Keeping Serverless Computing Alive With Greedy-Dual Caching

Anon

Abstract

Functions as a Service (also called serverless computing) promises to revolutionize how applications use cloud resources. However, it suffers from cold-start problems due to the overhead of initializing code and data dependencies before a function can start executing. Keeping the functions alive and warm after they have finished execution can alleviate the cold-start overhead. Keep-alive policies must keep functions alive based on their resource and usage characteristics, which is challenging due to the diversity in FaaS workloads. Our insight is that keep-alive is analogous to caching, and we show that caching-inspired keep-alive policies can be effective in reducing the cold-start overhead by more than $5\times$ compared to the current cloud policies. We hope that our caching analogy opens the door to more principled and optimized keep-alive policies for future FaaS workloads and platforms.

1 Introduction

Functions as a Service (FaaS) is an emerging and popular cloud computing model, where applications use cloud resources through user defined "functions" that execute application code [11]. By handling all aspects of function execution, including resource allocation, cloud platforms can thus provide a "serverless" computing model where users do not have to explicitly provision and manage cloud resources (i.e., virtualized servers). FaaS employs a fine-grained pricing model allowing applications to pay for the resources they use, and provides many other advantages such as near-infinite horizontal scaling. FaaS services are being offered by all major cloud platforms (such as Amazon Lambda [6], Google Functions [8], and Azure Functions [7]), and are being used by different applications such as web services, API services, parallel and scientific computing, and in machine learning pipelines.

Cloud platforms execute each function invocation in a virtualized execution environment such as a container or a virtual machine (VM). Using virtualization techniques, a single physical server can execute many functions concurrently and safely. Each function invocation entails creating and launching a container or a VM, and fetching and installing the necessary libraries and dependencies, before the function itself can be executed. This "initialization" phase can take non negligible time, and adds to the overall function execution latency observed by the user. Reducing this function "startup overhead" is a key challenge in serverless computing [3, 5, 10, 12].

To mitigate the startup overhead, a common technique is to keep the execution environment alive or "warm" for a small duration, so that future invocations of the same function do not incur the cold-start overhead. Keeping functions warm can reduce the function latency by up to $10\times$. However, keeping the execution environment alive and running, instead of immediately terminating it, has some drawbacks. Keeping a container or a VM alive consumes computing resources on the physical servers, and reduces the number of functions that can be concurrently executed. Thus, while keep-alive can reduce the effective function execution latency, it can also reduce the overall system utilization and efficiency.

In this paper, we explore this tradeoff, and argue for and develop principled keep-alive techniques. Current cloud platforms and FaaS systems employ simple keep-alive policies. For example, AWS Lambda will keep all functions "warm" for the maximum duration of the function, which is currently 15 minutes [2, 4, 9]. These simple keep-alive policies are not ideally suited for handling a diverse range of functions with different initialization overheads, resource footprints (i.e., CPU and memory consumption), and request frequencies. An optimized keep-alive policy must balance the overhead of keeping a function warm with the likelihood that the function will be called again in the near future, and the duration of keep-alive should depend on the function's characteristics.

Our primary insight is that the problem of which function to keep alive/warm, is analogous to object caching. Keeping a function warm is equivalent to caching an object, and a warm function execution is equivalent to a cache hit. Terminating a function's execution environment means that the next invocation will incur the cold-start penalty, and is thus equivalent to evicting an object from a cache. The objective is to keep functions warm such that the effective function latency is reduced, which is equivalent to caching's goal of reducing object access time. *By mapping keep-alive to the exhaustively studied caching problem, we can leverage principles and techniques from caching, and apply them to serverless computing.*

Specifically, we use and adapt the "Greedy Dual" caching framework [**?**], and develop a keep-alive policy based on it. Our keep-alive policy is cognizant of the memory footprint, access frequency, and execution latency of different functions. We show that this Greedy-Dual approach can improve both the function latency and the server utilization compared to the current simple keep-alive policies.

