XFaaS: Hyperscale and Low Cost Serverless Functions at Meta

Alireza Sahraei[+], Soteris Demetriou[†], Amirali Sobhgol[+], Haoran Zhang[◇], Abhigna Nagaraja[+], Neeraj Pathak[+], Girish Joshi[+], Carla Souza[+], Bo Huang[+], Wyatt Cook[+], Andrii Golovei[+], Pradeep Venkat[+], Andrew McFague[+], Dimitrios Skarlatos[▽], Vipul Patel[+], Ravinder Thind[+], Ernesto Gonzalez[+], Yun Jin[+], and Chunqiang Tang[+]

[+] Meta Platforms, Inc. [†] Imperial College London [◇] University of Pennsylvania [▽] Carnegie Mellon University

Abstract

Function-as-a-Service (FaaS) has become a popular programming paradigm in Serverless Computing. As the responsibility of resource provisioning shifts from users to cloud providers, the ease of use of FaaS for users may come at the expense of extra hardware costs for cloud providers. Currently, there is no report on how FaaS platforms address this challenge and the level of hardware utilization they achieve.

This paper presents the FaaS platform called XFaaS in Meta's hyperscale private cloud. XFaaS currently processes trillions of function calls per day on more than 100,000 servers. We describe a set of optimizations that help XFaaS achieve a daily average CPU utilization of 66%. Based on our anecdotal knowledge, this level of utilization might be several times higher than that of typical FaaS platforms.

Specifically, to eliminate the cold start time of functions, XFaaS strives to approximate the effect that every worker can execute every function immediately. To handle load spikes without over-provisioning resources, XFaaS defers the execution of delay-tolerant functions to off-peak hours and globally dispatches function calls across datacenter regions. To prevent functions from overloading downstream services, XFaaS uses a TCP-like congestion-control mechanism to pace the execution of functions.

CCS Concepts: • **Computer systems organization → Distributed architectures**; **Cloud computing**; **Reliability**.

Keywords: FaaS, serverless, cloud, systems, measurement.

ACM Reference Format:
Alireza Sahraei[+], Soteris Demetriou[†], Amirali Sobhgol[+], Haoran Zhang[◇], Abhigna Nagaraja[+], Neeraj Pathak[+], Girish Joshi[+], Carla Souza[+], Bo Huang[+], Wyatt Cook[+], Andrii Golovei[+], Pradeep Venkat[+], Andrew McFague[+], Dimitrios Skarlatos[▽], Vipul Patel[+], Ravinder Thind[+], Ernesto Gonzalez[+], Yun Jin[+], and Chunqiang Tang[+]. 2023. XFaaS: Hyperscale and Low Cost Serverless Functions at Meta . In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3600006.3613155

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP '23, October 23–26, 2023, Koblenz, Germany
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613155

1 Introduction

In recent years, Function as a Service (FaaS) [29] has become a popular programming paradigm in Serverless Computing. With FaaS, developers can create individual functions and upload them to a cloud provider's FaaS platform, where the functions are executed in response to external events. The FaaS platform automatically provisions resources to run a function when it is triggered, freeing developers from the burden of managing virtual machines (VM) or containers. Examples of FaaS platforms include AWS Lambda [4], Azure Functions [5], and Google Cloud Functions [17].

However, the ease of use of FaaS for users may come at the expense of extra hardware costs for cloud providers. Prior to FaaS, users were responsible for the cost for underutilized VMs that were over-provisioned to handle intermittent function calls. With FaaS, as the responsibility of resource provisioning shifts from users to cloud providers, the cost of over-provisioned resources will be borne by cloud providers. It was reported that, in Azure Functions, "*81% of the applications are invoked once per minute or less on average. This suggests that the cost of keeping these applications warm, relative to their total execution (billable) time, can be prohibitively high [39].*" Despite FaaS being a popular topic in research, the research community lacks a quantitative understanding of this problem, mainly due to the absence of reports on the actual hardware utilization of large-scale FaaS platforms, let alone solutions to this problem.

At Meta, we operate a hyperscale private cloud that includes a FaaS platform called *XFaaS*. XFaaS processes trillions of function calls per day on more than 100,000 servers spread across tens of datacenter regions. Due to the hyperscale of XFaaS, reducing hardware costs is a top priority for us. Below, we first describe the risk of extra hardware costs and then explain how we tackle them in XFaaS.

```
INITIALIZATION PHASE:
 (1) Start the VM.
 (2) Fetch the container image and the function's code.
 (3) Initialize the container.
 (4) Start the language runtime such as Python or PHP.
 (5) Load common libraries into memory.
 (6) Load the function code into memory.
 (7) Optionally, do JIT compilation.

INVOCATION PHASE:
 (8) Invoke the function multiple times as needed.

SHUTDOWN PHASE:
 (9) Stop the container if it receives no requests for X minutes
     (X=10/20/10 minutes for AWS/Azure/OpenWhisk respectively).
 (10) Optionally, stop the VM.
```

Figure 1. Lifecycle of a function.

1.1 Risk of Extra Hardware Costs

Lengthy cold start time. A significant challenge of FaaS is the lengthy cold-start time of a function, which can result in prolonged response times and hardware inefficiencies. Figure 1 shows the lifecycle of a function, where steps (1)-(7) and (9)-(10) are all overheads and their costs are borne by the cloud provider. Only step (8) performs the actual work that can be charged to FaaS users. If the wait time X in step (9) is too long, the container sits idle and is wasted. Conversely, if the wait time is too short, the container is shut down too quickly, and the next request must go through the overheads in steps (1)-(7) to initialize the function again.

High variance of load. A high variance in the load can result in either extra hardware costs due to over-provisioning, or an overloaded system when the load surges. Shahrad et al. reported that the peak-to-trough ratio of function calls in Azure Functions is approximately two [39]. This ratio is even more skewed in XFaaS, as high as 4.3 (see the "*Received*" curve in Figure 2). Therefore, XFaaS runs the risk of severe waste if resources are provisioned to handle the peak load.

Overloading downstream services. Even if a FaaS platform can perfectly manage its own load, its functions can still cause resource contention at the downstream services that they invoke. For instance, in XFaaS, many functions do offline computation on our social-graph database [9], which is also accessed by our online services that serve billions of users. In the past, we experienced outages caused by a spike in calls from non-user-facing functions overloading the database and resulting in a high error rate for user-facing online services. Existing systems like AWS Lambda set a static concurrency limit per function, which is insufficient. If the limit is set too low, both the FaaS platform and the database will waste unused resources. Conversely, setting the limit too high may result in a high error rate for the online services.

1.2 Solutions for Extra Hardware Costs

Solution for cold start time. To maximize hardware efficiency, we want to achieve or closely approximate the effect of a *universal worker*, where every worker can instantly execute every function without any cold start overhead.

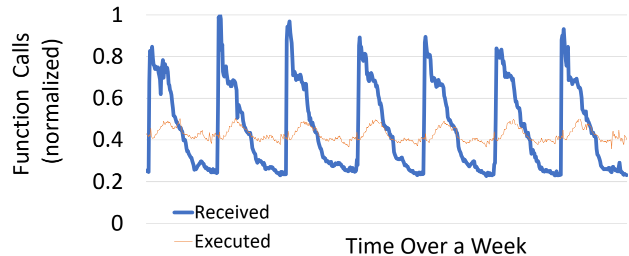

Figure 2. Received vs. executed function calls per minute.

