An Empirical Investigation of Container Building Strategies and Warm Times to Reduce Cold Starts in Scientific Computing Serverless Functions

André Bauer*, Maxime Gonthier†‡, Haochen Pan†, Ryan Chard‡, Daniel Grzenda†, Martin Straesser§, J. Gregory Pauloski†, Alok Kamatar†, Matt Baughman†, Nathaniel Hudson†‡, Ian Foster‡†, and Kyle Chard‡†
*Illinois Institute of Technology, Chicago, USA
†University of Chicago, Chicago, USA
‡Argonne National Laboratory, Lemont, USA
§University of Würzburg, Würzburg, Germany
andre.bauer@iit.edu

2024 IEEE 20th International Conference on e-Science (e-Science) | 979-8-3503-6561-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/e-Science62913.2024.10678668

***Abstract*—Serverless computing has revolutionized application development and deployment by abstracting infrastructure management, allowing developers to focus on writing code. To do so, serverless platforms dynamically create execution environments, often using containers. The cost to create and deploy these environments is known as "cold start" latency, and this cost can be particularly detrimental to scientific computing workloads characterized by sporadic and dynamic demands. We investigate methods to mitigate cold start issues in scientific computing applications by pre-installing Python packages in container images. Using data from Globus Compute and Binder, we empirically analyze cold start behavior and evaluate four strategies for building containers, including fully pre-built environments and dynamic, on-demand installations. Our results show that pre-installing all packages reduces initial cold start time but requires significant storage. Conversely, dynamic installation offers lower storage requirements but incurs repetitive delays. Additionally, we implemented a simulator and assessed the impact of different warm times, finding that moderate warm times significantly reduce cold starts without the excessive overhead of maintaining always-hot states.**

***Index Terms*—Scientific Computing, Serverless, Cold Start, Measurements, Simulation**

I. Introduction

The emergence of serverless computing has reshaped how applications are developed and deployed. By abstracting infrastructure management, serverless platforms promise to streamline development workflows and optimize resource utilization. At the core of this paradigm shift lies the use of Function-as-a-Service (FaaS), where users deploy small, event-driven functions without worrying about the underlying servers. FaaS platforms are elastic and can scale to zero. When a function is invoked, it is deployed by the provider with all necessary resources for execution. Environments are typically constructed by installing all dependencies in a container. Once the function completes its execution and no further requests follow, all allocated resources are released. Although this scaling has apparent benefits, such as reduced costs, it introduces *cold start* overhead to restart the resources for subsequent executions. For instance, AWS Lambda exhibits a latency of up to a few seconds for cold start [1]. A recent study even demonstrated that the cold start of a function can take up to 166 times the actual function runtime [2].

In addition to its increasing adoption in industry, scientists and engineers have begun embracing FaaS for more efficient application execution [3]. Consequently, reducing cold start times in scientific serverless applications is critical to enable low-latency, time-sensitive workloads, such as experiment steering [4], [5]. However, the highly diverse and specialized nature of scientific computational tasks, both within and across domains, makes it impractical to maintain custom environments for each task due to strict data quotas.

Traditional approaches to mitigating or minimizing cold starts involve either shortening the time required for container preparation or reducing the provisioning of function dependencies [6]. However, these approaches may not fully address the unique demands of scientific computing workloads, which often comprise functions invoked infrequently, exacerbating cold start delays [7].

In this context, we investigate the effectiveness of pre-installing Python packages in container images to alleviate the cold start problem, specifically in scientific computing applications. Leveraging insights from two prominent datasets—the Globus Compute dataset [7] and the Binder dataset [8]—we empirically analyze the cold start behavior of scientific computing functions and investigate four distinct container build strategies for mitigating cold start delays. These strategies range from installing all required packages during container build time to dynamically installing only missing packages based on historical usage data. We then investigate the effect of keeping containers "warm" for different periods of time to avoid future cold starts.

In summary, our contributions in this paper are threefold: (i) We explore the Globus Compute and Binder datasets, shedding light on the unique challenges posed by scientific computing workloads in serverless environments. (ii) We conduct an empirical study on the cold start characteristics of scientific computing functions, with a specific focus on the

impact of pre-installing Python packages in container images. (iii) We evaluate the effectiveness of different warm times in reducing cold starts for scientific computing functions, providing insights into optimizing serverless deployments in this domain.

Our findings indicate that the choice of container build strategy significantly impacts cold start times. The naïve strategy, which has the longest cold start times, remains a viable option when historical usage data is insufficient. The two strategies based on historical information reduce the initial cold start but introduce high storage requirements. Furthermore, our simulation and analysis of warm times reveal that moderate warm intervals effectively reduce cold starts without the excessive costs associated with maintaining always-hot states.

The remainder of this paper is structured as follows: Section II provides background information and introduces the datasets used. Section III analyses the datasets utilized and highlights the proposed strategies for reducing cold starts. Section IV presents the experimental setup and discusses the results of our empirical analysis. Section V investigates various times for keeping function warm and simulates their impact on cold start reduction. Section VI discusses related work. Finally, Section VII concludes the paper.

II. Background

In this section, we introduce containerization and cold start. We also describe the Globus Compute and Binder datasets used in this paper.

A. Containerization and Cold Start Times

In recent years, containers have become the dominant deployment technology for serverless functions and microservices. This is because containers are significantly lighter than virtual machines, as OS and kernel functions can be shared between containers while still providing the necessary isolation and portability of environments. The lightweight nature of containers also enables significantly faster start times, which in turn enables increased application scalability. However, as recent work [9] has shown, there are also significant variations in start times between different container images.

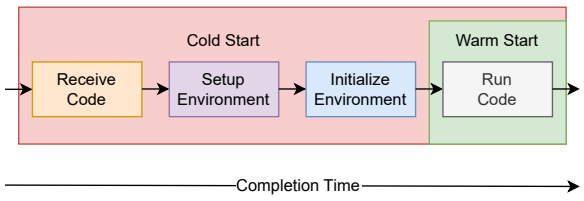

Fig. 1. Schematic overview of cold vs. warm starts.

