions, resulting in a dissimilarity matrix of size 2825×2825. To facilitate visualization and density-based clustering with HDBSCAN [24], we applied principal component analysis to reduce the matrix dimensionality to 2825×2. After fine-tuning the HDBSCAN hyperparameters, we identified two meaningful clusters alongside an unclustered set in the reduced embedding space.

The clustering result is shown in Figure 24. Each point corresponds to one of the 2825 functions. The color of a function represents the cluster, while the x and y coordinates represent the sample mean and standard deviation of its idle time. In other words, the x-value indicates how long it typically takes for the function to be invoked again, while the y-value shows the variation until its next invocation. The left plot shows both axes in seconds with a logarithmic scale. Since 2371 out of the 2825 functions fall within the range of $(x, y) \in [0; 100] \times [0; 100]$, the right plot is non-scaled and limited to 100 seconds on both axes to provide a more focused view.

Upon analyzing the left plot, a significant amount of functions (1825 out of 2825) emerge within the range of $60 \leq x \leq 65$, indicating these functions exhibit a consistent invocation pattern with an interval of approximately one minute. When examining the zoomed-in plot, most of these functions (1818 out of 1825) can be found with $(x, y) \in [60; 65] \times [0; 20]$. 1806 of these functions belong to the blue cluster that we identified as **one-minute timer tasks**, which exhibit varying sample standard deviations but relatively consistent sample means. In total, the blue cluster contains 78.76% of the frequently-invoked

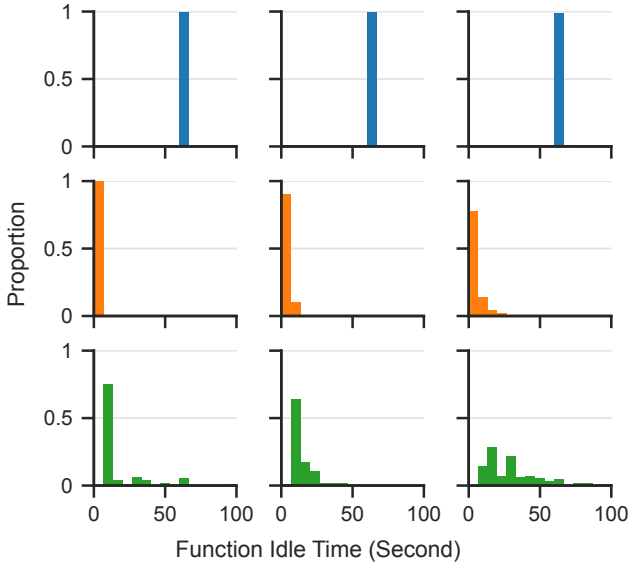

Figure 25: Nine selected sample functions showing idle time distributions, one row per cluster and the last row representing unclustered distributions.

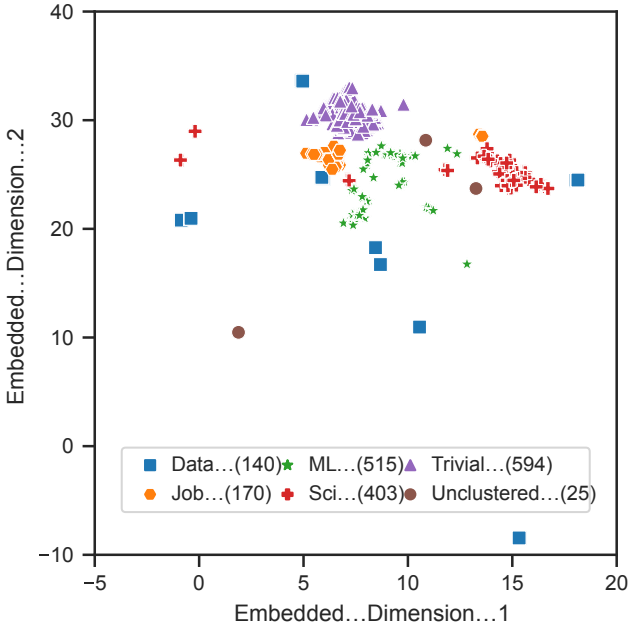

Figure 26: HDBSCAN clustering of the embedding semantics vectors.

functions, although occupying a smaller area in both plots.

