Proactive Serverless Function Resource Management

Erika Hunhoff, Shazal Irshad, Vijay Thurimella, Ali Tariq, Eric Rozner

Abstract

This paper introduces a new primitive to serverless language runtimes called `freshen`. With `freshen`, developers or providers specify functionality to perform before a given function executes. This proactive technique allows for overheads associated with serverless functions to be mitigated at execution time, which improves function responsiveness. We show various predictive opportunities exist to run `freshen` within reasonable time windows. A high-level design and implementation are described, along with preliminary results to show the potential benefits of our scheme.

ACM Reference Format:
Erika Hunhoff, Shazal Irshad, Vijay Thurimella, Ali Tariq, Eric Rozner. 2020. Proactive Serverless Function Resource Management. In *WoSC20: Workshop on Serverless Computing, Dec 07–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Serverless computing is an emerging paradigm in which cloud providers seamlessly scale developer-provided functions as demands change. Although seemingly simple, serverless functions have been shown to support a wide variety of workloads, from chat bots, video processing, machine learning, HCI, to even general compute. As serverless ecosystems mature, functions will be integrated into a set of larger and larger microservices and will also be relied upon to directly interface with users. As such, the execution latency of serverless functions becomes an important consideration.

However, the simplicity of today's serverless deployments may increase execution times. Consider a simple function, λ_1, which downloads a machine learning model from a server, analyzes an input image, and performs additional processing before writing a result to a datastore. Without special care, many overheads exist. After starting, the function must create a connection to the server hosting the model and then download the model from the server. This behavior could happen anew for subsequent instantiations of λ_1, even if running sequentially in the same warmed container. When writing the result, another connection must be established before the data is sent. Again, this overhead could reoccur for each successive invocation of λ_1. These per-invocation overheads (*i.e.,* establishing connections, refetching the model, incurring TCP slow start, etc.) quickly add up, which is problematic because many functions have short execution times.

To deal with such issues, developers can utilize *runtime reuse*. In runtime reuse variables can be *runtime-scoped* inside the language runtime executing within the container the serverless function runs in.[1] Runtime-scoped variables can be accessed across subsequent serverless function instantiations within a given runtime and container. Revisiting our example, network connections can be reused within a runtime when defined as a runtime-scoped variable to avoid per-instantiation connection overheads.

In this paper, we argue runtime reuse is insufficient to overcome many of the redundant overheads described earlier. Even with runtime reuse, fetched data could be out-of-date, connections may revert their congestion windows to small initial values or even time out, or application-level data could be stale from the last invocation. To combat these issues, we propose a new primitive called `freshen`, which can be *proactively* invoked by the serverless infrastructure. A `freshen` hook is implemented within the runtime, allowing developers or providers to establish or warm connections, proactively fetch data, or otherwise perform actions to reduce overheads when the serverless function runs. The `freshen` hook is designed to be run before its corresponding function is instantiated, and we contend this is possible because there are many opportunities to predict a function's instantiation before it is invoked.

This paper provides motivation and background in Section 2, a preliminary design in Section 3, and potential benefits of `freshen` in Section 4. Related Work is detailed in Section 5. Finally, a discussion and conclusion section is presented in Section 6.

2 Background and Motivation

This section provides background on runtime reuse and highlights scenarios where runtime reuse may be inefficient. Then, we motivate ways to predict function instantiations.

Serverless runtime reuse While all providers allow runtime reuse, here we explain how an open-source platform, OpenWhisk, enables reuse. OpenWhisk runs functions within Docker containers, listening as a daemon on port 8080. After the Docker container is initialized, the `init` hook starts

[1] We use "container" to generally refer to VMs or containers

WoSC20, Dec 07–11, 2020, Delft, Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the language runtime within the container and also loads the actual function code. When the `run` hook is invoked, the function will be scheduled to run. Thus, the persistent runtime invoked during `init` can be thought of as a program that listens for the `run` hook, executes the function, and returns the result.