In the context of serverless functions, a distinction is often made between cold and warm starts. In general, the start process of a function can be divided into four steps, as shown in Figure 1. In the first step, the code is received, and its metadata (e.g., environment settings and dependency versions) are evaluated. In the second step, an environment (e.g., a Python runtime with a specific version) is configured and prepared based on these settings. Next, the environment is instantiated and initialized. In the final step, the actual code is executed. In a warm start, all static setup steps have already been completed, and only the code needs to be executed. Typically, the image is saved after the very first cold start, eliminating the need to repeat the first two steps with each subsequent cold start.

To achieve maximum performance and low execution times, it would be optimal to avoid cold starts completely, i.e., always provide a sufficient number of "warm" containers. However, there is a tradeoff between response and resource utilization. Further, resources are limited, and in practice, especially with dynamic workloads, it is difficult to predict future function requirements and to provide sufficient resources at all times [10].

B. Globus Compute Dataset

There are few publicly available serverless datasets, with most coming from industry. The most notable are Azure's datasets [11], [12] and, although not directly related to FaaS, Alibaba's microservices dataset [13]. However, these datasets do not include information about the functions that are run or the packages that are used.

To this end, we utilize the Globus Compute dataset [7], which offers insights into Python functions used in scientific use cases. This dataset[1] spans a period between November 28th, 2022 and July 3rd, 2023 from the Globus Compute platform. It details information on 2,121,472 tasks submitted by 252 distinct users across 580 geographically distributed endpoints. The dataset also includes details about 277,386 registered Python functions with 1,847 distinct function bodies clustered into 29 clusters.

Globus Compute, formerly funcX [14], is a scientific FaaS platform that implements a unique hybrid cloud-edge architecture. It integrates a central cloud service with user-deployed and managed endpoints that can be set up on *any* computing resource, from edge devices to high-performance clusters. Users can register Python functions and submit them for execution on a chosen endpoint.

In contrast, hosted FaaS platforms like Amazon Lambda support a wide range of applications in various languages and are deeply integrated with their cloud ecosystems. Amazon Lambda uses Firecracker [15], while Globus Compute supports containerized Python functions with Docker for local and cloud use, and Singularity [16] and Shifter [17] for HPC environments.

C. Binder Dataset

We use the Binder dataset [8] to analyze real-world usage of Python packages in containerized applications. This dataset[2] contains information from the `repo2docker` tool [18] about deployed containers with Python environments from November, 2018 to June, 2021. During this period, 18,230,454 containers were deployed from 90,713 different Git repositories.

[1]Globus Compute dataset: https://doi.org/10.5281/zenodo.10044780
[2]Binder dataset: https://zenodo.org/records/4915858

The dataset includes information about the Python packages installed in 159,646 applications.

The Binder Project facilitates scientific reproducibility by developing environments that are shareable, interactive, and reproducible [19]. Through an online service, Binder enables the execution of interactive notebooks sourced from Git repositories. Users have the flexibility to define their required environment, typically encompassing datasets, application code, and documentation within a Git repository. Leveraging `repo2docker`, Binder dynamically constructs and deploys containers according to these specifications. These containers are then hosted on public cloud computing platforms, granting users browser-based interaction.

III. Methodology

In this section we analyze the Globus Compute and Binder datasets to gain insights into the required environments for real-world applications. Moreover, we discuss how we use these datasets in our subsequent experiments. Finally, we introduce the four container build strategies.

A. Analysis and Usage of the Datasets

As our interest is in reducing the time needed to set up the environment for the first cold start, we removed all functions/applications from both datasets that do not import any packages. In the Globus Compute dataset, we find 1441 unique function bodies located in 24 clusters (see Section III-D) and performed by 1,140,431 associated tasks. In the Binder dataset, we find 86,086 unique applications. Table I shows the package characteristics of the remaining functions/applications. For the Globus Compute dataset, functions import 3.03 packages on average. In total, this dataset has 130 unique packages and 246 combinations of these packages. Out of these 130 packages, 37 are from the Python standard library. For the Binder dataset, applications required 10.93 packages on average. In total, this dataset has 13,309 unique packages and 19,390 combinations of these packages. Moreover, 6,394 out of 86,086 images install more than 20 packages, with a maximum of 1043. When considering only the images with 20 or fewer packages, the average number of installed Python packages is 6.39. Please note that the Binder dataset includes only packages that have to be installed; that is, it does not list packages from the Python standard library.

TABLE I
Python package characteristic comparison between the Globus Compute and Binder dataset.

Characteristic	Globus	Binder
Average imported/installed packages	3.03	10.93
Unique packages	130	13,309
Unique standard library packages	37	—
Unique package combinations	246	19,390

B. Package Installation Time

We explore the installation time of each package in these two datasets by timing the installation of each package in a new Docker container on our reference system (see Section IV-A). Each package was installed ten times, and Table II shows the average installation time for the most common packages and those with the longest installation time. `torchvision` and `torch` both take more than 180 seconds to be installed. `torch` is also the seventh most imported package of the installed packages. On the right side, the most imported packages are shown. `numpy` is the most imported package in the dataset, and it takes 8.47 seconds to install on average. So, if a container with `numpy` already installed was provided, the cold start time could be reduced by 8.47 seconds. Except for `torch` and `tensorflow`, the top 10 imported installed packages require less than 22 seconds to be installed. On average, a package from this dataset requires 9.16 seconds to be installed.

TABLE II
Top 10 packages with the longest install time (left) vs. top 10 installed packages frequently imported (right) based on the Globus Compute dataset.

Package	Install time [s]	Package	Install time [s]
torchvision	182.88	numpy	8.47
torch	180.47	pathlib	4.66
tensorflow	83.49	pandas	18.46
keras	77.23	torch	180.47
mlflow	64.65	tensorflow	83.49
mplsoccer	34.59	proxystore	9.61
statsmodels	29.18	sklearn	21.74
quickstats	26.98	datetime	5.80
imblearn	21.83	pyhf	19.12
sklearn	21.74	matplotlib	16.15

Table III reports the ten packages that took the longest time to be installed (left) and the top 10 installed packages (right) in the Binder dataset. Here, `lux-api` took the longest time to be installed. The remaining nine packages require at least 245 seconds to be installed. On average, a package from this dataset requires 30.11 seconds to be installed. `bokeh` is the most imported package. Like the Globus Compute dataset, the Binder dataset also includes `matplotlib`, `numpy`, `pandas`, and `sklearn` packages among the top 10 packages used. These four packages are also among the 20 most common PyPI packages [20].