We use several techniques to eliminate the cold start time of functions. First, XFaaS proactively pushes the latest code of all functions of the same programming language (e.g., PHP) to the local SSD disk of all servers that execute functions of this language. These servers keep the language runtime up and running all the time. Upon receiving a call for a function for the first time, the runtime loads the pre-populated function code from its local SSD and executes it immediately, skipping steps (1)-(5) in Figure 1.

Second, XFaaS runs multiple functions concurrently in one Linux process, enabled by the high degree of trust in a private cloud. Async enforces resource isolation through fine-grained quotas and ensures data isolation using a Bell-La-Padula style information flow approach [7].

Third, XFaaS skips steps (6)-(7) for regularly invoked functions through *cooperative JIT compilation*. While running a new version of a function's code, one server collects the profiling data needed for JIT compilation and pushes it to other servers, enabling them to pre-compile the function.

Finally, to improve the hit rate of the JIT code cache, which helps skip steps (6)-(7), XFaaS ensures that only calls for a subset of functions, rather than all functions, are dispatched to a given server. XFaaS dynamically adjusts this subset for each server in response to global load changes.

Overall, these techniques allow XFaaS to eliminate the overhead of steps (1)-(5) and (9)-(10) for all functions and further eliminate the overhead of steps (6)-(7) for functions that are regularly invoked.

Solution for high variance of load. To reduce hardware costs, we intentionally provision XFaaS with hardware that is insufficient for its peak demand and then leverage several techniques to manage overload situations. First, it employs time-shift computing to postpone the execution of certain functions. Each function has a criticality and a deadline, with the deadline ranging from seconds to 24 hours. When XFaaS reaches its capacity limit, delay-tolerant functions are deferred to off-peak hours for execution. If the capacity is still insufficient to run all functions with a short deadline, low-criticality functions are also deferred. Second, XFaaS globally dispatches function calls across datacenter regions to balance the load. Third, XFaaS defines a quota for each function and throttles functions that exceed their quota. Finally, it allows the caller to specify a future execution start time for

a function instead of executing it immediately, which helps spread out the load in a predictable manner.

Thanks to these optimizations, the curve of the "*Executed*" functions in Figure 2 is much less spiky compared to the curve of the "*Received*" function calls. Consequently, XFaaS only needs to provision sufficient capacity to match the "*Executed*" curve instead of the "*Received*" curve.

Solution for overloading downstream services. XFaaS addresses this problem through *adaptive concurrency control*. First, it leverages a global resource isolation and management (RIM) system to enforce resource isolation across distributed components. Instead of making decisions locally, RIM collects global metrics across different systems to assist XFaaS operate in real-time coordination with downstream services. Second, downstream services can send back-pressure signals to XFaaS. In response, XFaaS uses a TCP-like congestion-control mechanism to pace the execution of functions.

Contributions. We make the following contributions:

- This is the first work that uses production data to shed light on whether the ease of use of FaaS comes at the expense of extra hardware costs for the FaaS platform. With a set of optimizations, XFaaS achieves a daily average CPU utilization of 66%. Based on our anecdotal knowledge in the industry, this level of utilization might be several times higher than that of typical FaaS platforms, despite the much spikier load in XFaaS.
- We propose a holistic set of techniques that work together to closely approximate the effect of a *universal worker*, where every worker can instantly execute every function without any cold start overhead.
- We propose a holistic set of techniques to manage load spikes, including time-shift computing, dispatching function calls across datacenter regions, and prioritizing function execution based on criticality and quota. None of these have been applied to FaaS before.
- We propose *adaptive concurrency control* to prevent functions from overloading downstream services. This problem has not been studied in FaaS before.

2 Background

To set the stage for future discussions, we provide some background on FaaS in our private cloud.

2.1 Rapid Growth of FaaS in Our Private Cloud

As the popularity of FaaS has been growing rapidly in public clouds, one may wonder whether a private cloud has sufficient FaaS workloads to support similar growth. Figure 3 shows that the number of daily function invocations in XFaaS has increased 50 times over the past 5 years, currently totaling trillions of function calls daily. The rapid growth at the end of 2022 is due to the launch of a new feature that allows for the use of Kafka-like data streams [12] to trigger function calls. Overall, the rapid adoption of XFaaS shows that FaaS is a successful programming paradigm that is equally applicable to both public and private cloud environments. The hyperscale of XFaaS requires us to efficiently use hardware, especially in coping with spiky loads.

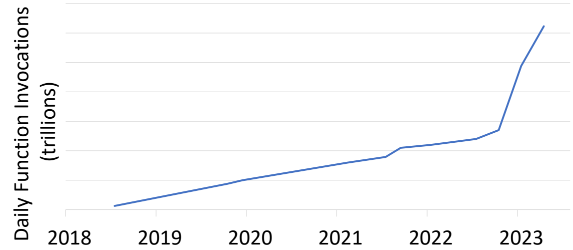

Figure 3. Growing popularity of FaaS in our private cloud.

2.2 Spiky Load

The clients of XFaaS submit function calls in a highly spiky manner, as depicted by the "*Received*" curve in Figure 2. The peak demand is 4.3 times higher than the off-peak demand. The midnight peak is caused by function calls triggered by Hive-like [24] big-data pipelines that create data tables around midnight. The availability of the data triggers the invocation of many functions at a high volume. As a specific example, Figure 4 illustrates the spiky load of a function with almost 20 million function calls submitted within a 15-minute time window. Allocating capacity to accommodate the peak demand indiscriminately would result in considerable hardware waste during off-peak times.

One of our key insights is that certain XFaaS functions may not need immediate execution. Most XFaaS functions are triggered by queue, timer, storage, orchestration and event bridge workflows. These functions tend to be more delay-tolerant than functions triggered through direct RPC like HTTP. Functions with relaxed latency expectations, present opportunities for smoothing out the load.

XFaaS adopts a holistic set of techniques to handle spiky loads. When necessary, it defers the execution of delay-tolerant functions and low-criticality functions. Moreover, it globally dispatches function calls across datacenter regions

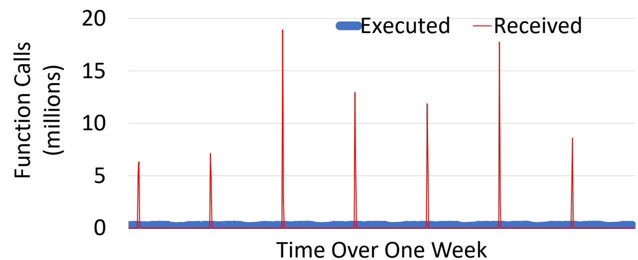

Figure 4. The load of a spiky function. This function allows its function calls to be executed with a 24-hour SLO. XFaaS leverages this property to spread out its function execution.

to balance the load. Additionally, it enforces a per-function quota to ensure fairness. Finally, it allows the caller to specify a future time for function execution, which spreads out the load in a predictable manner. In combination, these techniques drastically reduce spikes in function execution, as demonstrated by the "*Executed*" curves in Figures 2 and 4.