2 FaaS Keep-alive Background

Serverless computing is now being provided by all large public cloud providers: Amazon Lambda, Google Functions, and Azure Functions are becoming an increasingly popular way to deploy applications on the cloud. Functions as a Service (FaaS) can also be realized on private clouds and dedicated clusters through the use of frameworks such as OpenWhisk, OpenFaas, OpenLambda, etc. In this new cloud paradigm, users provide functions in languages such as Python, Javascript, Go, and others. The functions are executed by the FaaS platform, greatly simplifying resource management for the application.

However, the execution of FaaS functions entails performance overheads that we must be cognizant of. FaaS functions cannot assume that state will persist across invocations, and function definitions must first import and load all code and data dependencies on each execution. Each functions is run inside a containers such as Docker or runC, or a lightweight VM such as Firecracker. By encapsulating all of the function state and any side-effects, the virtual execution environment provides isolation among multiple functions, and also allows for concurrent invocations of the same function. Due to the overhead of starting a new virtual execution environment (i.e., container or VM), and initializing the function by importing libraries and other data dependencies, function execution thus incurs a significant "cold-start" penalty. Thus, FaaS can result in significant performance (i.e., total function execution latency) overheads compared to conventional models of execution where applications can reuse state and do not face the high initialization overheads.

Two main techniques are used to alleviate the cold-start penalty. Once a container for a function is created and the function finishes execution, the container can be kept alive instead of immediately terminating it. Subsequent invocations of the function can then *reuse* the already running container. This *keep-alive* mechanism can alleviate the cold-start overhead due to container launching (which can be ~ 100 ms). The second technique for reducing the cold-start overhead is to explicitly initialize functions before running them, and resolving most of the function's code and data dependencies during the initialization phase.

An example of function initialization is shown in Figure 1,

```
1   #Initialization code
2   import numpy as np
3   import tensorflow as tf
4
5   m = download_model('http://model_serve/img_classify.pb')
6   session = create_tensorflow_graph(m)
7
8   def lambda_handler(event, context):
9       #This is called on every function invocation
10      picture = event['data']
11      prediction_output = run_inference_on_image(picture)
12      return prediction_output
```

Figure 1: Initializing functions by importing and downloading code and data dependencies can reduce function latency by hiding the cold-start overhead.

which shows a pseudo-code snippet of a function that performs machine learning inference on its input. For ML inference, the function downloads an ML model and initializes the TensorFlow ML framework (lines 5 and 6). If the function's container is kept alive, then invocations of the function do not need to run the expensive initialization code (lines 2–6). Thus, the execution latency of functions can be minimized with a combination of careful function initialization and keeping the containers alive.

However, keep-alive is not a panacea for all FaaS latency problems. Keeping a container alive consumes valuable computing resources on the servers, and reduces the number of functions that can be executed concurrently. Specifically, a running container occupies memory, and "warm" containers being kept alive in anticipation of future function invocations can reduce the multiplexing and efficiency of the servers.

Thus, we require keep-alive *policies* that reduce the cold-start overhead while keeping the server utilization high. Designing keep-alive policies is not trivial due to the highly diverse and expanding range of applications that are using FaaS platforms. Conventionally, FaaS has been used for hosting web services, which is attractive because of the pay-per-use properties. Event handling functions for web responses typically have a small memory footprint but require low execution latency. On the other hand, FaaS is also being used for "heavy" workloads with high memory footprint and large initialization overheads such as highly parallel numerical computing (such as matrix operations [], scientific computing [], and machine learning. Keep-alive policies must therefore balance the resource footprint of the containers with the benefits of keeping containers alive—and do so in manner that is applicable across a wide range of applications.

3 Keep-alive Policies

In this section, we discuss the system model for function keep-alive, the goals and considerations for keep-alive policies, and present our caching-inspired greedy-dual policy.