For the measurements presented in the following sections, we only use the Globus Compute dataset, while the Binder dataset is used as a reference point. The reasons for this decision are (i) given the vast amount of Python packages and images of the Binder dataset, the experiments as described in Section IV-A would be infeasible; (ii) and the focus of this paper is on scientific serverless computing as reassembled by the Globus Compute dataset.

C. Container Building Strategies

The typical process (i.e., cold start) for invoking a function for the very first time in a serverless setting is illustrated in

TABLE III
TOP 10 PACKAGES WITH THE LONGEST INSTALL TIME (LEFT) VS. TOP 10 INSTALLED PACKAGES FREQUENTLY IMPORTED (RIGHT) BASED ON THE BINDER DATASET.

Package	Install time [s]	Package	Install time [s]
lux-api	248.64	bokeh	70.91
netallocation	247.76	bs4	13.16
ioos-tools	247.23	fuzzywuzzy	9.24
scikit-surprise	246.38	geopandas	69.55
andes	246.13	matplotlib	16.15
py-heat-magic	245.97	numpy	8.47
vasppy	245.83	pandas	18.46
fluxengine	245.70	pycountry	12.68
icepyx	245.62	seaborn	72.98
bionetgen	245.22	sklearn	21.74

Figure 1. To diminish the cold start duration, we focus on reducing the setup time of the environment (purple box). To do so, we investigate four different strategies in this paper:

Naïve: As a baseline, this approach entails fetching a standard Python container (amd64/python:3.11) and installing all dependencies during container building.

All Top 10: For this strategy, a pre-installed Docker image is fetched. Based on a standard Python container (amd64/python:3.11), this image is pre-installed with the top 10 most frequently used packages across all historical functions recorded in the environment. Each of these installations was represented as a single line within the Dockerfile, resulting in the creation of a separate layer for each dependency. This way, Docker can apply layer optimization during the start.

Cluster Top 10: This strategy involves assigning functions to one of 24 clusters. Each cluster has a unique Docker image built on a standard Python container (amd64/python:3.11) with the ten most commonly used packages from all historical functions within the cluster pre-installed. Similar to *All Top 10*, each installation was represented as a single line within the Dockerfile.

On-the-fly: As about 40% of the functions in the Globus Compute dataset were only executed once, this strategy installs all required dependencies from scratch every time the container is started. In essence, while *Naïve* installs the packages during the building of the docker container, this strategy installs the packages after the container is built and started. The image used is the same as for *Naïve* and also follows the conventional approach.

The concept behind *All Top 10* and *Cluster Top 10* is to have specific necessary dependencies pre-installed based on the most used dependencies across all historical functions. Consequently, only the missing dependencies have to be installed. For *All Top 10*, we choose ten packages (coincidently, also 10% of all packages), as we can cover almost 60% of all package combinations of the Globus Compute dataset as depicted in Figure 2. For comparison, we also set *Cluster Top 10* to ten packages. Indeed, any other number could be chosen, but we wanted to keep the number of installed packages relatively small.

Please note that we chose to use Docker for our experiments. However, the approach is not limited to Docker; alternative container technologies such as Singularity or Podman can be used instead.

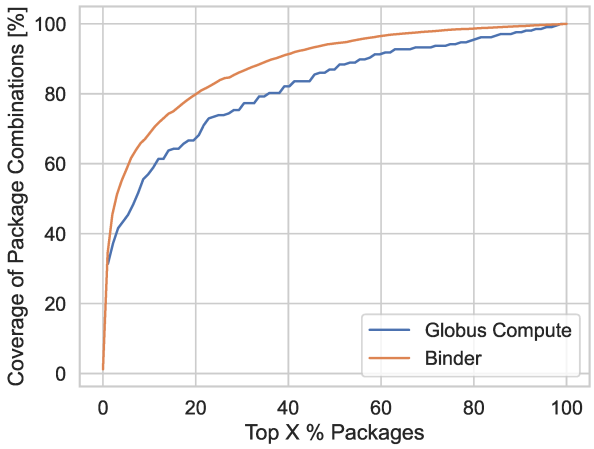

Fig. 2. Coverage of package combinations using top x% of packages.

D. Clustering of Functions

As highlighted in Section III-A, the dataset we use has 230 unique package combinations. In general, the number of package combinations can be numerous. Consequently, having a pre-installed environment at hand for every combination is infeasible. We cluster the functions to reduce the number of pre-installed environments to a manageable set. Although clustering based on only packages might seem intuitive, we chose to cluster the functions based on their functionality. This approach takes into account both dependencies and code, allowing us to group functions with similar purposes, such as data manipulation, together. This strategy aims to facilitate potential performance enhancements in future research.

For the clustering approach, we tried different methods. HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) clustering [21] exhibited the best results based on the silhouette coefficient. To have a typical representation of each function, we utilized an embedding model from OpenAI [22], transforming each function into a high-dimensional array with values $v_i \in [-1; 1]$. Finally, we grouped these representations of the functions with HDBSCAN into 24 clusters.

IV. EMPIRICAL STUDY OF CONTAINER BUILDING STRATEGIES

In this section, we perform an empirical study on the four container building strategies. We first introduce the experimental setup and discuss different aspects of the container building strategies: (i) time for setting up the environment, (ii) cold start time, and (iii) overhead in terms of storage requirements and start time. Also, we discuss threats to validity and summarize our results.

A. Experimental Setup

To measure cold start times, we performed ten randomized repetitions with randomized multiple interleaved trials [23]. That is, we randomized the time to start and the order of the invoked functions for each repetition. Additionally, at the beginning of each repetition, we deleted all images to ensure an empty system, necessitating the build of all images. In addition, after we invoked a function with a cold start, we ran each function five times to have information on warm starts.

Although we had access to the function source code, we did not have access to the data with which the functions were called. To this end, for each function, we replaced the function body with a dummy function; each function imports the packages and then calls `sleep(1)`.