2.3 Datacenters and Uneven Hardware Distribution

Our private cloud comprises tens of datacenter regions and millions of machines. Each region comprises multiple datacenters that are physically close to one another. Due to the low-latency and high-bandwidth network connecting datacenters in the same region, we can largely consider hardware within a region to be fungible. However, the cross-region network bandwidth is about 10 times lower than the bandwidth between datacenters within a region, and the cross-region network latency is about 100-1000 times longer than the latency within a region. As a result, our services often treat communication within-region and cross-region differently. Therefore, XFaaS needs to strike a balance between regional locality and cross-region load balancing.

Figure 5 shows the capacity of XFaaS's worker pools in a subset of datacenter regions. Due to the incremental acquisition of capacity and availability of capacity [14], XFaaS's capacity is unevenly distributed across regions. This requires XFaaS to optimize for regional locality while also balancing the load across regions to drive hardware to high utilization.

2.4 Terminology

We will define some terminology to set the stage for future discussions. Each execution of a function is referred to as a *function call* or *function invocation*. A language *runtime* is an environment in which functions are executed. Currently, XFaaS supports runtimes for PHP, Python, Erlang, Haskell, and a generic container-based runtime for any language. A *worker* is a server, i.e., a physical machine that hosts a specific runtime to execute functions.

A *namespace* is a strongly isolated environment in XFaaS that consists of a pool of dedicated workers and a set of functions. A function belongs to a single namespace, and a worker also belongs to a single namespace. Each namespace

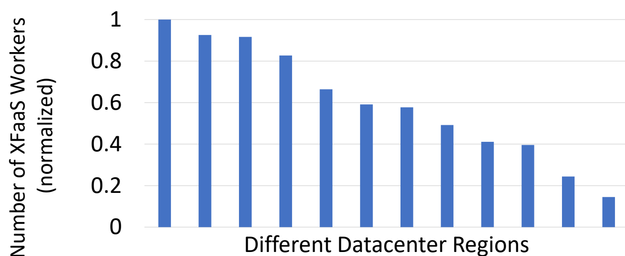

Figure 5. Uneven distribution of XFaaS's hardware capacity across datacenter regions.

supports only one runtime, and currently XFaaS has over 20 namespaces. A namespace is a multi-tenant environment that can run functions from multiple teams. The creation of a new namespace is a rare occasion that only happens when there is a strong need for security or performance isolation, or when a new runtime is introduced.

XFaaS allows one instance of a runtime (i.e., one Linux process) to concurrently execute multiple functions. To minimize interference, these functions are restricted to using only a subset of the language runtime features. For instance, forking a process is not allowed. The language runtime provides data isolation among functions by using a multilevel security-information flow approach that incorporates access control in a Bell-La-Padula style [7].

A function has several attributes that developers can set, such as function name, arguments, runtime, criticality, execution start time, execution completion deadline, resource quota, concurrency limit, and retry policy. The execution completion deadline can range from seconds to 24 hours. When XFaaS is low on capacity, functions that can tolerate delays are postponed until off-peak hours.

3 Diverse Workloads

XFaaS supports a range of diverse workloads. Its functions come from thousands of teams, supporting all major products, along with their corresponding diverse runtimes (PHP, Python, Erlang, Haskell, and C++). Additionally, it accommodates nearly every FaaS trigger available in public clouds, including queues, orchestration workflows, timers, storage, data streams, and event bridges. In this section, we summarize the characteristics of XFaaS workloads.

3.1 Workload Categories

We classify functions into three categories based on their triggers: (1) queue-triggered functions, which are submitted via a queue service; (2) event-triggered functions, which are activated by data-change events in our data warehouse and data-stream systems; and (3) timer-triggered functions, which automatically fire based on a pre-set timing.

Over one month, XFaaS executed 18,377 unique functions. The function count, invocation count, and compute usage of these functions are summarized in Table 1. In terms of the function count, queue-triggered functions dominate due to their longest history of usage. However, once XFaaS started to support event-triggered functions, numerous data-processing services began utilizing them, leading to rapid growth. These functions tend to run frequently but have shorter execution times. Consequently, event-triggered functions account for 85% of invocations but only 14% of compute usage. The support for timer-triggered functions is the latest addition to XFaaS, and is still gaining momentum.

Triggers	Functions	Function Calls	Compute Usage
Queue-triggered	89%	15%	86%
Event-triggered	8%	85%	14%
Timer-triggered	3%	<1%	<1%

Table 1. Breakdown of functions by categories.

3.2 Workload Examples

In general, XFaaS workloads seldom handle user-facing interactive requests that demand sub-second response times, such as newsfeed display, search result ranking, or video playback. Traditionally, at Meta, these interactive user requests are handled by long-running services, as serverless functions do not offer significant advantages in these scenarios.

Serverless functions are typically lauded for two primary benefits: (1) pay-as-you-go and no upfront capacity planning, and (2) streamlined deployment where developers only write code, and the serverless platform handles deployment automatically. However, for user-facing requests requiring sub-second response times, the first benefit loses relevance because meticulous capacity planning is needed to provide guaranteed capacity and ensure delightful experiences for the billions of users of Meta products. This is very different from the scenario of a small product with a limited user base, where occasional user experience degradation is acceptable, and cloud providers are expected to have spare capacity to accommodate load spikes of small products.

Moreover, at Meta, the second benefit can be achieved through full deployment automation without employing serverless functions. Notably, our continuous deployment tool, Conveyor [18], already deploys 97% of all services without any human intervention, and even serverless functions are deployed through Conveyor. Due to these reasons, at Meta, serverless functions are rarely used to handle user-facing requests requiring sub-second response times.

Next, we describe several examples of typical XFaaS workloads. To report resource usage, we use million instructions per second (MIPS) as the metric for CPU. For memory usage, we measure the peak memory consumption of each function invocation within one-minute intervals. The characteristics of these workloads are summarized in Table 2.

Workload	Trigger	Calls/s	% of call count	CPU (MIPS)	Memory (MB)	Execution Time (s)
Recommendation	Queue	35K	0.03%	3K - 4K	220 - 260	10 - 20
Falco	Data Stream	25M	22.4%	1 - 4	20 - 90	0.004 - 0.1
Productivity Bot	Queue	6K	0.005%	40 - 120	5 - 10	0.15 - 0.25
Notifications	Data Warehouse	3.4M	3%	65 - 200	10 - 90	0.55 - 1.1
Morphing Framework	Queue	25K	0.02%	1.5M - 27M	30 - 230	65 - 155

Table 2. Examples of XFaaS workloads. As each workload utilizes multiple functions, the last three columns report both the minimum and the maximum of CPU usage, memory usage, and execution time of these functions.

Recommendation System invokes functions to generate recommendations on certain user events. For example, accepting a friend request might trigger the execution of a function to generate a new set of friend recommendations. Although this functionality supports a user-facing feature, it does not require real-time response and will not block accepting a friend request as friend recommendation runs asynchronously.