System model: We assume that each function invocation runs

Application	Mem size	Run time	Init. time
ML inference	2 GB	2 min	1 min
Video Encoding	200 MB	20 s	200 ms
Matrix Multiply	80 MB	770 ms	110 ms
Web-serving	100 MB	100 ms	10 ms

Table 1: FaaS workloads are highly diverse in their resource requirements and running times.

in its own container. A FaaS platform may use a cluster of physical servers, and forward the function invocation requests to different servers based on some load-balancing policy. Our aim is to investigate general keep-alive techniques that are independent of the load-balancing, and we therefore focus on *server-level* policies. Even on a single server, a function can have multiple independent and concurrent instantiations, and hence containers. Each function has its own container image and initialization code, and thus containers cannot be used by multiple functions. A function's containers are nearly identical in their resource utilization, since they are typically running the same function code. When a function finishes execution, its container may be terminated, or be kept alive and "warm" for any future invocations of the same function. At any instant of time, each container is either running a function, or is being kept alive/warm (see Figure **??**). Thus, server resources are consumed by running containers, and containers being kept alive in anticipation for future invocations.

3.1 Policy Goals and Considerations

The primary goal of keep-alive is to reduce the initialization and cold-start latency, by keeping functions alive for different durations based on their characteristics. Because servers run hundreds of short lived functions concurrently, keep-alive policies must be generalizable and yield high server utilization. Functions can have vastly different characteristics, and keep-alive polices must work efficiently in highly dynamic and diverse settings.

We have identified the following characteristics of functions that are the most pertinent for keep-alive policies. The **initialization time** of functions can vary based on the code and data dependencies of the function. For example, a function for machine learning inference may be initialized by importing large ML libraries (such as tensorflow, pytorch, etc.), and fetching the ML model, which can be hundreds of megabytes in size and take several seconds to download. Functions also differ in terms of their **total running time**, which includes the initialization time and the actual execution time. Again, functions for deep-learning inference can take several seconds, whereas functions for HTTP servers and microservices are extremely short-lived (few milliseconds). The **resource footprint** of functions comprises of their CPU, memory, and I/O use, and also differs widely based on the application's requirements. Finally, functions have different **popularities**, and are called with different rates. Some functions may be invoked several times a second, whereas other functions may only be invoked rarely (if they are used to serve a very low-traffic web-site, for instance).

Because server resources are finite, it is important to prioritize functions that should be kept alive, based on the aforementioned characteristics. A function which is not popular and is unlikely to be called again in the near future, sees little benefits from keep-alive. In fact, keeping such functions alive consumes valuable server computing resources for no gain in efficiency. Thus, keep-alive policies should prioritize popular functions. Similarly, the resource consumption of the functions is also important: since keeping large-footprint functions alive is more expensive than smaller functions, smaller functions should be preferred and kept alive for longer. Finally, functions can also be prioritized based on their initialization overhead, since it is effectively wasted computation.

The problem of designing keep-alive policies is complicated by the fact that functions may have vastly different keep-alive priorities for the different characteristics. Consider a function with a large memory footprint (like those used in ML inference), high initialization overhead, and a low popularity. Such a function should have a low priority due to its size, high priority due to large initialization overhead, and a low priority due to its low popularity. Thus, keep-alive policies must carefully balance all the different function characteristics and prioritize them in a coherent manner.

Current cloud FaaS offerings such as AWS Lambda, and frameworks such as OpenWhisk, keep all functions alive for a *constant* period of time (30 minutes in case of Lambda). This constant "time-to-live" policy is agnostic to different function characteristics such as resource footprint and initialization overheads, and only loosely captures popularity. A more principled approach is needed, which we provide next.

3.2 Keep-alive is Equivalent to Caching

Formulating a keep-alive policy that balances the characteristics of all competing functions, seems daunting. *The central insight of this paper is that keeping functions alive is equivalent to keeping objects in a cache.*

Keeping a function alive reduces its effective execution latency, in the same way as caching an object reduces its access latency. When all server resources are fully utilized, the problem of which functions *not* to keep alive is equivalent to which objects to *evict* from the cache. The high-level goal in caching is to improve the distribution of object access times, which is analogous to our goal of reducing the effective function latencies.

This caching analogy provides us a framework and tools for understanding the tradeoffs in keep-alive policies, and improving server utilization. Caching has been studied in wide range of contexts and many existing caching techniques can be applied and used for function keep-alive. Our insight is that we can use classic observations and results in object caching to formulate equivalent keep-alive policies, that can provide us with well-proven and sophisticated starting point for understanding and improving function keep-alive.