We captured the execution time and other times of interest of each function invocation in a fine-grained manner as follows:

Build time: This time includes fetching the container image, performing all commands specified in the Dockerfile, and creating a new container image. In other words, it is the time to complete the step *Setup Environment* (see Figure 1).

Start time: The time Docker requires to provide a runnable environment from a container image [9]. We consider this time as overhead.

Completion time: The duration from invoking a function until it finishes. In the case of a cold start, it includes all steps within the red box illustrated in Figure 1. Otherwise, if a container is warm, it is the time for executing the function (green box).

All experiments were conducted in a self-hosted cloud managed by Proxmox[3] (version 7.4). The cloud consists of 17 servers with identical hardware. The servers are HP ProLiant DL360 Gen9 with Intel(R) Xeon(R) CPU E5-2640, 2x16 GiB HP 752369-081 DDR4 RAM, and a 500 GB HDD disk of type HP MB0500GCEHE. Dynamic frequency scaling is enabled as default and also further CPU-oriented features are not changed. The used Docker version is 25.0.3.

TABLE IV
COMPARISON OF THE NUMBER OF PACKAGES TO INSTALL.

Packages to install	Mean	Median	SD	Range
Naïve	1.99	1	1.40	[1, 7]
All Top 10	0.79	1	0.96	[0, 5]
Cluster Top 10	0.07	0	0.30	[0, 2]
On-the-fly	1.99	1	1.40	[1, 7]

B. Environment Setup Analysis

To compare the four different strategies, we calculate the number of packages to be installed and measure the container build time. In this analysis, we consider only functions from the Globus Compute dataset that imported at least one package that has to be installed. Results are presented in Table IV.

As the *Naïve* and *On-the-fly* images have only the Python Standard Library packages installed, every other package must be installed. Consequently, *Naïve* and *On-the-fly* give us an insight into how many packages have to be installed across the considered functions. On average, 1.99 packages must be installed with a median value of 1. In total, the number of packages to be installed ranges from 1 to 7 packages. In contrast, *All Top 10* and *Cluster Top 10* have to install, on average, only 0.79 and 0.07 packages, respectively. Moreover, in 50% of the considered functions, *All Top 10* and *Cluster Top 10* have to install 1 and 0 packages. Zero indicated that all needed packages were already installed. The maximum amount of packages to be installed for *All Top 10* and *Cluster Top 10* is 5 and 2, respectively. Although we were able to reduce the number of packages to be installed with *All Top 10*, 94% of the considered functions have all required packages installed in the more fine-grained approach of *Cluster Top 10*. However, on average, 8.00 and 6.93 packages for *All Top 10* and *Cluster Top 10*, respectively, are present in the image but not utilized by the function.

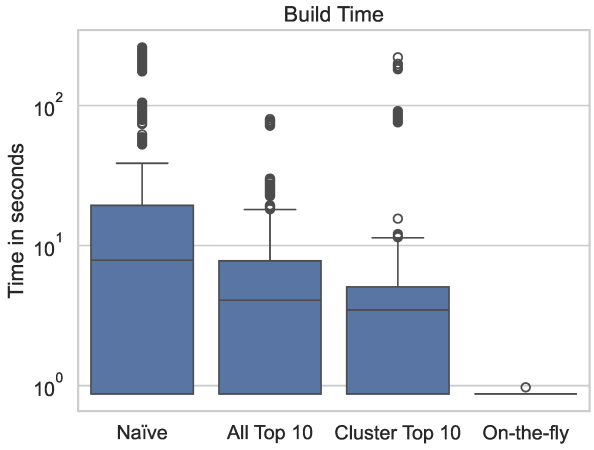

Fig. 3. Comparison of the build time of the different approaches.

TABLE V
SPEED-UP OF BUILD TIME COMPARED TO *Naïve*.

Speed-up	Geometric mean	Median	Geo. SD	Range
All Top 10	2.12	1.24	2.82	[0.86, 56.66]
Cluster Top 10	2.53	1.67	3.04	[0.9, 71.63]
On-the-fly	7.28	6.95	6.41	[0.9, 296.95]

To quantify the reduction in packages to be installed, we investigate the building time of the different strategies (see Figure 3). *Naïve* takes an average of 32.58 seconds to build the entire environment. The median time is 6.07 seconds, with values ranging from 0.87 seconds to 259.47 seconds. Installing on average 1.2 packages less than *Naïve*, *All Top 10* requires, on average, just 5.21 seconds to build the environment. Similarly, *Cluster Top 10* has a mean build time of 4.61 seconds. As *On-the-fly* installs all dependencies during the runtime of the container, it takes only an average of 0.89 seconds. In fact, installing Python packages during the runtime of the container (*On-the-fly*) is, on average, 24% faster than during the building

[3]Proxmox: https://www.proxmox.com/en/

of the container (*Naïve*). We also evaluated the speed-up of the strategies compared to *Naïve* as shown in Table V. Please note that we reported the geometric mean and standard deviation for the speed-up. On average, using *All Top 10* results in a speed-up of 2.12, with a maximum of 56.66 compared to *Naïve*. *Cluster Top 10* builds the environment on average 2.53 times faster than *Naïve*, with a maximum speed-up of 71.63. *On-the-fly* builds the environment on average 7.28 times faster, with a maximum speed-up of 296.95.

C. Cold Start Analysis

To investigate the cold start of the four strategies, we compare the completion time for the very first cold start of each function with each subsequent cold start, with the results listed in Table VI.

For the very first cold start, the *Naïve* strategy takes, on average, 34.89 seconds and up to 266.13 seconds to complete the function invocation. In contrast, with *All Top 10* and *Cluster Top 10*, it takes an average of 7.53 and 6.88 seconds, respectively. As *All Top 10* has already `torch` and `torchvision` installed for all functions but a few functions have to install these packages while using *Cluster Top 10*, *All Top 10* has a maximum of 86.77 and *Cluster Top 10* 227.00 seconds, respectively. *On-the-fly* has an average time of 27.11 seconds. Although *Naïve* and *On-the-fly* have to install all dependencies, *On-the-fly* is faster. Consequently, installing dependencies in a running container is faster than installing them during the docker build.