Falco is a logging platform for all types of events, including those from frontend, backend, and mobile clients. When the logging server receives a request to log an event (e.g., when an A/B test parameter is consumed by an app on a mobile device), it writes the event to a data stream, which triggers the execution of a function to process the logged event. Although log processing is not on the critical path of handling user interactions, its latency still matters because the logged events may trigger downstream processing that subsequently affects user experiences. For instance, the logged data may be used promptly for online ML training to provide users with better recommendations immediately. Therefore, Falco functions are subject to the SLO of execution within 15 seconds on average and within one minute at the 99th percentile.

Productivity Bot is a platform for automating various tasks based on rules. For example, one can create a rule to send a message when a code change is deployed into production. Internally, the productivity bot leverages XFaaS functions that are triggered by events like code deployment to execute various automations.

Notification System schedules notification campaigns via communication channels such as SMS, email, and push notifications. Following a per-product configuration, a scheduling system selects target users from data warehouse at preset times and activates XFaaS functions to send notifications.

Morphing Framework is a platform that programmatically generates ephemeral functions for executing custom data transformation across data stores (e.g. MySQL, key-value stores). These functions run for minutes and consume orders of magnitude more CPU cycles than ordinary functions.

3.3 Workload Resource Usage

Across all functions supported by XFaaS, their resource consumption varies widely, as shown in Table 3. CPU usage per function call varies from 0.37 MIPS at P10 to 1,064,280 MIPS at P99. Overall, 31% of functions have per-invocation CPU usage below 1 MIPS, 65% below 10 MIPS, and 89% below 100 MIPS. In terms of per-invocation memory usage, 60% of functions use below 16 MB, 92% use below 256 MB, while 2% exceed 1 GB. Lastly, the execution time varies from milliseconds to over 10 minutes. Specifically, 33% of function calls finish within 1 second, 94% finish within 60 seconds, while 1% exceed 5 minutes.

Function Type	CPU Usage (MIPS)					Memory Usage (MB)					Execution Time (ms)				
	P10	**P50**	**P90**	**P95**	**P99**	**P10**	**P50**	**P90**	**P95**	**P99**	**P10**	**P50**	**P90**	**P95**	**P99**
Queue-triggered	20.40	221.80	7,611	31,421	1,064,280	0.08	2.16	63.8	194.6	5,952	219.20	2,652	23,060	76,711	266,446
Event-triggered	0.54	11.36	189	441	2,981	0.02	1.20	26.0	44.7	220	3.96	97.20	1,270	2,817	11,232
Timer-triggered	0.37	576.00	44,839	144,020	369,282	0.00	0.17	66.9	134.5	388	23.74	3,625	96,352	150,998	659,162

Table 3. Percentiles of CPU usage, memory usage, and execution time of all functions. P10 means the 10th percentile.

Queue-triggered functions have a longer tail with high CPU usage, as some of them (e.g., Morphing Framework functions) are long-running and execute complex data transformations. Timer-triggered functions exhibit high variation, with an execution time of 24 ms at P10 and almost 11 minutes at P99. Lastly, event-triggered functions execute at a high frequency but with low CPU usage. These functions are triggered by data changes in our data warehouse and data-stream system. The Notification System and the Falco logging system fall under this category.

In summary, serverless functions demonstrate significant variability in their resource consumption. This necessitates XFaaS to employ intelligent scheduling and load balancing techniques to optimize hardware utilization.

4 XFaaS Design

We follow the architecture diagram in Figure 6 to present the design of XFaaS. First, we provide an overview of how a function call is processed end-to-end, and then we elaborate on the details of each component.

4.1 Overview

To initiate a function call, a client sends a request to a *submitter*. The submitter enforces rate limiting and forwards the request to a *QueueLB* (queue load balancer), which then selects a *DurableQ* (durable queue) to persist the function call until it completes. The *scheduler* periodically pulls function calls from the DurableQs and stores them in its in-memory *FuncBuffer* (function buffer), with a separate buffer designated for each function. The scheduler determines the execution order of function calls based on their criticality, completion deadline, and quota, and transfers function calls that are ready for execution to the *RunQ* (run queue). Finally, the function calls in the RunQ are dispatched to the *WorkerLB* (worker load balancer), which routes them to the appropriate workers for execution.

In Figure 6, DurableQ is the only sharded and stateful component, unlike the other components which are stateless and unsharded. DurableQ uses a sharded database that is highly available to store function calls permanently. The submitter, QueueLB, scheduler, and WorkerLB are all stateless, unsharded, and replicated without a designated leader, so their replicas play equal roles. This architecture concentrates state management in a single component, simplifying the overall design. Scaling a stateless component can be easily achieved by adding more replicas, and if a stateless component fails, any peer can take over its work.

With XFaaS's hyperscale, each component in Figure 6 typically runs on hundreds or more servers, except for the workers and DurableQs. These components serve as the compute and storage engines for function calls, and their capacity needs are proportional to the volume of function calls. Currently, the workers and DurableQs consume over 100,000 and 10,000 servers, respectively.

XFaaS operates across multiple geo-distributed datacenter regions, with the capability to dispatch function calls to any region. However, it aims to strike a balance between load balancing and regional locality, by executing most function calls in the same region where they are submitted. As shown in Figure 5, XFaaS's hardware capacity varies significantly across regions, and hence load balancing is critical. To achieve this, three components work together. First, QueueLBs balances the load across DurableQs in different regions. Second, schedulers distribute function calls proportionally to each region's worker pool capacity. Finally, WorkerLB balances the load of individual workers while also ensuring that each worker handles a stable and subset of functions for improved locality, rather than all functions.

To ensure fault tolerance, the central controllers at the top of Figure 6 remain separate from the critical path of function execution. The controllers continuously optimize the system by periodically updating key configuration parameters, which are consumed by critical-path components like workers and schedulers. Since these configurations are cached by the critical-path components, they continue to execute functions using the current configurations even if the central controllers fail. However, when the central controllers are down, XFaaS would not be reconfigured in response to workload changes. For example, the traffic matrix for cross-region function dispatching won't be updated even if the actual traffic has shifted. Typically, this does not lead to significant imbalance immediately and can withstand central controller downtime for tens of minutes.

For simplicity, Figure 6 shows only one work pool per region for one *namespace*. In reality, a region may host multiple namespaces and each component depicted in Figure 6 can support multiple namespaces simultaneously, except that each namespace has its own dedicated worker pools. Below, we describe each XFaaS component in detail.