3.3 Greedy-Dual Keep-Alive Policy

While many caching techniques can be applied to the function keep-alive policies, we now present one such caching-inspired policy that is simple and yet captures all function characteristics and their tradeoffs. Our policy is based on Greedy-Dual-Size-Frequency object caching [**?**], which was designed for caches with objects of different sizes, such as web-caches. Classical caching policies such as LRU or LFU do not consider object sizes, and thus cannot be completely mapped to the keep-alive problem where the resource footprint of functions is an important characteristic. As we shall show, the Greedy-Dual-Size-Frequency approach provides a general framework to design and implement keep-alive policies that are cognizant of the frequency and recency of invocations of different functions, their initialization overheads, and sizes (resource footprints).

Fundamentally, our keep-alive policy is a function *termination* policy, just like caching focuses on eviction policies. Our policy is resource conserving: we keep the functions warm whenever possible, as long as there are available server resources. This is a departure from current constant time-to-live policies implemented in public clouds, that are *not* resource conserving, and may terminate functions even if resources are available to keep them alive for longer.

Our policy decides which container to terminate if a new container is to be launched and there is not enough resource availability. The total number of containers (warm + running) is constrained by the total server physical resources (cpu and memory). Intuitively, our termination policy computes a "priority" for each container based on the cold-start overhead and resource footprint, and terminates the container with the lowest priority.

Priority Calculation. Our greedy dual policy is based on greedy dual caching [**?**]. For each container, we assign a *keep-alive priority*, which is computed based on the frequency of function invocation, its running time, and its size. The priority is given by:

$$\text{Priority} = \text{Clock} + \frac{\text{Freq} \times \text{Cost}}{\text{Size}} \tag{1}$$

On every function invocation, if a warm container for the function is available, it is used, and its frequency and priority are updated. Reusing a warm container is thus a "cache hit", since we do not incur the initialization overhead. When a new container must be launched if there are insufficient resources, then containers are terminated based on their priority order—lower priority containers are terminated first. We now explain the intuition behind each parameter in the priority calculation:

Clock is used to capture the recency of execution. We maintain a "logical clock" per server. Each time a container is used, the server clock is assigned to the container and the priority is updated. Thus, containers that are not recently used will have smaller clock values (and hence priorities), and will be terminated before more recently used containers.

The logical clock is updated only when a container is terminated. Containers are terminated only if there are insufficient resources to launch the new container and if existing warm containers cannot be used. Specifically, if a container j is terminated (because it has the lowest priority), then $\text{Clock} = \text{Priority}_j$. All subsequent uses of other, non-terminated containers then use this clock value for their priority calculation. In some cases, *multiple* containers may need to be terminated to make room for new containers. If E is the set of these terminated containers, then $\text{Clock} = \max_{j \in E} \text{Priority(j)}$

We note that the priority computation is on a per-container basis, and containers of the same function share some of the attributes (such as size, frequency, and cost). However, the clock attribute is updated for each container individually. This allows us to evict the oldest and least recently used container for a given function in order to break ties.

Frequency is the number of times a given function is invoked. A given function can be executed by multiple containers, and frequency denotes the *total* number of function invocations across all of its containers. The frequency is set to zero when all the containers of a function are terminated. The priority is proportional to the frequency, and thus more frequently executed functions are kept alive for longer.

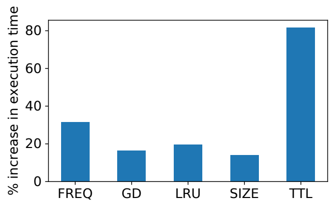
Cost represents the termination-cost, which is equal to the initialization time, i.e., s_i. This captures the benefit of keeping a container alive and the cost of a cold-start. The priority is thus proportional to the initialization overhead of the function.