While investigating subsequent cold starts, *Naïve*, *All Top 10*, and *Cluster Top 10* require, on average, less than 1.5 seconds for a complete function invocation. This is because the docker image for each function is only built once and then stored and reused. In fact, the differences between these three strategies are in the order of milliseconds. However, *On-the-fly* installs for each cold start every dependency. Therefore, on average, it takes 23.80 seconds for a function invocation, while the median and maximum times are almost identical.

TABLE VI
COMPARISON OF COLD START COMPLETION TIMES.

Characteristic	Mean	Median	SD	Range
1st cold start completion time [s]				
Naïve	34.89	7.53	61.93	[2.18, 266.13]
All Top 10	7.53	5.41	8.04	[2.2, 86.77]
Cluster Top 10	6.88	4.76	12.81	[2.18, 227.0]
On-the-fly	27.11	6.72	46.92	[2.18, 219.0]
Subsequent cold start completion time [s]				
Naïve	1.48	1.20	0.68	[1.01, 4.57]
All Top 10	1.44	1.08	0.69	[1.02, 4.37]
Cluster Top 10	1.44	1.10	0.69	[1.01, 4.42]
On-the-fly	23.8	6.87	43.88	[1.02, 219.3]

D. Storage Requirement

Since *All Top 10* provides a single image to build the environment for each function and *Cluster Top 10* 24 images, we investigate the storage requirements to store built containers. The results of each strategy are provided in Table VII.

TABLE VII
COMPARISON OF STORAGE REQUIREMENTS.

Image size [GB]	Mean	Median	SD	Range	
Naïve	1.73	1.02	1.61	[1.01, 7.33]	2492.93
All Top 10	8.47	8.45	0.09	[8.45, 9.56]	12205.27
Cluster Top 10	3.33	1.58	2.44	[1.01, 7.33]	4798.53
On-the-fly	1.01	1.01	0	[1.01, 1.01]	1445.41

On average, a function image with *Naïve* has an average image size of 1.74 GB. When no packages were installed, the image size was just 1.01 GB. In contrast, the biggest image is 7.33 GB. We see that *All Top 10* requires significantly more space, with every image larger than the biggest image with *Naïve*. With *All Top 10* images range from 8.45 to 9.45 GB with an average of 8.47 GB. These high values result from the "big" Python packages that were installed beforehand. In contrast, *Cluster Top 10* has a similar storage range to *Naïve* but has an average of 3.33 GB. As *On-the-fly* installs all packages once the container has started, it has a constant image size of 1.01 GB. Considering the number of unique functions (1441) within the Globus Compute dataset, the storage requirements to host all functions are 2,493 GB, 12,205 GB, 4,799 GB, and 1,445 GB for *Naïve*, *All Top 10*, *Cluster Top 10*, and *On-the-fly*, respectively. This number would be significantly larger for the Binder case with its 86,086 unique applications.

In addition to the traditional approach of saving a separate image for each function, there are alternative strategies for saving images. One such strategy, proposed by Kumari and Sahoo [24], involves deploying multiple functions together within a single image. If we consider this approach and deploy functions sharing the same packages together, *Naïve* and *On-the-fly* would both require 427.31 GB, *All Top 10* would need 728.42 GB, and *Cluster Top 10* would require 106.56 GB.

TABLE VIII
COMPARISON OF THE START TIMES. START TIME REFERS TO THE TIME DOCKER REQUIRES TO PROVIDE A RUNNABLE ENVIRONMENT [9].

Start time [ms]	Mean	Median	SD	Range
Naïve	247.20	247.10	23.58	[172.61, 374.08]
All Top 10	246.74	246.27	22.54	[174.78, 369.92]
Cluster Top 10	248.15	247.49	23.87	[163.28, 393.68]
On-the-fly	247.82	245.58	28.84	[167.49, 383.57]

E. Start Time Overhead

Besides the storage requirements, we asses the overhead by the time taken to start the whole Docker image for the function invocation. With start time, we refer to the time Docker requires to provide a runnable environment [9]. The results are visualized in Figure 4 and listed in Table VIII.

On average, *Naïve* exhibits an average start time of 247.20 milliseconds with a median value of 247.01 milliseconds. *All Top 10* and *Cluster Top 10* experienced a slower mean start

time of 246.74 milliseconds and 248.15 milliseconds. *On-the-fly* has an average start time of 247.82 ms. As all these values are remarkably similar, we performed t-tests to investigate whether there is a significant difference in the means of these strategies. As we have $50 \cdot 1441$ measurements, the sampled distribution approximated a normal distribution according to the Central Limit Theorem. The resulting p-values for the tests *Naïve* vs. *All Top 10*, *Naïve* vs. *Cluster Top 10*, and *Naïve* vs. *On-the-fly* are 0.44, 0.29, and 0.56, respectively. That is, there is insufficient evidence to reject the null hypothesis. In other words, there is no significant difference between the means of the strategies. Also, the other combinations did not exhibit any significance. Consequently, we can conclude that the start time is not influenced by the choice of the strategy.

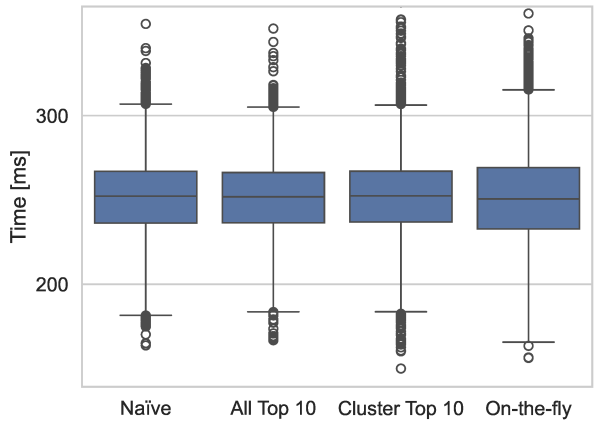

Fig. 4. Comparison of the start time of the different approaches.