Without runtime reuse, variables are scoped for use within a single invocation only, which we term *invocation-scoped*. In contrast, *runtime-scoped* variables variables can be reused across serverless function instantiations in a given runtime. Common use cases for *runtime-scoped* variables are persistent network connections (ensuring connection quotas are not exhausted) and frequently-accessed data fetched during the first function invocation and then stored in the runtime for the lifetime of the container.

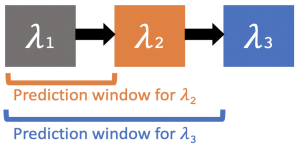

Figure 1. Opportunities for `freshen` within a function chain

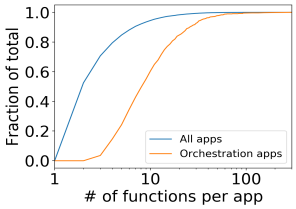

Figure 2. Orchestration apps have more functions in chains

Runtime reuse inefficiencies While runtime reuse can be used by application developers to increase application efficiency, numerous issues may arise. First, there may exist cases in which the runtime has not been initialized, such as when cold starts occur, meaning reuse is not possible. Some studies have shown inefficient container reuse across function invocations, which increases cold start frequency [12]. Other studies indicate some serverless infrastructures disallow container sharing between functions, which can also increase cold starts when container resources are limited [13]. Second, there may be cases when the runtime is initialized, but the data held within the runtime is stale. For example, an object stored within the runtime may need to be retrieved from a datastore because a newer version is available. Network connections may have timed out or have reset their TCP state (*e.g.,* congestion window, round trip times, etc). The Linux congestion control algorithm reduces the congestion window (CWND) on inactive connections. Last, approaches to reduce connection (re)establishment overheads may not apply. The Linux `tcp_no_metrics_save` capability allows metrics like RTT and ssthresh to be cached between TCP connections to the same destination, but does not apply to important parameters such as CWND. TCP Fast Open requires sender/receiver support and limits the amount of data sent in initial handshakes to small amounts. As a result, we believe several inefficiencies remain, even with runtime reuse, that can be addressed with `freshen` called proactively before function instantiations.

Regaining efficiency via prediction To alleviate the above concerns, we introduce a `freshen` hook into the runtime, which can be called before a function is set to run. The `freshen` hook allows providers or developers to execute arbitrary code intended to speed up function execution times. `freshen` could warm pre-existing network connections, ensure locally-cached items are up-to-date, or even proactively retrieve a needed object. In order for `freshen` to be effective, we must be able to predict when a function may run. There are several cases in which prediction may be possible. First, in serverless function chains, such as in Figure 1, explicit knowledge of a serverless function chain could predict impending function invocations within the chain. Function chains are often explicitly provided (as in Orchestration frameworks like AWS Step Functions) or can be derived via tracing or service mesh techniques [6]. To better understand prediction opportunities, we briefly study function chains in Orchestration frameworks. Figure 2 shows a CDF of the number of functions within a single serverless application for Orchestration applications on Azure (data from traces in [9]), compared to the number of functions within a single application over all applications. Orchestration frameworks are specifically designed to support function chains, and hence applications utilizing Orchestration frameworks typically consist of more functions: 8 functions in the median Orchestration case versus 2 functions in the median case of all. Considering a median function runtime of ~700ms [9], opportunities for prediction could be as high as ~5.6s in the extreme case of a linear chain dependency (as in Figure 1).

In addition, functions within chains may be triggered by other services, such as a storage trigger, a pub/sub trigger, or a direct invocation. Table 1 shows the median delay, over 20k runs, between invoking a function via the listed service and the actual subsequent triggered function start time in AWS. Cold starts are carefully avoided, and the methodology in [12] is used to obtain overheads by measuring timestamps just before the function trigger and at the start of the triggered-function. The table shows latencies range from 60ms to 1.28s, allowing time for the previous function within the chain, or the serverless provider, to call and execute `freshen` on the next function within the chain.