Size is the resource footprint of the container. The priority is inversely proportional to the size, and thus larger containers are terminated before smaller ones. We can use multi-dimensional resource vectors to represent the size, in which case we convert them to scalar representations by using the existing formulations from multi-dimensional *bin-packing*. For instance, if the container size is $\mathbf{d_i}$, then the size can be represented by the magnitude of the vector $||\mathbf{d_i}||$. Other size representations can also be used. A common technique is to normalize the container size by the physical server's total resources ($\mathbf{a}$), and then compute the size as $\sum_j \frac{d_j}{a_j}$ where d_j, a_j are the container size and total resources of a given type (either CPU, memory, I/O) respectively. Cosine similarity between $\mathbf{d}$ and $\mathbf{a}$ can also be used, as is widely used in multi-dimensional bin-packing.

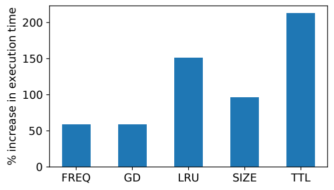
In most scenarios, the number of containers that can run is limited by the physical memory availability, since CPUs can be multiplexed easily, and memory swapping can result in severe performance degradation. Thus for ease of exposition and practicality, we can consider only the container *memory* use as the size, instead of a multi-dimensional vector.

4 Evaluation

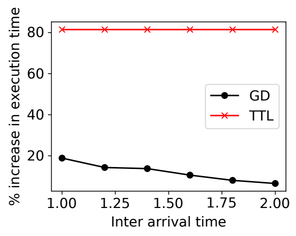
We now present preliminary evaluation of our caching-inspired keep-alive policy. We have implemented our Greedy-Dual keep-alive policy described in the previous section, as well as its various variants. We evaluate all keep-alive poli-



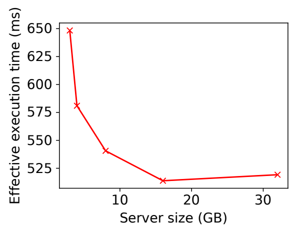
(a) Heterogeneous functions.



(b) Different initialization times.



(c) Our policy provides better performance at higher inter-arrival times.



(d) The "Miss-rate curve" with our Greedy-Dual policy.

Figure 2: Keep-alive policy comparison.

cies in a simulation framework that allows us to evaluate their effectiveness for different mixes of functions with different initialization times, memory sizes, invocation frequency, and inter-arrival times. *We are primarily interested in the increase in the effective execution time of functions due to cold starts.* To understand the tradeoffs and differences between various caching-inspired keep-alive policies, we examine different variants of the Greedy-Dual policy:

GD: Greedy-Dual policy described in the previous section

FREQ: Priority=Clock + Frequency. Ignores initialization cost and size.

LRU: Classic least recently used based termination. Ignores size and cost.

SIZE: Considers recency and size, and ignores frequency and cost.

TTL: AWS Lambda policy with a fixed "time to live" of 30 minutes.

Heterogeneous functions. We first evaluate the effectiveness of keep-alive policies when there is a large heterogeneity in function characteristics. This is a key characteristic of FaaS platforms, since a single physical server may run 100s of different kinds of functions. We use four function classes that are representative of commonly deployed FaaS functions as shown in Table **??**. The function memory sizes range from $[10, 1000]$ MB, and running times ranging from $[10, 1000]$ ms. Figure 2a shows this increase in execution time for the various keep-alive policies. We can see that the function execution time for the various caching-inspired policies is much lower (by up to $4\times$) than the constant-TTL policy being used by current FaaS platforms.

Interestingly, for the above function class configuration, we see that simpler policies such as SIZE are as effective as the full Greedy-Dual policy. The SIZE policy simply terminates the largest containers first—this is extremely effective in our heterogeneous configuration that has very large (1000 MB) containers, which is $100\times$ bigger than the smallest container. In cases of such extreme size heterogeneity, terminating a single big container can make room for lots of other smaller containers, and thus SIZE can be a competitve keep-alive policy.

Different Initialization Overheads. Minimizing the initialization overhead is a crucial task of keep-alive policies. In this experiment (Figure 2b), we consider two function classes that are identical in every way except their initialization overhead (which is in a $1:4$ ratio). Because the GD policy considers the initialization overhead when computing the keep-alive priority, it significantly outperforms other policies such as LRU and constant-TTL by $3\times$ and $4\times$ respectively.