F. Threats to Validity

For the Globus Compute dataset, we do not have information on which package version was required for the function execution. To this end, we used the latest version of each package in our experiments. In the case of the pre-installed container images, if a function would require a different version, it could be up- or downgraded during the building of the container. That is, changing the version would increase the building time but less than installing the whole package from scratch. Although we run every experiment several times, we run the experiments in one cloud environment. Consequently, our results could change if run in a different environment. But we would expect the order of results to stay the same.

G. Summary of the Results

The *Naïve* strategy has the longest build time and thus exhibits the slowest first cold start time. However, this strategy can be used when insufficient historical information is available to use strategies like *All Top 10* or *Cluster Top 10*. Also, this strategy requires less storage compared to *Cluster Top 10* and *All Top 10*. While comparing *All Top 10* with *Cluster Top 10*, the latter strategy has a shorter building time and requires less storage. After the first cold start, these three strategies exhibit similar cold starts. *On-the-fly* is for the very first cold start faster than *Naïve* and has the lowest storage requirement. However, for every cold start, it has to install every dependency again, and thus, it does not benefit from saving an installed image as the other strategies. That is, *On-the-fly* is only recommendable when functions are only run once, and there is not enough historical information available for *All Top 10* or *Cluster Top 10*.

V. Simulation

To experiment with the effect of different warm times on a FaaS service, it would be necessary to keep container images alive on compute nodes for longer than usual. This would be disruptive to users of that service. For this reason, we chose to run simulations based on the timestamps of the Globus Compute Dataset and the times gathered during our experimental evaluation. This section describes first our simulator and the warm selection. Then, we discuss the simulation based on (i) completion time, (ii) saved time, and (iii) cold starts. Finally, we summarize the simulation results.

A. Simulation & Warm Time Selection

To assess the impact of our approach at scale, we built an event-based Python simulator based on our measurements (see Section IV) and timestamps from the Globus Compute dataset. For each task in the simulation, we use the associated function (i.e., the Python package dependencies), the recorded submission time of the task, the measured build time for building the associated container for the function, the measured start time, and the recorded execution time of the task. We used the mean values for the measured values over all experimental runs.

At the beginning of the simulation, the system is empty. This means that the first time a function is called, the container must be built, which is indicated by the build time. To investigate the reduction of cold start times, we explore five different warm strategies (each function has its own image as specified in Section III-C):

$Cold_{\infty}$: After a function is terminated, the container is shut down. That is, each function execution is a cold start. This behavior reflects the worst-case scenario.

$Warm_5$: The container is kept alive for 5 minutes after the completion of its function. If the same function is called again in this time window, it uses the active container without the need to start a new instance of the container. We set the window to 5 minutes according to the previous analysis of the Globus Compute dataset [7].

$Warm_{10}$: The container is kept alive for 10 minutes. This is a typical time window used by AWS and Azure [25], the OpenWhisk serverless platform, used by IBM Cloud [26], and other serverless providers [27].

$Warm_{15}$: The container is kept alive for 15 minutes. This is another standard duration used by AWS Lambda [26], [28].

$Warm_{\infty}$: The container is kept alive forever. That is, each subsequent invocation of the same function uses the same container. This is the optimal scenario and allows us to estimate the potential gain that is missed by the limited time windows currently used.

For each warm time and building strategy, we simulated 271 days with a total of 1,140,431 task invocations and reported the build time, start time, and completion time. As we discussed in Section IV the build and start time, we focus in this section on the completion time.

B. Simulation Results

To investigate the different warm times, we compare the completion time, the total saved time, and the number of cold starts. The results are listed in Table IX.

For each building strategy, the most significant decrease in both average and median completion times occurs when transitioning from always cold ($Cold_\infty$) to a 5-minute warm period ($Warm_5$). This is primarily because 93% of the functions in the Globus Compute dataset are invoked with a frequency equal to or less than five minutes [7]. In our refined dataset (refer to Section III-A), we observe a cold start occurrence of 4.04% with $Warm_5$. This percentage further improves to 2.24% and 1.58% with warm periods of 10 minutes ($Warm_{10}$) and 15 minutes ($Warm_{15}$), respectively. As $Cold_\infty$ by definition results in 100% cold starts. $Warm_\infty$ results in 0.1% cold starts (i.e., just the 1st start of each function). Overall, the completion time decreases correspondingly with increasing warm times. However, the most significant reduction in cold starts happens between $Cold_\infty$ $Warm_5$. The improvements beyond this point are only marginal and may not justify the associated resource overhead. Nonetheless, these results demonstrate the unique challenges that scientific computing faces, as Shahrad et al. studied workload in industry, for example, and in their dataset, there is a significant difference between 5, 10, and up to 60 minutes of warm time.

Similar to the results in Section IV, the ranking of the building strategies remains consistent across all warm times. On average, *Cluster Top 10* has the shortest completion time followed by *All Top 10*, *Naïve*, and *On-the-fly*. For 50% of the function invocations, those deployed with *All Top 10* demonstrate a slightly quicker completion time (by a tenth of a millisecond) compared to those deployed with *Cluster Top 10*.

In our simulation, to explore warm times and building strategies on a large scale, we quantified the time saved for each scenario. We set *Naïve* with $Cold_\infty$ as the baseline and compared its total time against the other scenarios, calculating the percentage of time saved. For instance, *On-the-fly* with $Cold_\infty$ took 9.56% longer compared to baseline. However, introducing a warm time with *On-the-fly* resulted in a minimal time saving of 0.52% of the total time. Similarly, incorporating a warm time with *Naïve* led to a minimum saving of 3.18%. Both *All Top 10* and *Cluster Top 10* demonstrated savings of at least 0.41% with $Cold_\infty$ and at least 3.50% with a warm time. Overall, across all scenarios, longer warm times translated to greater time savings.

C. Effect on Individual Users

In the previous section, we were interested in how the different building strategies and warm times affect the whole Globus Compute landscape. Now, we investigate how this affects individual users. To this end, we choose three representative users from the Globus Compute trace:

Power: This user invoked frequently six different functions. Each function has at least one package to be installed and was run over 10,000 times. On average, 9.00 Python packages per function had to be installed.

Standard: This user ran a total of 18 different functions. The top invoked function was run more than 1,000 times, while the other functions were run between 10 to 100 times. Also, each function has at least one Python package to be installed. On average, 2.86 packages per function had to be installed.