Trigger Service	Delay (s)
Step Functions	0.064
Direct (Boto3)	0.060
SNS Pub/Sub	0.253
S3 bucket	1.282

Table 1. Trigger overhead

3 Design and Implementation

The following sections address when a `freshen` hook could run (Section 3.1), what a `freshen` hook could do (Section

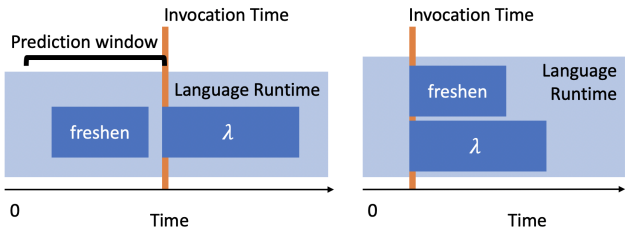

Figure 3. Predicted (left) and unanticipated (right) timing of `freshen`

3.2), and how `freshen` could be implemented (Section 3.3). Throughout, we will refer to λ, an example serverless function shown in pseudocode in Algorithm 1, to illustrate how `freshen` could warm a connection and prefetch data. λ first fetches some data (`DataGet`) over a network connection, performs some calculation based over the fetched data and the λ's parameters, writes an output value to an external resource (`DataPut`), and finally returns whether the write was successful or not.

Algorithm 1 Sample Serverless Function λ

```
1: Runtime Constants: CREDS, ID_1, ID_2
2: procedure λ(args)
3:     data := DataGet( CREDS, ID_1 )
4:     ...
5:     result := ...
6:     ...
7:     ret := DataPut( CREDS, ID_2, result )
8:     return ret
9: end procedure
```

3.1 When to *freshen*

The serverless framework would attempt to run `freshen` before the serverless function (best case) or simultaneously (worst case). `freshen` would be non-blocking and run within a separate thread in the language runtime so the logic and timing of function invocation via the `run` hook is unmodified. Figure 3 shows the two examples of when `freshen` could run in relation to the λ it is freshening.

3.2 Opportunities to *freshen*

`freshen` could perform a variety of actions, including TCP connection establishment, TCP connection warming, state maintenance of other connection-oriented protocols, and proactive data fetching.

Connection establishment and checks If a serverless function uses a resource with an underlying TCP connection, the function developer can either establish a runtime-scoped socket connection to take advantage of runtime reuse or create the connection as an ephemeral invocation-scoped variable. In both cases, `freshen` could help reduce function latency. If the connection is runtime-scoped, `freshen` would send a TCP keepalive to ascertain connection liveness; if the connection is not alive, `freshen` could reestablish the connection. If the connection is invocation-scoped, `freshen` could proactively establish the connection before the function attempts to create it.

`freshen` would only be able to perform connection establishment for connections with constant arguments (*e.g.,* constant IP and port). We posit this is often the case, as serverless functions often interact with known services such as storage.

Connection warming `freshen` could also take steps to warm TCP connections used by the serverless function, for instance setting the CWND. This could be facilitated via a new system call, `warm_cwnd`, which would determine an appropriate value of CWND based on current network conditions and anticipated workloads. The CWND can be estimated via techniques like packet pair probing to determine the current bandwidth [5] or analyzing the CWND of recent similar TCP connections to the same destination. Repetitive invocations can be used to anticipate workload characteristics, which could guide the warming function on whether warming is appropriate. The `warm_cwnd` function can set initial congestion windows or alter congestion windows on longer-running, inactive connections. Since `warm_cwnd` is implemented as a system call, final determination of actual CWND values, as well as permissions on whether such values can be altered, resides within the provider who is running the underlying host infrastructure.

Other connection-oriented protocols `freshen` can establish and warm other connection-oriented protocols and protocols that run on top of TCP such as TLS, as long as the credentials are constant. However, for TLS establishment and other user-space protocols, the serverless provider would require some knowledge of the libraries in order to create provider-generated `freshen` hooks for those resources. Developers who write their own `freshen` hooks, as detailed in Section 3.3, would have access to such knowledge.

Proactive data fetching Consider the λ in Algorithm 1: if the data fetched with `DataGet` is retrieved using constant credentials and resource identifiers, it is possible to prefetch the data before λ is invoked.