We note that based on the workload, simpler policies may match or even outperform our GD policy. For example, SIZE and LRU in Figure 2a and FREQ in Figure 2b. However, the strength of the GD policy is in its generality, and its ability to consistently work well in many different scenarios.

One particular scenario where constant-TTL suffers is when function executions are "rare", and their inter-arrival times is longer than the fixed TTL. In such cases, the containers are terminated, and the benefit of keep-alive is lost. Figure 2c shows the performance of the GD and TTL schemes for different inter-arrival times, that are scaled and normalized. We can see that as the inter-arrival times increase, the function latencies decrease with GD caching. However, constant-TTL shows a constant function latency which is more than $8\times$ that of GD.

The caching analogy can also be used by FaaS providers for capacity planning. Figure 2d shows the effective execution time as a function of the server size, which equivalent of a "miss ratio curve". We can see that the server size shows diminishing returns, similar to the classic caching behavior.

5 Discussion

In this paper, we have focused on the keep-alive policies for FaaS systems. However, the effectiveness and validity of these policies fundamentally depends on many uncertain aspects such as FaaS usecases, abstractions, workloads, virtualization techniques, and cloud pricing models. Ongoing developments in research and production FaaS platforms will affect how these aspects evolve, and understanding their impact requires community-wide discussion.

FaaS Workload Characterization. Our preliminary evaluation uses synthetic traces partially informed by FaaS application characteristics. However, there is a dearth of FaaS workload traces and benchmarks that can be used to drive realistic and empirically sound research. An empirical understanding of the diverse FaaS applications and their characteristics such as their computational requirements, initialization overhead, and inter-arrival times, will be key in developing, understanding, and evaluating more sophisticated keep-alive policies. Empirically informed FaaS workload traces can play a similar crucial role as YCSB [] or the Google data center traces []. To overcome the challenge of constantly evolving applications and platforms, it may be necessary to develop parametrized workload generators.

Keep-alive mechanisms and virtualization techniques. The cold-start overhead can be also be reduced using many virtualization and container image caching optimizations. Keep-alive mechanisms for emerging special-purpose "light weight VMs"such as Firecracker [**?**] may be an interesting area of future work. While hardware virtualization typically increases the cold-start overhead, it also provides opportunities to use techniques like page deduplication [**?**] and VM forking [**?**], that can reduce the effective footprint of functions and startup times. Keep-alive mechanisms can provide an interesting point of comparison for different virtualization technologies such as containers, VMs, unikernels. In any case, the cold-start overhead depends on underlying keep-alive mechanisms that will change over time, and thus requires general and tunable keep-alive policies such as Greedy-Dual.

Programming and cost models. Given the critical role of function initialization, different programming and execution models may be required. It will be interesting to explore the burden of explicit function initialization (like provided by OpenWhisk) on the application developers. Incentivizing applications to make use of initialization also promises to be an interesting research area. Performance is a good incentive, but do we need to consider new FaaS cost models that are tied to the caching effectiveness? Or will that complicate things and be far removed from the spirit of FaaS? Tiered pricing models in which users pay more for longer keep-alive is another potential solution.

References

[1] . .

[2] AWS Lambda Limits. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

[3] Keeping Functions Warm - How To Fix AWS Lambda Cold Start Issues. https://serverless.com/blog/keep-your-lambdas-warm/.

[4] How long does AWS Lambda keep your idle functions around before a cold start? https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-arc 2017.

[5] Lambda Warmer: Optimize AWS Lambda Function Cold Starts. https://www.jeremydaly.com/lambda-warmer-optimize-aws-lambda-function-cold-start 2018.

[6] AWS Lambda. https://aws.amazon.com/lambda/, 2020.

[7] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, 2020.

[8] Google Cloud Functions. https://cloud.google.com/functions, 2020.

[9] Erwan Alliaume and Benjamin Le Roux. Cold start / warm start with aws lambda. https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/, 2018.

[10] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[11] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv:1902.03383 [cs]*, February 2019. arXiv: 1902.03383.

[12] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. USENIX ATC:14, 2018.