Unique: This user just ran 48 different functions exactly once. Each function has at least one Python package to be installed, and on average, 3.40 packages had to be installed.

TABLE IX
COMPARISON OF THE DIFFERENT WARM TIMES.

Characteristic	$Cold_\infty$	$Warm_5$	$Warm_{10}$	$Warm_{15}$	$Warm_\infty$
Average completion time [ms]					
Naïve	9158.33	8867.18	8861.79	8859.78	8855.05
All Top 10	9121.02	8837.74	8832.47	8830.51	8825.88
Cluster Top 10	9116.20	8837.07	8831.88	8829.94	8825.38
On-the-fly	10034.04	9110.86	9107.57	9105.54	9097.14
Median completion time [ms]					
Naïve	336.68	33.01	32.71	32.61	32.41
All Top 10	331.67	32.95	32.65	32.55	32.35
Cluster Top 10	321.93	33.05	32.75	32.65	32.45
On-the-fly	337.05	335.35	335.35	335.35	335.35
Total saved time [%]					
Naïve	0.00	3.18	3.24	3.26	3.31
All Top 10	0.41	3.50	3.56	3.58	3.63
Cluster Top 10	0.46	3.51	3.56	3.59	3.64
On-the-fly	-9.56	0.52	0.55	0.58	0.67
Cold starts [%]	100	4.04	2.24	1.58	0.01

TABLE X
COMPARISON OF THE DIFFERENT BUILDING STRATEGIES WITH $Warm_{10}$.

Characteristic	*Power*	*Standard*	*Unique*
Average completion time [ms]			
Naïve	77.71	5961.38	24897.24
All Top 10	76.45	5833.33	20895.34
Cluster Top 10	75.65	5804.51	5333.18
On-the-fly	141.15	6019.45	17566.58
Median completion time [ms]			
Naïve	50.01	5069.91	5077.72
All Top 10	50.02	5069.15	4094.95
Cluster Top 10	50.00	5069.17	3397.01
On-the-fly	49.77	5070.60	4335.48

Table X lists the average and median completion time for all three users. For this investigation, we set a warm time of 10 minutes, as this is the standard for most industry providers. The fastest average completion time for *Power* is maintained with *Cluster Top 10*. However, *Naïve* and *All Top 10* are just 1 ms slower. For 50% of its function invocations, *On-the-fly* exhibits the shortest completion time. For *Standard*, the median completion time is almost identical for all building strategies, while *Cluster Top 10* provides the fastest average completion time. The most interesting user is *Unique*, as each

function is called only once. That is, containers cannot be reused, which is one use case for *On-the-fly*. Consequently, this strategy exhibits a faster completion time than *Naïve* and *All Top 10*. However, *Cluster Top 10* is by far the fastest due to its fine-grained clusters, and therefore no packages have to be installed.

D. Summary of the Simulation Results

As the warm-up time increases, the number of cold starts decreases as expected. However, the change between $Cold_{\infty}$ to $Warm_5$ is much higher than from $Warm_5$ to $Warm_{10}$ and even less reaching $Warm_{\infty}$. $Warm_5$ to $Warm_{15}$ reflect state-of-the-art warm times, and compared to the best scenario (i.e., $Warm_{\infty}$), there is no huge gain justifying the unlimited warm time. Overall, the rankings and findings from the experiments (see Section IV) also hold true for the large-scale simulation. Instead of looking at a whole system but only one selected user, the choice of the building strategy has more impact on their completion time.

VI. Related Work

Vahidinia et al. [6] surveyed approaches for reducing cold start duration and classifies them into two categories: Shortening the time required for container preparation or reducing the provisioning of function dependencies. Therefore, we focus on related work from both categories.

Sandboxing and Isolation Techniques: SAND [29] uses sandboxing to efficiently manage function resources. It leverages operating system mechanisms to share preloaded code, like loaded libraries, which are then forked to handle incoming requests. Boucher et al. [30] developed a multi-tenant worker process that directly executes functions by dynamically loading function code and executing it as a thread. Kumari and Sahoo [24] analyze and group similar functions together, deploying them in one container while isolating them through different processes.

Container and Unikernel Techniques: SEUSS [31] utilizes unikernels at the system level to snapshot serverless applications, cloning them as needed. Silva et al. [32] employ a cloning technique based on Checkpoint/Restore In Userspace (CRIU) to restore the state of a function in a cloned process later for scalability needs. Xu et al. [1] introduced an adaptive container pool scaling strategy, establishing a repository of pre-launched containers to respond to requests immediately. The quantity of containers within each category is dynamically adjusted according to historical invocation patterns. Oakes et al. [33] proposed a provisioning strategy using fully packaged containers that can be immediately deployed without requiring library imports or initialization. Despite its benefits, this approach raises security concerns as the cache can be shared among multiple processes. HotC [34] manages a pool of live container runtimes, analyzing user input or configuration files to match the environment and load code accordingly.

Pre-warming and Caching Techniques: PipBench [35] provides a package-aware compute platform where a common package repository can be shared among different microservice handlers belonging to various customers. Pagurus [36] incorporates an intra-function manager to transform an idle warm container into one available for other functions' use without introducing additional security risks. Shahrad et al. [11] apply a histogram policy, estimating the following invocations for each individual function. In addition, they use pre-warming to reduce resource waste.

Optimizing Function and Container Usage: Lee et al. [37] suggest merging two functions into one to avoid the cold start of the second function. However, fusing parallel functions may increase workflow response time as parallel functions are executed sequentially. Zhou et al. [2] introduce Multi-Level Container Reuse, which groups packages into OS, language, and runtime categories, and employs Deep Reinforcement Learning to optimize the reuse of warm containers.

In contrast to existing approaches, our work focuses specifically on alleviating the cold start problem in scientific computing applications by pre-installing Python packages in container images. While many related works address cold start issues through various strategies such as sandboxing [29], unikernels [31], and container pre-warming [33], our research takes a distinct empirical approach: We evaluate different container build strategies and explore the impact of different warm times on cold start delays, aiming to balance between maintaining container warmth and avoiding excessive costs.