Another concentration of functions (478 orange dots) is located within $(x, y) \in [0; 20] \times [0; 20]$, suggesting these functions are invoked consecutively with low and almost constant idle times. In total, there are 575 functions in the orange cluster, which we refer to as **consecutive tasks** as they are submitted consecutively with a sample mean of less than a minute and varying sample standard deviations. The remaining 25 functions are labeled as **unclustered** and exhibit relatively high sample means and sample standard deviations. Please note that as the HDBSCAN clustering is based on the function idle time distributions, functions with a similar mean and deviation can belong to different clusters.

We also visualize representative CDFs of samples from each cluster in Figure 25. By comparing these plots with the plots of the Azure dataset [3], we find that our function idle time distributions are comparable. For example, most functions in the blue cluster also exhibit a low number of bins, and the functions from the orange cluster follow a right-skewed distribution. The difference is that we calculate the idle time on a function level while they perform it on an application level.

5.6. Analysis of Function Bodies

We proceed to explore the types of functions used on the Globus Compute platform. For this purpose, we cluster the 1847 unique functions, focusing on their source code. Specifically, we use an embedding model from OpenAI [25], which transforms each function into a 1536-dimensional array with values ranging from -1 to 1, capturing various semantic and syntactic aspects of the code. Before we applied HBSCAN clustering, we reduced the dimensionality from 1847×1536 to 1847×2 with UMAP [26].

The clustering identified 29 meaningful clusters and one unclustered set. We manually assigned one of five labels to each of these 29 clusters, indicating the function type for most, if not all, of the functions in each group. The labels include **Data Manipulation** (Data), **Job Scheduling and Program Entry** (Job), **Machine Learning** (ML), **Non-ML Scientific Computing** (Sci), and **Trivial Program** (Trivial). Clusters with the same label were merged, resulting in five clusters along with the unclustered functions, which are visualized in Figure 26.

The Data Manipulation cluster is the smallest cluster and contains 140 function bodies. Functions in this cluster transmit or retrieve data from online platforms (like Globus) and perform local data manipulation tasks. As shown in Figure 26, the eight groups (blue squares) in this cluster found by HBSCAN are relatively distant from others, indicating that data manipulation tasks are written very differently based on the nature of the tasks and user preferences. The Job Scheduling and Program Entry cluster comprises 170 function bodies, primarily involving functions related to sending or querying HPC jobs or starting other Python or shell programs. They appear in two groups (orange hexagons) found by HBSCAN. The small group on the top right in Figure 26 comprises the uses of `qsub` commands, while the large group on the left contains the uses of `psij` commands [27], `mpirun` commands, and Python, shell, and binary scripts. The machine learning cluster includes 515 function bodies and is spread across twelve groups (green stars). Each group represents one or more ML algorithms or strategies, such as graph neural networks, n-nearest neighbors, long short-term memory networks, federated learning, AlphaFold, etc. The Non-ML Scientific Computing cluster has 403 function bodies and involves functions related to data collection, processing, and analysis for scientific data (e.g., from simulations or instruments). They are aggregated from six groups (red crosses) found by HBSCAN and are relatively close to each other, as shown in the lower right portion of Figure 26. The

largest cluster is the Trivial Program cluster, with 594 function bodies. The functions are either minimal function bodies used by tutorial users or individuals exploring Globus Compute. These trivial programs (purple triangles) appear in one extremely large group found by HBSCAN. Lastly, 25 function bodies (brown circles) remain unclustered by HBSCAN, indicating their unique patterns or functionalities that were not captured within the clustering analysis.

After we clustered the different function bodies, we investigated how different groups of functions vary in code and performance measures. We report the median, minimum value, and maximum value for each measure in Table 5. The code-related measures consist of Lines of Code (LoC), cyclomatic complexity, and the number of imported libraries, all of which are determined from the function bodies. The performance measures encompass the argument size, end-to-end time, and function idle time, which are derived from tasks.

When examining the code-related measures, we observed that the Trivial Program cluster comprises the shortest function bodies, with the lowest complexity and the fewest imported libraries. The Job Scheduling and Program Entry cluster follows closely, also having relatively short and simple function bodies with few imports. In contrast, functions from the Machine Learning cluster and Scientific Computing cluster have longer function bodies compared to the other clusters. Further, the Machine Learning cluster functions display higher complexity compared to others. Interestingly, the Data Manipulation cluster's median function has the second-highest median complexity, but its functions have the second-lowest maximum complexity. Regarding the number of imported libraries, functions from the Machine Learning, Scientific Computing, and Data Manipulation clusters import the most libraries, with the Machine Learning cluster having the most imports.