4.2 Submitter

A function can be executed in response to various events. The first type of event involves clients submitting function calls to submitters, as illustrated in Figure 6. The second type

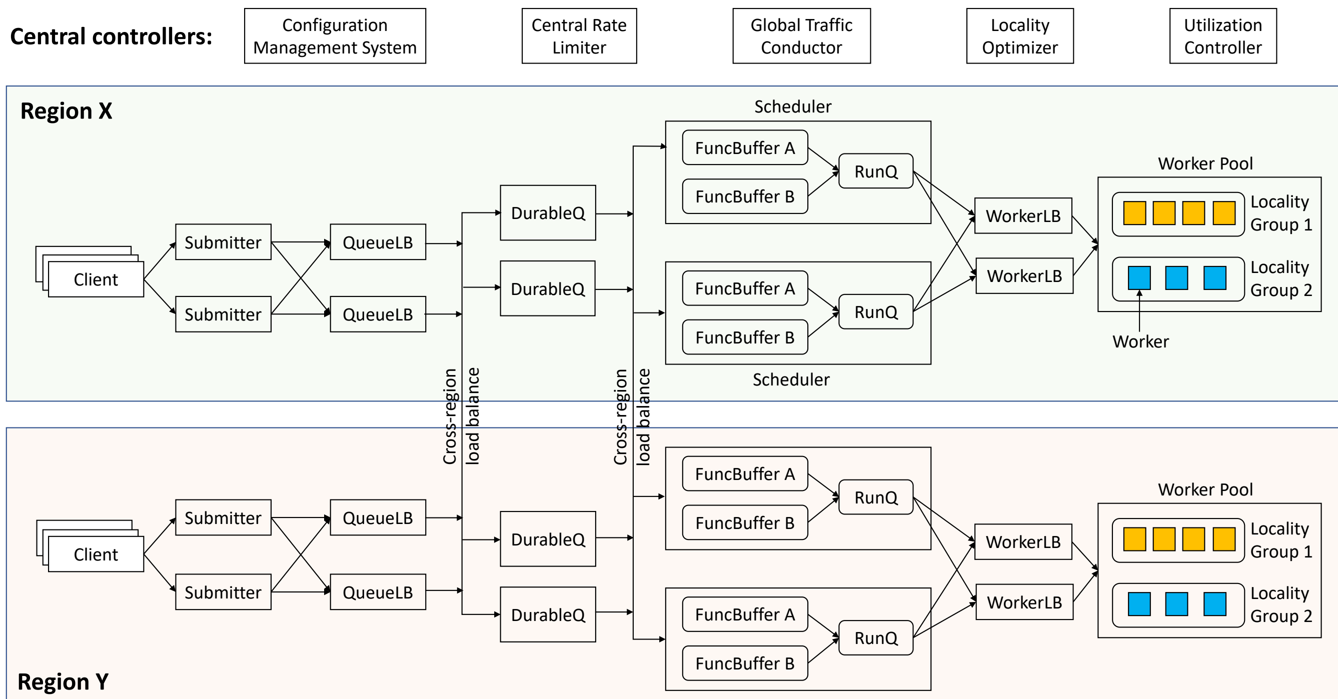

Figure 6. High-level architecture of XFaaS.

of event involves data changes in our data warehouse [42], while the third type of event is triggered by the arrival of new data in our data-stream system [12]. For simplicity, the last two types of events are not shown in Figure 6.

Initially, XFaaS allowed clients to directly write into DurableQs the IDs and arguments of the functions to be called. As the rate of function calls increased, we introduced the submitter to improve efficiency by batching calls and writing them to a DurableQ as one operation. If a function's arguments are too large, the submitter stores the arguments separately in a distributed key-value store. Moreover, the submitter enforces policies like rate limiting to avoid overloading the submitter and the downstream components. To achieve this, each submitter independently consults the *Central Rate Limiter* shown in Figure 6 to keep track of the global resource usage.

Finally, because some clients have very spiky submission rates of function calls, as shown in Figures 2 and 4, each region has two sets of submitters, one for normal clients and another for very spiky clients (such as the one in Figure 4) to prevent spiky clients from overly impacting normal clients. XFaaS monitors extremely spiky clients and alerts its operators to negotiate with the customer about moving them to the spiky submitters; otherwise, XFaaS will throttle them by default. Note that this needs human involvement because it is an explicit SLO change for the customer.

4.3 DurableQ and QueueLB

Upon receiving a function call, the submitter forwards it to a QueueLB. The Configuration Management System depicted at the top of Figure 6 is called Configerator [40]. It stores and delivers a routing policy to each QueueLB, which specifies the distribution of the submitter-to-QueueLB traffic for each ¡source-region, destination-region¿ pair. This helps balance the load across DurableQs in different regions as the hardware capacity of DurableQs also varies wildly across regions, similar to that in Figure 5. The mapping of function calls to DurableQs is sharded by a random UUID to distribute the load evenly across DurableQs. This means that a function call can be queued at any DurableQ.

Each DurableQ maintains a separate queue for each function, ordered by the function call's *execution start time*, which is specified by the caller and can be a future time, such as eight hours from now. The scheduler periodically queries DurableQs to retrieve function calls whose start time is past the present time. Once a DurableQ offers a function call to a scheduler, it will not offer it to another scheduler unless the scheduler fails to execute it. After a function call is executed by a worker, the scheduler notifies the DurableQ with either an *ACK* message to indicate that the function was executed successfully or a *NACK* message to indicate otherwise. Upon receiving an *ACK*, the DurableQ permanently removes the function call from its queue. If it receives a *NACK* or neither an *ACK* nor a *NACK* after a timeout, it makes the function

call available for another scheduler to retrieve and retry. This means that a DurableQ offers the at-least-once semantics.

4.4 Scheduler

A scheduler's main responsibility is to determine the order of function calls based on their criticality, execution deadline, and capacity quota. As depicted in Figure 6, the inputs to the scheduler are multiple *FuncBuffers* (function buffers), one for each function, and the output is a single ordered *RunQ* (run queue) of function calls that will be dispatched for execution. FuncBuffers and RunQ are in-memory data structures.

Each scheduler periodically polls different DurableQs to retrieve pending function calls. Since the mapping of function calls to DurableQs is sharded by a random UUID, the scheduler may retrieve different callers' invocations for the same function from different DurableQs. Those invocations are merged into the single FuncBuffer for the function, which is ordered first by the criticality level of function calls and then by their execution deadline. As XFaaS often runs under constrained capacity, prioritizing criticality first ensures that important function calls are more likely to be executed during a capacity crunch or a site outage.

The scheduler selects the most suitable function call among the top items of all FuncBuffers and moves it to the RunQ for execution. The selection criteria involve quota management and will be described in detail in §4.6. The RunQ also serves the purpose of flow control. If the RunQ builds up due to slow function execution, the scheduler will slow down both the movement of items from FuncBuffers to the RunQ and the retrieval of function calls from DurableQs.

When workers in a region are underutilized, to balance the load, the region's schedulers may retrieve function calls from DurableQs in other regions whose workers are overloaded. The *Global Traffic Conductor* (GTC) in Figure 6 maintains a near-real-time global view of the demand (pending function calls) and supply (capacity of worker pools) across all regions. It periodically computes a traffic matrix where its element T_{ij} specifies the fraction of function calls that the schedulers in region i should pull from region j. To compute the traffic matrix, the GTC starts by setting $\forall i, T_{ii} = 1$ and $\forall i \neq j, T_{ij} = 0$, meaning that all schedulers will only pull from DurableQs in their local region. However, this might lead to the workers in certain regions becoming overloaded. The GTC calculates the shift of traffic in those overloaded regions to their nearby regions until no region is overloaded or all regions are equally loaded. The GTC periodically distributes a new traffic matrix to all schedulers in all regions via the *Configuration Management System* in Figure 6. The schedulers then follow the traffic matrix to retrieve function calls from DurableQs in different regions.

4.5 Workers and WorkerLB

Each namespace supports a single runtime, and has its dedicated worker pool. Functions that use the same programming language but require strong isolation are separated into different namespaces. For the sake of brevity, the discussion below assumes a single namespace, meaning a single runtime and a single worker pool.