VII. Conclusion

In this article, we have addressed the cold start problem in serverless computing for scientific applications by examining various strategies for pre-installing Python packages in container images. Our findings indicate that the choice of build strategy significantly impacts cold start times and resource utilization. *Naïve*, while exhibiting the longest cold start times, remains a viable option when historical usage data is insufficient. *Cluster Top 10* offers a balanced approach, reducing both build time and storage requirements compared to *All Top 10*. *On-the-fly*, incurs repetitive cold start delays, making it suitable only for infrequent function executions. Furthermore, our simulation and analysis of warm times reveal that moderate warm intervals effectively reduce cold starts without the excessive costs associated with maintaining always-hot states. These insights underscore the importance of tailoring serverless deployment strategies to the specific needs of scientific workloads, enabling more efficient and responsive scientific computing in serverless environments.

According to our experimental framework, future studies could expand by conducting experiments using datasets from cloud serverless providers to enhance the reliability of the outcomes. Also, running the experiments in different cloud providers could reveal potential variations in performance and resource utilization, providing a more comprehensive understanding of the effectiveness of different build strategies across diverse cloud infrastructures. Another avenue worth exploring is dynamic version control. Essentially, developing strategies for dynamic version management within container images could optimize build times and resource utilization.

ACKNOWLEDGEMENT

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 510552229. This work was partially supported by NSF 2004894 and Argonne National Laboratory under U.S. Department of Energy under Contract DE-AC02-06CH11357.

REFERENCES

[1] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems*. IEEE, 2019, pp. 9–16.

[2] A. C. Zhou, R. Huang, Z. Ke, Y. Li, Y. Wang, and R. Mao, "Tackling cold start in serverless computing with multi-level container reuse," in *2024 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2024.

[3] R. Chard, T. J. Skluzacek, Z. Li, Y. Babuji, A. Woodard, B. Blaiszik, S. Tuecke, I. Foster, and K. Chard, "Serverless supercomputing: High performance function as a service for science," *arXiv preprint arXiv:1908.04907*, 2019.

[4] A. V. Babu, T. Zhou, S. Kandel, T. Bicer, Z. Liu, W. Judge, D. J. Ching, Y. Jiang, S. Veseli, S. Henke *et al.*, "Deep learning at the edge enables real-time streaming ptychographic imaging," *Nature Communications*, vol. 14, no. 1, p. 7059, 2023.

[5] 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, and I. T. Foster, "Linking scientific instruments and computation: Patterns, technologies, and experiences," *Patterns*, vol. 3, no. 10, p. 100606, 2022.

[6] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems*, 2020, pp. 1–7.

[7] 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, 4 2024.

[8] T. Shaffer, K. Chard, and D. Thain, "An empirical study of package dependencies and lifetimes in binder Python containers," in *2021 IEEE 17th International Conference on eScience*. IEEE, 2021, pp. 215–224.

[9] M. Straesser, A. Bauer, R. Leppich, N. Herbst, K. Chard, I. Foster, and S. Kounev, "An empirical study of container image configurations and their impact on start times," in *Proceedings of the 23rd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 5 2023, pp. 94–105.

[10] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien, "Real-time serverless: Enabling application performance guarantees," in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, p. 1–6.

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

[12] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, "Faster and cheaper serverless computing on harvested resources," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, p. 724–739.

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

[14] 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*, Jun 2020, pp. 65–76.

[15] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[16] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[17] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, pp. 33–49, 2015.

[18] Project Jupyter, "repo2docker," https://github.com/jupyterhub/repo2docker, accessed: 2024-04-29.

[19] B. Ragan-Kelley, C. Willing, F. Akici, D. Lippa, D. Niederhut, and M. Pacer, "Binder 2.0-reproducible, interactive, sharable environments for science at scale," in *Proceedings of the 17th Python in science conference*. F. Akici, D. Lippa, D. Niederhut, and M. Pacer, eds., 2018, pp. 113–120.

[20] K. Mahajan, S. Mahajan, V. Misra, and D. Rubenstein, "Exploiting content similarity to address cold start in container deployments," in *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*, 2019, p. 37–39.

[21] R. J. Campello, D. Moulavi, A. Zimek, and J. Sander, "Hierarchical density estimates for data clustering, visualization, and outlier detection," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 10, no. 1, pp. 1–51, 2015.

[22] OpenAI, "Openai: Text and code embeddings," 2022. [Online]. Available: https://platform.openai.com/docs/guides/embeddings

[23] A. Abedi and T. Brecht, "Conducting repeatable experiments in highly variable cloud computing environments," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 287–292.

[24] A. Kumari and B. Sahoo, "ACPM: adaptive container provisioning model to mitigate serverless cold-start," *Cluster Computing*, vol. 27, no. 2, pp. 1333–1360, 2024.

[25] M. Shahrad, R. Fonseca, I. n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, 2020.

[26] A. Fuerst and 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, p. 386–400.

[27] R. B. Roy, T. Patel, and 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, p. 753–767.

[28] "AWS lambda quotas," https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html, accessed: 2024-04-29.

[29] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *2018 Usenix Annual Technical Conference*, 2018, pp. 923–935.

[30] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the" micro" back in microservice," in *2018 USENIX Annual Technical Conference*, 2018, pp. 645–650.

[31] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020.

[32] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference*, 2020, p. 1–13.

[33] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Sock: rapid task provisioning with serverless-optimized containers," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, 2018, p. 57–69.

[34] K. Suo, J. Son, D. Cheng, W. Chen, and S. Baidya, "Tackling cold start of serverless applications by efficient and adaptive container runtime reusing," in *2021 IEEE International Conference on Cluster Computing*, 2021, pp. 433–443.

[35] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean lambdas with large libraries," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops*, 2017, pp. 395–400.

[36] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li *et al.*, "Help rather than recycle: Alleviating cold startup in serverless computing through {Inter-Function} container sharing," in *2022 USENIX Annual Technical Conference*, 2022, pp. 69–84.

[37] S. Lee, D. Yoon, S. Yeo, and S. Oh, "Mitigating cold start problem in serverless computing with function fusion," *Sensors*, vol. 21, no. 24, p. 8416, 2021.