Regarding performance-related measures, functions from the Data Manipulation cluster and Scientific Computing cluster receive the highest median input sizes, although they do not have the highest maximal argument size. On the other hand, the Trivial Program cluster contains functions with the lowest median input size, followed by functions from the Job Scheduling and Program Entry cluster. Functions from the Job Scheduling and Program Entry cluster and Machine Learning cluster exhibit the shortest median end-to-end times, indicating for the latter cluster a higher likelihood of being used for inference rather than training. In contrast, functions from the Data Manipulation cluster have the longest median end-to-end times. Interestingly, functions from the Trivial Program cluster have the longest maximal end-to-end times and the third-longest median end-to-end times. Moreover, functions from the Trivial Program and Machine Learning clusters are called much more frequently than those of the other clusters. On the other hand, functions from the Job Scheduling and Program Entry cluster are the least frequently called.

5.7. Sub-Processes and External Function Calls

In order to understand the function usage better, we investigated what code is run within the functions by manually examining function bodies. Of the 1847 unique function bodies, 297 (i.e., 16%) functions call sub-processes or external scripts. We classified the type of external target and reported the results in Table 6. The most frequent usage is to call shell scripts (36.71%), followed by running binary programs (33.23%). Also, 4.43% of the calls run another Python script. We hypothesize that users are using Globus Compute to run various external programs for several reasons: (i) they may have existing code bases; (ii) Globus Compute supports only Python functions; or (iii) Scientific computing applications are often written in low-level languages and use higher level languages for coordination. We expect that this behavior is unlike other FaaS systems.

5.8. Imported Libraries

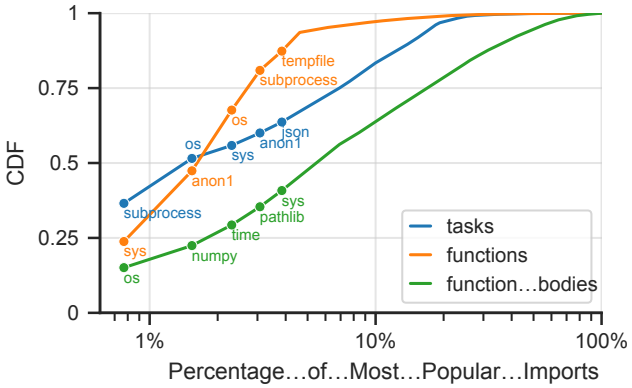

Figure 27: Distributions of most popular imported libraries in percentage. The blue curve represents data from 1 854 104 submitted tasks, the orange curve from 268 476 functions, and the green curve from 1847 unique functions.

After examining the statistical distribution of important libraries in Section 4.3, we now investigate the most frequently used libraries (see Figure 27). As mentioned in Section 4.3, we consider for this analysis also the reduced dataset (1 854 104 tasks, 268 476 functions, 1847 unique function bodies). In total, 130 different Python libraries were imported. To protect privacy, we requested permission from users to report packages that are not available on PyPI. In some cases, we anonymized the names as `anon1`, `anon2`, etc. Among the top libraries used in all submitted tasks, `subprocess` stands out, followed by `os`, `sys`, `anon1`, and `json`. Collectively, these five libraries were imported a total of 1 773 211 times, accounting for 64% of all imports. When considering only unique functions, the frequently employed libraries are `sys`, `anon1`, `os`, `subprocess`, and `tempfile`. These libraries were imported together 428 965 times, which represents 87% of all imports. For unique function bodies, the most commonly used libraries are `os`, `numpy`, `time`, `pathlib`, and `sys`. These libraries were imported 1780 times and account for 41% of all imports.