Prefetching leads to the concept of a `freshen`-maintained cache of prefetched data. If the function is invoked frequently within the same runtime and accesses a read-only data resource, it may only be necessary to fetch the data once every n seconds instead of every time the function is run, reducing network traffic. The time-to-live (TTL) of values within the `freshen` cache could be set by a default value, by `freshen` configuration values specified by the function developer, or by modifying the `DataGet` library to configure the TTL

value on a per-resource level. In the more general case, associated timestamps or version numbers could be used to determine the freshness of items in the runtime `freshen` cache, and data could be updated the next time `freshen` or the serverless function is called.

3.3 Implementation

In the simplest implementation of `freshen`, the function developer would write `freshen` for each serverless function that requires optimization. This would provide the most opportunity for customized optimization. As an interesting alternative, for common resources and for popular serverless languages (*e.g.,* JavaScript, Python) `freshen` code could be inferred by the serverless framework itself.

Code generation would be complex but here we rely on several observations about serverless functions and frameworks to reduce the scope of the problem:

- If `freshen` were unable to be inferred, the serverless framework could continue unmodified with no major performance loss. Hence, failure to infer is not fatal.
- Source code is available for static analysis for such tasks as identification of read-only data fetched using constant parameters.
- Identical function code is run multiple times, so dynamic tracing of functions to identify commonly accessed resources is possible (similar to the tracing technique presented in [4]).
- The latency cost of the network operations `freshen` seeks to optimize are much slower than CPU speeds so some overhead for learning how to infer `freshen` is permissible.
- Implementing inference only for libraries used to access other cloud services offered by the serverless provider has the potential to lower latency for a majority of functions without having to infer `freshen` behavior for unknown resources.

One option for implementing `freshen` for scripting languages is to use added runtime-scoped state and dynamically-inserted wrapper functions. The purpose of the runtime-scoped state is to track and coordinate `freshen` resources between the `freshen` call and the actual function invocation. The purpose of the dynamically-inserted wrappers is to intercept access to freshened resources. We will illustrate a simplified example of what an inferred `freshen` could resemble for λ in Algorithm 1.

The runtime-scoped state would minimally be a collection of ordered `freshen` resources. A *freshen resource* is any object or resource that the `freshen` code may interact with, such as a socket or a data object. In our example, the `freshen` resources are kept in an ordered runtime-scoped list called fr_state. In Algorithm 1, the `DataGet` operation which `freshen` can fetch or prefetch, will be assigned index 0 since it is the first resource accessed by λ. `DataPut`, which `freshen` can warm, is assigned index 1. Each entry in `fr_state` could contain a variety of metadata, such as a *state* (*e.g.,* `running`, `finished`, etc.), a *result* (*e.g.,* the prefetched data), a *TTL* for the result, and a *timestamp* recording the last time that entry was freshened. For simplicity, we only consider *state* and *result* in the following algorithms.

Algorithm 2 Freshen Function for λ

```
1:  Runtime State: fr_state
2:  procedure Freshen
3:      fr_state[0] := running
4:      fr_state[0].result := DataGet( CREDS, ID_1 )
5:      fr_state[0] := finished
6:      fr_state[1] := running
7:      DataPut.warm( CREDS )
8:      fr_state[1] := finished
9:      return
10: end procedure
```

Algorithm 2 illustrates an example `freshen` function for λ. As mentioned, `DataGet` is assigned to index 0 and `DataPut` is assigned to index 1. The states *running* and *finished* surround the `DataPut` and `DataGet` calls of `freshen`, and are used to coordinate the execution of `freshen` with the execution of λ. Algorithm 3 is the annotated version of Algorithm 1. The function wrappers appear at lines 3 and 7. The function wrappers used are `FrFetch` (for *freshen fetch*) and `FrWarm` (for *freshen warm*).