To maximize hardware efficiency, we want to closely approximate the effect of a *universal worker*, where every worker can instantly execute every function without any startup overhead. XFaaS achieves this through multiple techniques. First, it allows multiple functions to execute concurrently in one instance of the runtime, i.e., one Linux process. Second, it uses an efficient peer-to-peer system [15] to proactively push the latest code of all functions in a namespace to the local SSD disk of every worker in that namespace. The workers keep the runtime up and running all the time. Upon receiving a call for a function for the first time, the runtime loads the pre-populated function code from its local SSD and executes it immediately. One instance of the runtime can concurrently execute different functions in different threads. Third, XFaaS uses *cooperative JIT* compilation among workers to eliminate the overhead and delay of every worker redoing profiling for JIT compilation (§4.5.1). Finally, to improve the cache hit rate of the function code and JIT code in a worker's memory, XFaaS uses *locality groups* to ensure that only calls for a stable and small subset of functions, rather than all functions, are dispatched to a given worker (§4.5.2).

4.5.1 Cooperative JIT Compilation. We use PHP as the primary example in our discussion. Our PHP runtime is called HHVM [23], which uses instrumentation-based profiling to enable region-based JIT compilation [36]. However, it is inefficient to have tens of thousands of workers each independently performing profiling, as previous work [37] shows that HHVM needs up to 25 minutes to finish profiling and produce high-quality JIT code for a code size of around 500MB. This is problematic for XFaaS because it frequently pushes new code for functions to all workers.

To solve this problem, XFaaS adopts cooperative JIT compilation among workers. Every three hours, XFaaS bundles all new and changed function code into a file and pushes it to all workers through peer-to-peer data distribution [15]. Workers start using the new function code in three phases. In the first phase, a small set of workers run the new code to catch potential bugs. In the second phase, 2% of the workers run the new code to catch harder-to-detect bugs, and some seeder workers perform profiling to collect the data needed for JIT compilation. In the third phase, a seeder's profiling data is distributed to all workers in the same *locality group* as the seeder, allowing them to perform JIT compilation for hot functions immediately, even before they receive function calls for the new code. Later, when they receive those calls, they can immediately execute the optimized JIT code without any startup or profiling delay.

4.5.2 Locality Groups.

Our goal is to closely approximate the effect of an *universal worker*, where every worker can instantly execute every function without any startup overhead. However, due to the limited memory capacity, it is infeasible to keep every function's JIT code in every worker's memory. Moreover, functions themselves need memory to cache data and perform computations. If a worker happens to execute multiple memory-hungry functions concurrently, it may run out of memory. For example, functions of the Morphing Framework described in Section 3.2 are long–running and consume increasingly more memory until they finish.

To address this problem, the *Locality Optimizer* depicted in Figure 6 partitions both functions and workers into locality groups to ensure that only calls for a subset of functions are dispatched to a given worker. Based on the profiling data of functions, the Locality Optimizer partitions functions into non-overlapping *locality groups* and ensures that memory-hungry functions are spread out into different locality groups. Specifically for the Morphing Framework, since it programmatically generates many ephemeral functions and those functions share similar characteristics, the Locality Optimizer simply assign them to different locality groups in a round-robin fashion.

In addition to mapping functions to locality groups, each locality group of functions is mapped to a corresponding locality group of workers, such as Locality Groups 1 and 2 in Figure 6. When a WorkerLB in Figure 6 routes a function call, it randomly chooses two workers from the function's corresponding worker locality group, and dispatches the function call to the worker with less load. This approach introduces locality into the traditional load-balancing approach of the power of two random choices [32].

As the resource-consumption profiles of functions change, the Locality Optimizer can dynamically reassign functions across locality groups. Moreover, if the mix of function calls changes, such as one locality group experiencing a surge in its function calls, the Locality Optimizer can move workers from one locality group to another to balance the load.

4.6 Handling Load Spikes

XFaaS uses a set of techniques to smooth out load spikes, which allows it to operate efficiently without over-provisioning resources to accommodate the peak load.

4.6.1 Quota for Functions.

Every function is associated with a quota set up by its owner, which defines the total number of CPU cycles it can use per second globally. This quota is transformed into a requests-per-second (RPS) rate limit by dividing the quota by the function's average cost per invocation. The RPS for a function is aggregated globally, and each scheduler consults the *Central Rate Limiter* in Figure 6 to determine whether to throttle the invocations to a function, based on whether the function globally exceeds its RPS limit.

4.6.2 Time-Shifting Computing.

XFaaS provides two types of quotas: *reserved* and *opportunistic*. If a function utilizes reserved quota and is currently within its allotted limit, XFaaS strives to initiate the execution of its function call on a worker within seconds of receiving the call. This is part of XFaaS's SLO. If a function uses opportunistic quota, XFaaS's execution SLO for that function is set to 24 hours. This enables XFaaS to schedule execution of these delay-tolerant functions during off-peak hours when capacity is available.

When workers are underutilized or overloaded, XFaaS dynamically adjusts the rate of invocations for functions using opportunistic quota to run at a rate above or below the RPS limit derived from their quota. Let r_0 denote an opportunistic function's preset RPS limit. Its real RPS limit is dynamically adjusted to $r = r_0 \times S$, where S reflects the current utilization of workers. If workers are underutilized, S increases. Conversely, if workers are overloaded, S can decrease all the way down to zero, causing the scheduling of opportunistic functions to stop. The *Utilization Controller* in Figure 6 monitors the utilization of workers, dynamically adjusts S to reach a target utilization level of workers, and stores S in a database. The schedulers periodically retrieves S from the database, and uses it to compute the adjusted RPS limit for opportunistic functions. Meta's hardware budget allocation process incentivizes teams to utilize opportunistic quota whenever possible, similar to public cloud's pricing incentives for using harvest VMs.

4.6.3 Protecting Downstream Services.

Even if XFaaS can perfectly manage its own load, its functions can still cause resource contention at the downstream services that they invoke. The following production outage initially motivated us to develop solutions to address this issue. At one time, the social-graph database called TAO [9] experienced high load during its peak hours, which is expected. However, additional load from a high-volume function running on XFaaS unexpectedly exacerbated the situation, overloading TAO and causing excessive failures and retries. These failures and retries amplified queries to several downstream services, resulting in degraded availability of TAO and further overloading an indexing service, subsequently causing a domino effect. The investigation to identify the root cause of this domino effect and the subsequent manual mitigation took place over the course of a full day, eventually leading to a large portion of function executions on XFaaS being paused manually to temporarily mitigate the situation.

To avoid overloading downstream services, XFaaS takes a TCP-like additive-increase-multiplicative-decrease (AIMD) approach to dynamically adjust the RPS limit for functions. A downstream service can throw a *back-pressure* exception to indicate that it is overloaded. When the back-pressure for a function exceeds a threshold, its RPS limit r is adjusted to a fraction M of its current RPS limit, $r_{k+1} = r_k \times M$. When it is free of back pressure, the RPS limit is increased additively,

$r_{k+1} = r_k + I$. Both M and I are tunable parameters. The back-pressure threshold is defined per downstream service and is set by the service owner based on their domain knowledge. For example, for two of our largest downstream services, the threshold was set at 5,000 exceptions per minute, which has worked robustly without any changes for the past two years.