5.9. Endpoint Provider Types

Globus Compute implements a unique model in which the endpoints can be deployed on heterogeneous resources, ranging from supercomputers and clouds to edge devices. To accommodate the differences between these systems, Globus Compute

Function Category	Lines of Code	Cyclomatic Complexity	Number of Imports	Argument Size (Byte)	End-to-End Time (Second)	Function Idle Time (Second)
Data Manipulation	23 [4; 225]	2 [1; 8]	3 [1; 9]	4089.00 [30.00; 6.75e+06]	378.97 [0.01; 3.2e+05]	8.69 [2.30e-06; 1.37e+06]
Job Scheduling	15 [2; 125]	1 [1; 20]	2 [1; 6]	62.00 [30.00; 1.03e+07]	0.28 [0.02; 1.79e+05]	61.48 [3.53e-04; 8.65e+04]
Machine Learning	32 [2; 145]	3 [1; 18]	3 [1; 12]	111.00 [30.00; 1.02e+07]	1.41 [0.03; 9.29e+04]	0.01 [2.89e-05; 366.90]
Sci. Computing	25 [2; 467]	1 [1; 18]	2 [1; 18]	4186.00 [30.00; 1.39e+06]	4.73 [0.03; 5.94e+05]	4.00 [1.56e-05; 2.87e+06]
Trivial Program	3 [3; 37]	1 [1; 6]	1 [1; 4]	46.00 [30.00; 5.71e+06]	2.75 [0.01; 1.17e+06]	0.02 [2.56e-07; 3.1e+06]

Table 5: Selected Statistics Per Function Cluster.

Call	Distribution
Shell scripts	36.71%
Binary programs	33.23%
Usage of echo	20.25%
Misc. shell commands	5.38%
Python scripts	4.43%

Table 6: Overview of sub-processes or external calls.

leverages Parsl [10] to support various systems. When confi uring a Globus Compute endpoint, the user can select and co figure an appropriate provider to enable the endpoint to acqui manage, and release resources. Table 7 provides a summary the providers used by endpoints included in the dataset. T most common endpoint provider type is a local provider wi 74.50%. The local provider is used by an endpoint to provisi resources on a single node to perform work by spawning a loc process to perform a function. It is deployed, for example, an edge device, laptop, cloud instance, or even a login node an HPC system. The second most used type is Slurm (10.50 followed by Kubernetes (7.75%). The least used type is L with 0.25%. These results show the vast majority of endpoints are configured to utilize a single node and manage execution on multicore processors. The Slurm and Kubernetes endpoints are likely to be significantly more computationally powerful as they are provisioning compute nodes from a cluster.

Provider	Distribution
Local	75.55%
Slurm	10.50%
Kubernetes	7.75%
PBSPro	5.00%
Cobalt	1.25%
LSF	0.25%

Table 7: Use of the different provider types on the endpoints.

5.10. Geographical Distribution of Endpoints

A key feature of Globus Compute is its federated architecture, which encompasses endpoints deployed on heterogeneous resources worldwide. Figure 28 displays the number of endpoints deployed at different locations. The locations were determined based on the IP addresses. In North America, endpoints are located in the United States, Mexico, and Canada. In Europe, endpoints can be found in Germany, Norway, Switzerland, Finland, Romania, the United Kingdom, and Spain. In Asia, endpoints are present in India, China, and Taiwan. Mean-

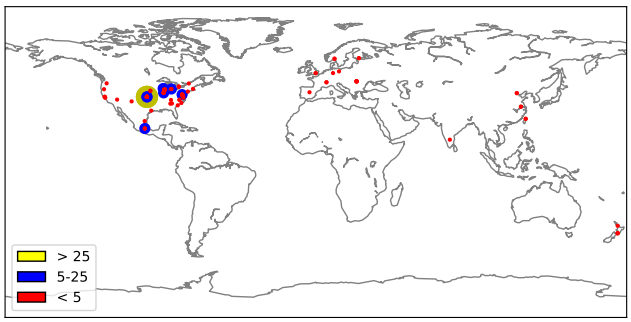

Figure 28: Worldwide distribution of Globus Compute endpoints. Colors indicate the number of endpoints in any single location.

6. Related Work

Prior research has explored various aspects of the serverless paradigm. We consider three categories relevant to our work: (i) examination and release of serverless (trace) datasets, (ii) benchmarking and measuring of serverless applications or systems, and (iii) characterization of serverless platforms, applications, or the paradigm itself.

6.1. Serverless Trace Datasets

To the best of our knowledge, there are few publicly accessible serverless (trace) datasets available, and most are provided by cloud providers like Azure and Alibaba. For instance, Azure's dataset from 2019 [3] is one of the most frequently referenced, containing 12.5 billion invocations over 14 days, organized based on the number of invocations per minute per function. This dataset also includes statistics on execution times, such as averages and medians, reported in per-day intervals per function. Researchers have used this dataset extensively for performance analysis [29, 30, 31, 32], cold-start optimization [33, 7], synthetic trace generation [34], and to characterize the behavior of serverless applications [35].