Algorithm 3 Annotated Sample Serverless Function

```
1: Runtime Constants: CREDS, ID_1, ID_2
2: procedure λ(args)
3:     data := FrFetch( 0, DataGet( CREDS, ID_1 ))
4:     ...
5:     result := ...
6:     ...
7:     ret := FrWarm( 1, DataPut( CREDS, ID_2, result ))
8:     return ret
9: end procedure
```

Algorithms 4 and 5 are the implementations of those wrappers. The main function of each wrapper is to synchronize `freshen` actions with λ's use of that resource. If the resource has already been freshened, the wrapper returns either the prefetched data (line 4 in Algorithm 4) or nothing where `freshen`'s only job is to warm the resource (line 4 in Algorithm 5). In Algorithm 5 it is assumed that there is already some knowledge of how to warm `DataPut` (*e.g.,* the call to `DataPut.warm()` in line 7 of Algorithm 2). If `freshen` has started freshening the resource (indicated by the state `running`), both wrapper functions wait for the

freshen thread to finish before returning (line 6 in Algorithm 4 and line 6 in Algorithm 5). Finally, if freshen either did not run or is executing slower than λ, the wrapper can perform the freshen action itself (line 10 Algorithm 4 and line 10 in Algorithm 5). Not included for brevity in Algorithm 2 are the checks to see if the resources have already been freshened by wrapper functions invoked by λ.

Algorithm 4 Freshen Fetch Function

```
1:  Runtime List: fr_state
2:  procedure FrFetch(id, code)
3:      if fr_state[id] == finished then
4:          return fr_state[id].result
5:      else if fr_state[id] == running then
6:          FrWait( id )
7:          return fr_state[id].result
8:      else
9:          fr_state[id] = running
10:         fr_state[id].result = Execute( code )
11:         fr_state[id] = finished
12:         return fr_state[id].result
13:     end if
14: end procedure
```

Algorithm 5 Freshen Warm Function

```
1:  Runtime List: fr_state
2:  procedure FrWarm(id, resource)
3:      if fr_state[id] == finished then
4:          return
5:      else if fr_state[id] == running then
6:          FrWait( id )
7:          return
8:      else
9:          fr_state[id] = running
10:         resource.warm()
11:         fr_state[id] = finished
12:         return
13:     end if
14: end procedure
```

Billing and accounting Since freshen runs in order to benefit the serverless application, the serverless application owner should pay for it. However, as outlined above, freshen would ideally be triggered based on predictions by the serverless framework. What happens if the platform mispredicts a function call? Confidence in prediction could be used to dictate if freshen is called or not. Metrics kept inside a container, or communicated to the serverless global scheduling entity, could be used to stop freshen from running if predictions have been too inaccurate. Service categories chosen by the application developer could also control freshen behavior: aggressive freshen invocation would be appropriate for latency-sensitive applications. freshen could be disabled for latency-insensitive functions. Last, we note providers may be incentivized to offer freshen because it allows them a way to monetize warmed containers that are otherwise sitting idle.

Preventing abuse and misconfiguration A danger if the application developer were allowed to implement their own freshen is that the application developer would try to implement their entire function in the freshen function. This is undesirable and unprofitable for the developer for several reasons: freshen has no access to function arguments, the application developer is paying for the compute and network resources regardless, and the application would have to handle spurious invocations (mispredictions) gracefully.

4 Evaluation

This section explores the advantages a freshen framework could provide. First, the benefits of file caching are evaluated and then improvements from connection warming are illustrated.

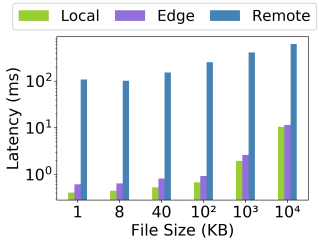

Figure 4. File retrieval overheads to save with freshen

File caching evaluation Figure 4 demonstrates the potential benefits of proactive file retrieval (file caching). In this benchmark, an OpenWhisk serverless function queries a server for a file of one of six different sizes (x-axis) over a TCP connection. The time measured (y-axis, log scale) is the duration from connection to when the file has been completely received. The file server is located in one of three locations: local on-host (green), edge on-site (purple), and remote off-site (blue). On-site resides on the same 10 Gbps LAN and off-site averages 50ms away. The experiment was conducted using CloudLab [2] with 20 iterations. The results show how much execution time freshen could save a serverless function if freshen is proactively run. Maximum benefits range from 11-622ms.