XFaaS also provides other traffic shaping features to assist downstream services. Let r denote a function's RPS limit and p denote a function's execution time. The function may have up to $R = r \times p$ running instances at any given time. If p is large, R can also be large, which may result in too many concurrent calls to a downstream service, possibly causing it to become overloaded. XFaaS allows each function to be configured with a *concurrency limit* to indicate the maximum number of instances that the function can run at any given time. The concurrency limit is less powerful than the AIMD approach described above, but serves as a proper safety net, since not every downstream service is well-implemented to throw back-pressure exceptions in overload situations.

While some downstream services may be able to handle high RPS in a steady state, abrupt changes such as a sudden increase of RPS from $\frac{r}{10}$ to $\frac{r}{2}$ may not be well-handled by such services. These services may need time to warm up their cache or use autoscaling to add more instances in response to a load increase. To assist these services, XFaaS uses *slow start* to gradually increase a function's RPS while imposing a limit on the rate of change to RPS. Specifically, if the number of function calls per time window W is already above a threshold of T, traffic will increase by a maximum factor of α per time window. Through empirical evaluation, we have set these parameters as $W = 1$ minute, $T = 100$ function calls, and $\alpha = 20\%$. Slow start slowly but steadily increases RPS until either the RPS limit or concurrency limit is reached or the function's finish rate is unable to keep up with the rate of scheduled function calls.

4.7 Data Isolation

Functions that require strong isolation for security or performance are assigned to different namespaces, each using distinct worker pools to achieve physical isolation. Within the same namespace, multiple functions can run in the same Linux process, owing to the high degree of trust within a private cloud, the mandatory peer review of all code changes, and the default security mechanisms that are already in place. In addition, we enforce data isolation across functions to prevent unintended data misuse. This is accomplished using a multilevel security-information flow approach that incorporates access control in a Bell-La-Padula style [7].

XFaaS offers a programming model where function owners can annotate the arguments of their function with semantics. A system automatically infers data type semantics and offer warnings on missing or potentially inaccurate annotations. To ensure data isolation across functions, XFaaS enforces the Bell-La-Padula security principles. Principals at higher classification levels do not write to lower classification levels, and principals at lower classification levels do not read from higher classification levels. In other words, data can only flow from lower to higher classification levels.

XFaaS enforces the policy at the boundaries between systems which are separated into *isolation zones* with different classification levels. Each function call is labelled with an isolation zone, and the labeling can be done either statically at the function coding time, or dynamically through propagation during RPC calls. The XFaaS scheduler checks if a function's arguments from a source isolation zone can flow to the function's execution isolation zone, respecting the Bell-La Padula properties. Similarly, workers also ensure that a function running in a zone follows these properties.

Overall, XFaaS's novel use of a combination of isolation models enables it to operate securely and efficiently. On one hand, it uses separate worker pools to execute functions that require strong isolation. On the other hand, functions within the same namespace rely on the language runtime to maintain data isolation at function granularity. This allows XFaaS to run multiple functions concurrently within a single Linux process, maximizing efficiency.

5 Evaluation and Experience

We use production data to evaluate XFaaS. Our evaluation-focuses on the following questions.

- Can XFaaS drive workers to high utilization? (§5.1)
- Can XFaaS closely approximate the effect of a *universal worker* so that every worker can execute every function instantly without any startup overhead? (§5.2)
- Can XFaaS effectively defer delay-tolerant functions to off-peak hours for execution? (§5.3)
- Can cooperative JIT compilation help workers achieve their maximum performance quickly? (§5.4)
- Can XFaaS automatically prevent functions from overloading downstream services? (§5.5)

5.1 CPU Utilization of Workers

Due to the hyperscale of XFaaS, reducing hardware costs is a top priority for us. The primary hardware cost is the worker pool, and improving its utilization effectively reduces hardware waste. Figure 7 shows the average utilization of workers in 12 different regions. We make several observations. First, despite the very spiky load in received function calls as depicted in Figures 2 and 4, XFaaS is effective in spreading out the actual execution of function calls evenly. The peak-to-trough ratio of CPU utilization is only 1.4x, which is a significant improvement over the peak-to-trough ratio of 4.3x depicted in Figure 2 for the "*Received*" curve. Second, overall, XFaaS achieves a high daily average CPU utilization of 66%, showing that it is effective in eliminating unnecessary wait time in a function's lifecycle as shown in Figure 1. Based on our anecdotal knowledge in the industry, this level

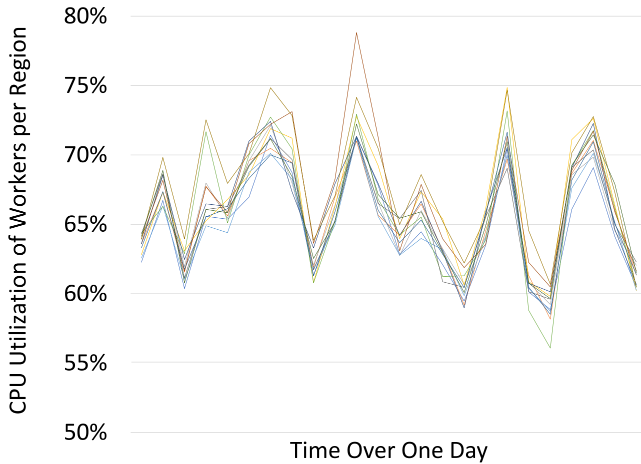

Figure 7. CPU utilization of workers in different regions.

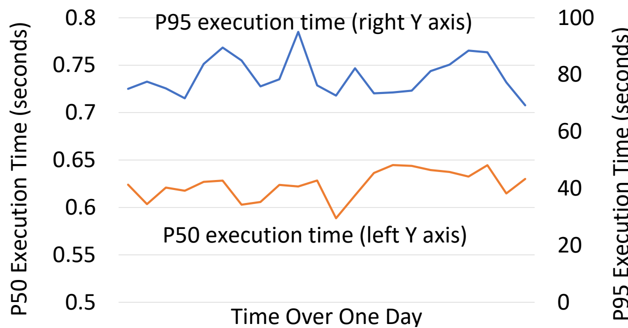

Figure 8. Wall-clock time of function execution.

of CPU utilization might be several times higher than that of typical FaaS platforms. Third, the CPU utilization levels of different regions stay within a narrow band, indicating that XFaaS is effective in balancing load across regions despite the uneven distribution of hardware capacity in different regions, as depicted in Figure 5.

Figure 8 shows that the majority of functions are lightweight and execute within less than one second. Therefore, it would be very wasteful to incur the overhead depicted in Figure 1 for each function call. Although the execution time of most functions is short, some functions can have a very long execution time, as the P95 execution time hovers around 80 seconds. Despite the complexity introduced by the high variance of execution time, XFaaS is effective in managing the load and achieving a high CPU utilization.