In a more recent study, Azure released another dataset [28], also covering 14 days of serverless invocations, but instead of

Dataset	Duration	Characteristics	Granularity
Azure19 [3]	14 days	14.7 K users, 24.3 K apps, 72.4 K functions, 12.5 B invocations	per-function per-minute invocation counts per-function per-day duration statistics
Azure21 [28]	14 days	119 apps, 424 functions, 2.0 M invocations	per-invocation end timestamp and duration
This work	217 days	252 users, 277.4 K functions, 2.1 M invocations	per-invocation six timestamps and duration

Table 8: Comparisons of real-world serverless datasets.

per-minute counts, it provides a timestamp and duration for each invocation. This dataset is the closest to our work.

Alibaba released 12-hour traces of its microservice architecture [23]. While not FaaS directly, microservices share similar characteristics (e.g., short, on-demand usage). Like Azure's 2019 dataset, microservices invocation counts are recorded in per-minute bins, but only the average execution time per bin is reported. This trace dataset highlights the call relations of microservices, a feature not available in Azure's datasets.

Although these datasets exist, the availability of comprehensive, in-depth serverless trace datasets remains limited. Consequently, the aim of our work is to enhance the available datasets by providing in-production serverless system traces spanning 217 days. Each invocation in the dataset includes six timestamps, accurately depicting the invocation's lifecycle in Globus Compute and spanning a set of 580 distributed endpoints. Additionally, we include code statistics, significantly extending the possibilities for future research using this dataset. An overview and comparison with the mentioned serverless datasets are listed in Table 8.

6.2. Serverless Measurements and Benchmarks

Another approach to gaining insights into serverless computing involves benchmarking or evaluating such systems and applications. Some studies utilize real-world traces to evaluate their solutions, while it is more common for serverless systems to be assessed under various synthetic workloads, encompassing variations in CPU, memory, disk, network usage, and degrees of parallelism. For instance, DeathStarBench [36], a widely used benchmarking set, comprises six Docker microservice applications (three of them are open-source), serving as a means to stress test serverless systems under load [37, 38]. ServerlessBench [39], which incorporates synthetic functions and applications, is used to evaluate function communication latency and startup latency on serverless platforms.

There are customized benchmarks tailored to scientific computing and data manipulation. For instance, the Faas-μbench-mark [40] includes two non-trivial functions, Fast Fourier Transformation and Matrix Multiplication, and can stress the CPU and memory at different levels. The FaaSdom benchmarking tool [41] contains testing suites that stress the CPU through integer factorization and matrix multiplication. Figiela et al. [42] used HPL Linpack to stress the CPU for floating-point performance evaluation. Kuhlenkamp et al. [43] focused on a workload to determine whether a number is a prime number, based on the Sieve of Eratosthenes algorithm, to stress the CPU and memory. Similarly, Lee et al. [44] designed a workload involving two-dimensional array multiplication with 0-100 concurrent invocations. Lloyd et al. [45] designed computational and memory-intensive workloads to reflect different stress levels, introducing 2 thousand to 10 million random math operations and 20 to 100 thousand operand array sizes to evaluate CPU and memory stress.

6.3. Serverless Characterization

The functions executed in our dataset share similarities with those run on other serverless systems concerning execution time, code complexity, and bursty workload patterns. Eismann et al. [22] found that over 75% of the serverless functions in their study run for less than a minute. They point out that execution time statistics are only discussed in Shahrad et al. [3] and two cloud SaaS company reports. Shahrad et al. rank the average of function execution time and reveal that the median of the average was less than a second. The authors also ranked the maximum function execution time and found that the median of the average was less than three seconds. Findings in a subsequent work by Zhang et al. [28] corroborate these statistics: the authors rank the average and 90 percentile of the application execution time and found that the median of the average is well less than a second, and the median of the 90-percentile is a round one second. They also provide some apple-to-apple comparison with our statistics: they rank 2.2 M invocations with recorded duration time and found 85% of these invocations take less than 1 second, and 96% of the invocations take less than 30s.