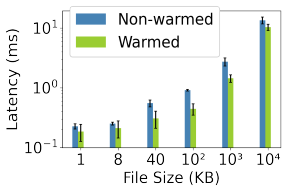

Figure 5. Warming to cloud

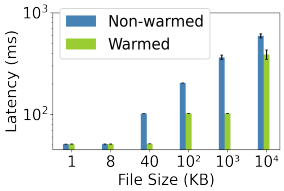

Figure 6. Warming to edge

Warmed connection comparison To demonstrate the benefits of `freshen` warming a TCP connection, we run an OpenWhisk serverless function on CloudLab which sends different file sizes to a server. We measure the time of a client initiating a file transfer to the response from the server indicating completion. To understand the potential benefits, we emulate a warmed TCP connection by sending a large file before sending our desired file size. The server is located at two locations, on the same cloud or at the edge (~50ms away). The experiment was conducted over 20 iterations. The cloud case is presented in Figure 5 and the edge case is presented in Figure 6. With smaller file sizes, the performance of warmed and non-warmed is similar. As file sizes grow, the benefit of warmed connection ranges from 51.22% to 71.94%. The edge performance is better because network delay, and not system overheads, dominate totals.

5 Related Work

A large body of research focuses on reducing cold start costs. We partition these works into two categories: those that do not propose radical changes to serverless architecture, and those that do. Of those that are compatible with existing serverless infrastructure, techniques include cold start avoidance (runtime reuse), light-weight isolation mechanisms [8], caching resources ranging from libraries [8] to virtual Ethernet infrastructure [7], and intelligent host scheduling [11]. Our work has a different focus, optimizing warm starts, but is compatible with these techniques. Works that focus on avoiding cold starts by predicting function execution [3, 9, 11] motivate our design because `freshen` would be most effective when function invocations are predicted. Of works that propose fundamental changes to serverless architecture such as running more than one function within the same isolation context [1] or adding distributed application state and/or message passing abilities between serverless functions [1, 10], the motivation for `freshen` remains but implementation strategies would vary.

Last, Containerless [4] avoids the cost of strong isolation mechanisms by transforming JavaScript serverless functions into Rust via dynamic tracing. Their dynamic tracing design, as well as analysis of the resulting traces, could help inform how `freshen` could be inferred by providers.

6 Discussion and Conclusion

Discussion There exists many opportunities for future work. First, the system should be fully deployed and thoroughly evaluated. Quantifying how `freshen` affects variability in application behavior would be an important component of this evaluation. Prediction success must be additionally quantified, especially in the case of non-deterministic function chains. In addition, the framework must be analyzed for misuse and hardened as necessary. Impact on developer burden, or the extent to which providers can automatically generate `freshen` must also be further studied. Finally, integrating `freshen` into serverless architectures that provide different isolation scopes is an additional area for future study (*e.g.,* Azure offers chain-level isolation).

Conclusion This paper proposes a new primitive to serverless language runtimes called `freshen`. With `freshen`, developers or service providers specify functionality to complete before a given function executes. This proactive framework allows for overheads associated with serverless functions to be mitigated at execution time, which improves function responsiveness. We argue predictive opportunities exist to enable `freshen` to be run with ample time. A high-level design and implementation are presented, along with preliminary results to show potential benefits of the scheme.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[2] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[3] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan Chidambaram, Mahmut T. Kandemir, and Chita R. Das. Fifer: Tackling underutilization in the serverless era. arXiv, 2020.

[4] Emily Herbert and Arjun Guha. A language-based serverless function accelerator. arXiv, 2019.

[5] Srinivasan Keshav. Packet-pair flow control. *IEEE/ACM transactions on Networking*, pages 1–45, 1995.

[6] Jonathan Mace and Rodrigo Fonseca. Universal context propagation for distributed system instrumentation. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 8:1–8:18, New York, NY, USA, 2018. ACM.

[7] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[8] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[9] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[10] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, Aug 2020.

[11] Amoghvarsha Suresh and Anshul Gandhi. Fnsched: An efficient scheduler for serverless functions. In *Proceedings of the 5th International*

Workshop on Serverless Computing, WOSC '19, page 19–24, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the Annual Symposium on Cloud Computing (SoCC)*, 2020.

[13] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.