As pointed out by Eismann et al. [22], in 2019, New Relic reports [46] that the median duration of all monitored invocations in the second half of 2019 was approximately 719 ms, with functions on Node.js 10.x and Python 3.8 run shortest and below 450 ms. Similarly, in 2021, Datadog reported [19] the median of the monitored Lambda function execution time of about 130ms in 2019, but only 60ms in 2020. These findings align with our observations that serverless function execution times are typically short, as the median execution time of Globus Compute functions in our dataset is 30ms.

Serverless functions are also known for their simplicity. Eismann et al. estimated that 53% of the applications in their study processed a data volume of less than 1MB [22]. Our findings reveal that the median argument size was only 62 bytes, and approximately 75% of the functions have argument sizes less than

4MB. The authors of AWSomePy found that 55% of the AWSomePy repositories contained less than 1000 LoC [21]. Although these statistics are not directly comparable to ours, as their statistics are per applications while ours are per Python function, their findings support our impression that serverless functions are simple in terms of source code complexity.

Furthermore, serverless functions often exhibit bursty workloads and hot-spot invocation patterns. Eismann et al. [22] characterized 86% of serverless applications as having bursty workloads. Shahrad et al. [3] revealed that over 99.6% of the invocations occur in 18.6% of the applications. A parallel pattern in Alibaba's microservice architecture [23] showed a mere 5% of the microservices appear in nearly 90% of all call graphs and are responsible for handling 95% of all invocations. This aligns with our observations from hourly task submission figures and hourly task submissions by clusters of endpoints, which showed spiky request rates that were high one hour and zero the next, with occasional periods of sustained high request rates.

While Globus Compute is primarily used for Python-based scientific computing and machine-learning tasks, the serverless application landscape is considerably more diverse regarding programming languages and use cases. This diversity was underscored in a 2020 study by Eismann et al. [47, 22], revealing that a relatively small proportion, only 16%, of these applications were specifically designed to handle scientific workloads, and ML applications only make up to 5% of the applications. In contrast, our dataset shows that Globus Compute is dominated by scientific computing and scientific ML tasks. By providing realistic system traces and function classifications, our dataset paves the way for future research from a complementary perspective of the application repositories.

7. Conclusion

We presented a unique and comprehensive serverless dataset derived from the Globus Compute platform. Over 31 weeks, we collected data from 2.1 million function invocations across 580 distributed endpoints, involving 252 users and over 277 000 registered functions. Our analysis offers valuable insights that can contribute to a deeper understanding of serverless architectures and their performance characteristics. For instance, we observed intriguing patterns in the dataset, including the concentration of task submissions among a small group of active users, a common trend in serverless platforms. Moreover, our fine-grained capture of function execution lifecycles and the consideration of diverse endpoint configurations allowed us to explore arrival rates, run time distributions, and scheduling across the computing continuum, encompassing edge, cloud, and HPC environments. By analyzing function source code, we categorized functions into clusters representing different types of tasks, such as data manipulation, job scheduling, machine learning, and non-ML scientific computing. This classification sheds light on the diversity of serverless use and provides insights into function complexity and library dependencies.

We believe that the availability of this open-source and FAIR dataset, along with the analysis scripts, will foster further research in serverless computing and related fields. Moreover, we encourage researchers to explore the dataset and leverage its unique features to advance our understanding of serverless architectures and their applications across different computing environments. As the Globus Compute platform continues to grow, we plan to update the dataset to capture evolving usage patterns and further expand the research opportunities it offers.

References

[1] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, K. Chard, funcx: A federated function serving fabric for science, in: 29th International Symposium on High-Performance Parallel and Distributed Computing, ACM, 2020, pp. 65–76. doi:10.1145/3369583.3392683.

[2] Z. Li, R. Chard, Y. Babuji, B. Galewsky, T. J. Skluzacek, K. Nagaitsev, A. Woodard, B. Blaiszik, J. Bryan, D. S. Katz, et al., fX: Federated function as a service for science, IEEE Transactions on Parallel and Distributed Systems 33 (12) (2022) 4948–4963.

[3] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, R. Bianchini, Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, in: 2020 USENIX Annual Technical Conference, USENIX Association, 2020, pp. 205–218.

[4] L. Zhao, Y. Yang, Y. Li, X. Zhou, K. Li, Understanding, predicting and scheduling serverless workloads under partial interference, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, St. Louis Missouri, 2021, pp. 1–15. doi:10.1145/3458817.3476215.
URL https://dl.acm.org/doi/10.1145/3458817.3476215

[5] K. Kaffes, N. J. Yadwadkar, C. Kozyrakis, Hermod: principled and practical scheduling for serverless functions, in: Proceedings of the 13th Symposium on Cloud Computing, ACM, San Francisco California, 2022, pp. 289–305. doi:10.1145/3542929.3563468.
URL https://dl.acm.org/doi/10.1145/3542929.3563468

[6] A. Fuerst, P. Sharma, Locality-aware Load-Balancing For Serverless Clusters, in: Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing, ACM, Minneapolis MN USA, 2022, pp. 227–239. doi:10.1145/3502181.3531459.
URL https://dl.acm.org/doi/10.1145/3502181.3531459

[7] R. B. Roy, T. Patel, D. Tiwari, Icebreaker: Warming serverless functions better with heterogeneity, in: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022, pp. 753–767.

[8] Apache Software Foundation, Openwhisk (n.d.).
URL https://openwhisk.apache.org/community.html

[9] OpenFaas Ltd., Openfaas (n.d.).
URL https://www.openfaas.com

[10] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, et al., Parsl: Pervasive parallel programming in python, in: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, 2019, pp. 25–36.

[11] M. M. Head, Nvbl (national virtual biotechnology laboratory) molecular therapeutics, Tech. rep., USDOE Office of Science (2020).

[12] A. A. Saadi, D. Alfe, Y. Babuji, A. Bhati, B. Blaiszik, A. Brace, T. Brettin, K. Chard, R. Chard, A. Clyde, et al., Impeccable: Integrated modeling pipeline for covid cure by assessing better leads, in: Proceedings of the 50th International Conference on Parallel Processing, 2021, pp. 1–12.

[13] Y. Babuji, B. Blaiszik, T. Brettin, K. Chard, R. Chard, A. Clyde, I. Foster, Z. Hong, S. Jha, Z. Li, X. Liu, A. Ramanathan, Y. Ren, N. Saint, M. Schwarting, R. Stevens, H. van Dam, R. Wagner, Targeting sars-cov-2 with ai- and hpc-enabled lead generation: A first data release (2020). arXiv:2006.02431.

[14] R. Vescovi, R. Chard, N. D. Saint, B. Blaiszik, J. Pruyne, T. Bicer, A. Lavens, Z. Liu, M. E. Papka, S. Narayanan, N. Schwarz, K. Chard, I. T. Foster, Linking scientific instruments and computation: Patterns, technologies, and experiences, Patterns 3 (10) (2022) 100606. doi:https://doi.org/10.1016/j.patter.2022.100606.

[15] R. Chard, J. Pruyne, K. McKee, J. Bryan, B. Raumann, R. Ananthakrishnan, K. Chard, I. T. Foster, Globus automation services: Research process automation across the space–time continuum, Future Generation Computer Systems 142 (2023) 393–409. doi:https://doi.org/10.1016/j.future.2023.01.010.

[16] R. Chard, Z. Li, K. Chard, L. Ward, Y. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. J. Franklin, I. Foster, Dlhub: Model and data serving for science, in: IEEE International Parallel and Distributed Processing Symposium, 2019, pp. 283–292. doi:10.1109/IPDPS.2019.00038.

[17] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, D. Hassabis, Highly accurate protein structure prediction with AlphaFold, Nature 596 (7873) (2021) 583–589. doi:10.1038/s41586-021-03819-2.

[18] MaxMind, Geolite2 free geolocation data.
URL https://dev.maxmind.com/geoip/geolite2-free-geolocation-data

[19] DataDog, The state of serverless 2021 (May 2021).
URL https://www.datadoghq.com/state-of-serverless-2021/

[20] J. Beningo, Software quality, metrics, and processes, in: Embedded Software Design: A Practical Approach to Architecture, Processes, and Coding Techniques, Springer, 2022, pp. 151–178.

[21] G. Raffa, J. B. Alis, D. O'Keeffe, S. K. Dash, Awsomepy: A dataset and characterization of serverless applications, in: Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies, 2023, pp. 50–56.

[22] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, A. Iosup, The state of serverless applications: Collection, characterization, and community consensus, IEEE Transactions on Software Engineering 48 (10) (2021) 4152–4166.

[23] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, C. Xu, Characterizing microservice dependency and performance: Alibaba trace analysis, in: Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 412–426.

[24] L. McInnes, J. Healy, Accelerated hierarchical density based clustering, in: Data Mining Workshops, 2017 IEEE International Conference on, IEEE, 2017, pp. 33–42.

[25] OpenAI, Openai: Text and code embeddings (2022).
URL https://platform.openai.com/docs/guides/embeddings

[26] L. McInnes, J. Healy, UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction, ArXiv e-prints (Feb. 2018). arXiv:1802.03426.

[27] 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, D. Laney, Psi/j: A portable interface for submitting, monitoring, and managing jobs, in: Accepted to eScience, 2023.

[28] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, R. Bianchini, Faster and cheaper serverless computing on harvested resources, in: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 724–739. doi:10.1145/3477132.3483580.

[29] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, J. Mace, Serving {DNNs} like clockwork: Performance predictability from the bottom up, in: 14th USENIX Symposium on Operating Systems Design and Implementation, 2020, pp. 443–462.

[30] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, A. Akella, Atoll: A scalable low-latency serverless platform, in: Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 138–152.

[31] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, C. Das, Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms, in: Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 153–167.

[32] B. Wang, A. Ali-Eldin, P. Shenoy, Lass: Running latency sensitive serverless computations at the edge, in: Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 239–251. doi:10.1145/3431379.3460646.

[33] A. Fuerst, P. Sharma, Faascache: keeping serverless computing alive with greedy-dual caching, in: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 386–400.

[34] D. H. Sallo, G. Kecskemeti, Towards Generating Realistic Trace for Simulating Functions-as-a-Service, in: R. Chaves, D. B. Heras, A. Ilic, D. Unat, R. M. Badia, A. Bracciali, P. Diehl, A. Dubey, O. Sangyoon, S. L. Scott, L. Ricci (Eds.), Euro-Par 2021: Parallel Processing Workshops, Springer International Publishing, Cham, 2022, pp. 428–439.

[35] L. Zhao, Y. Yang, Y. Li, X. Zhou, K. Li, Understanding, predicting and scheduling serverless workloads under partial interference, in: Proceedings of the International conference for high performance computing, networking, storage and analysis, 2021, pp. 1–15.

[36] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 3–18. doi:10.1145/3297858.3304013.

[37] Z. Jia, E. Witchel, Boki: Stateful serverless computing with shared logs, in: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 691–707. doi:10.1145/3477132.3483541.

[38] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, H. Chen, Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 467–481. doi:10.1145/3373376.3378512.

[39] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, H. Chen, Characterizing serverless platforms with serverlessbench, in: Proceedings of the 11th ACM Symposium on Cloud Computing, 2020, pp. 30–44.

[40] T. Back, V. Andrikopoulos, Using a microbenchmark to compare function as a service solutions, in: Service-Oriented and Cloud Computing: 7th IFIP WG 2.14 European Conference, ESOCC 2018, Como, Italy, September 12-14, 2018, Proceedings 7, Springer, 2018, pp. 146–160.

[41] P. Maissen, P. Felber, P. Kropf, V. Schiavoni, Faasdom: A benchmark suite for serverless computing, in: Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems, DEBS '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 73–84. doi:10.1145/3401025.3401738.

[42] K. Figiela, A. Gajek, A. Zima, B. Obrok, M. Malawski, Performance evaluation of heterogeneous cloud functions, Concurrency and Computation: Practice and Experience 30 (23) (2018) e4792.

[43] J. Kuhlenkamp, S. Werner, M. C. Borges, D. Ernst, D. Wenzel, Benchmarking elasticity of faas platforms as a foundation for objective-driven design of serverless applications, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, pp. 1576–1585.

[44] H. Lee, K. Satyam, G. Fox, Evaluation of production serverless computing environments, in: 2018 IEEE 11th International Conference on Cloud Computing, IEEE, 2018, pp. 442–450.

[45] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, S. Pallickara, Serverless computing: An investigation of factors influencing microservice performance, in: 2018 IEEE international conference on cloud engineering, IEEE, 2018, pp. 159–169.

[46] N. Relic, For the love of serverless (2020).
URL https://newrelic.com/resources/ebooks/serverless-benchmark-report-aws-lambda-2020#toc-function-faster-on-aws-lambda

[47] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, A. Iosup, Serverless applications: Why, when, and how?, IEEE Software 38 (1) (2020) 32–39.