5.2 Locality Group and Memory Consumption

To maximize hardware efficiency, our goal is to closely approximate the effect of a universal worker, where every worker can instantly execute every function without any startup overhead. However, due to limited memory capacity, it is infeasible to keep the JIT code for every function in every worker's memory. To address this problem, as described in §4.5.2, XFaaS partitions both functions and workers into

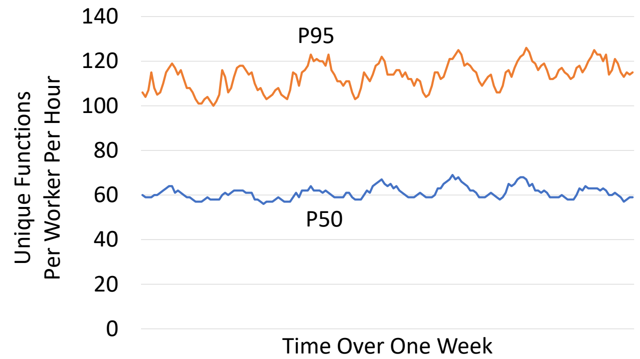

Figure 9. Unique functions executed by a worker per hour.

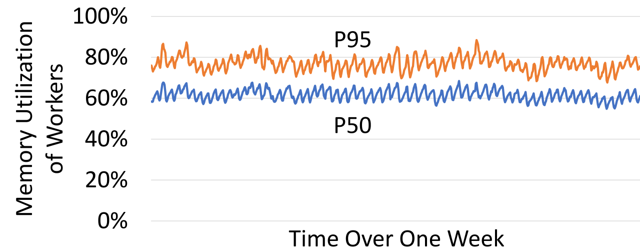

Figure 10. Memory utilization of workers.

locality groups to ensure that only calls for a subset of functions are dispatched to a given worker.

To evaluate the impact of locality groups, we conducted an experiment in production, by dividing all workers in a specific region into two partitions, with and without locality groups, respectively. Production traffic in the region was randomly distributed to the two partitions to ensure a fair comparison. Over a two-week-long period, workers in the partition with locality group on average consumed 11.8% and 11.4% less memory at $P50$ and $P95$, respectively. This experiment demonstrates that locality groups are effective in reducing workers' memory consumption.

Moreover, Figure 9 shows that, although there are tens of thousands of functions, each worker executes only about 61 and 113 distinct functions within a one-hour window at P50 and P95, respectively. Furthermore, Figure 10 demonstrates that a worker's memory consumption stays at a stable level while being highly utilized.

These results demonstrate that locality groups are able to effectively manage memory pressure and approximate the effect of a universal worker. Moreover, for power efficiency, all our workers are configured with only 64GB of memory. This demonstrates that a universal worker can be realistically approximated with a moderate amount of memory.

5.3 Time-Shifting Computing

As described in §4.6.2, XFaaS provides two types of quotas: *reserved* and *opportunistic*. Function using opportunistic

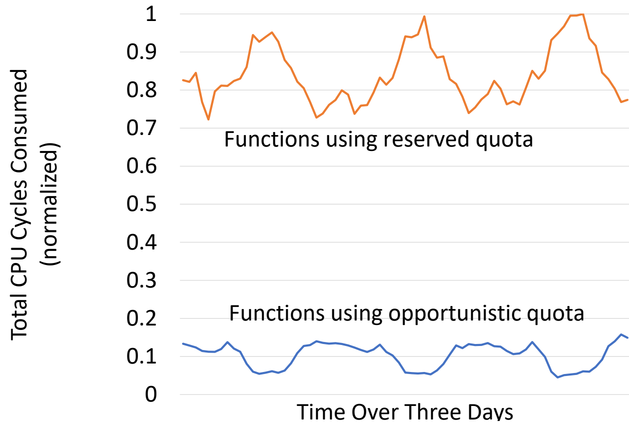

Figure 11. Total CPU cycles consumed by functions using *reserved* and *opportunistic* quotas, respectively.

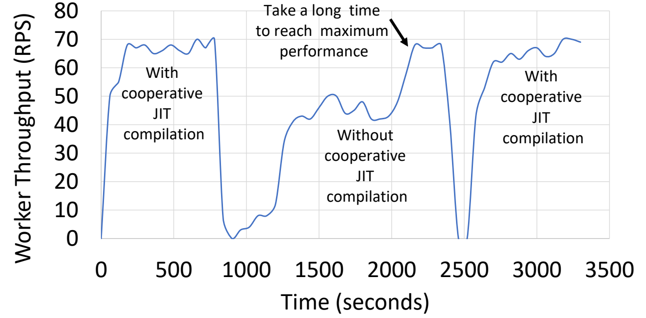

Figure 12. Restarting a worker's runtime with and without cooperative JIT compilation. Without cooperative JIT compilation, it takes 21 minutes (between time 900 and 2160 seconds) for the worker to reach its maximum performance.

quota can be deferred to off-peak hours for execution. Figure 11 compares the total CPU instructions consumed by functions using reserved quota versus those using opportunistic quota. We make several observations. First, functions using reserved quota demonstrate a diurnal pattern as most functions are triggered by events that are ultimately related to our user-facing products. Second, the CPU consumption of the reserved functions and opportunistic functions almost exactly complement each other, demonstrating that XFaaS is effective in scheduling opportunistic functions to run during off-peak hours. We are working aggressively to convert many functions that currently use reserved quota to use opportunistic quota. This is feasible because most functions do not actually have a tight deadline. Using opportunistic quota would allow XFaaS to further reduce its peak capacity needs, as well as run these functions with low-cost elastic capacity, which is similar to AWS' Spot Instances.

One may notice that the peak of the "*Received*" curve in Figure 2 is much higher than the opportunistic-quota curve in Figure 11. This is because multiple factors work together to smooth out the peak load: 1) the caller of a function can specify a future execution start time for the function; 2) XFaaS throttles functions that exceed their quota; 3) XFaaS can delay the execution of lower-criticality functions as needed; and 4) opportunistic functions are deferred to off-peak hours. Out of these factors, the opportunistic-quota curve in Figure 11 only reflects the last one. Although we expect that opportunistic quota will make up a larger percentage of the overall contribution to managing load spikes as we transfer more functions from reserved quota to opportunistic quota, the difference between Figures 2 and 11 emphasizes the importance of using a holistic set of techniques to manage load spikes, not just opportunistic quota alone.

5.4 Cooperative JIT Compilation

As explained in §4.5.1, our PHP runtime uses cooperative JIT compilation to address the inefficiency of workers taking tens of minutes to reach their maximum performance after a code update for functions. To compare the performance difference with and without cooperative JIT compilation, we perform an experiment on a worker running in production. We use T_K to denote the time at K seconds into the experiment. In Figure 12, at T0, we stop the worker's runtime to update the function code and restarts it with the JIT profiling data supplied by a seeder. With the assistance of the JIT data, the worker reaches its maximum RPS by T180, and the entire process takes three minutes. At T900, we restart the runtime without providing it with the JIT profiling data. As a result, the runtime had to perform its own instrumentation-based profiling, and it takes until T2160 for the worker to reach its maximum RPS. The entire process takes 21 minutes, which is much longer than the three minutes required to reach maximum RPS with cooperative JIT compilation. This experiment highlights the importance of cooperative JIT compilation, especially given XFaaS's practice of frequently pushing code updates for functions to all workers.

5.5 Protecting Downstream Services

We demonstrate XFaaS's effectiveness in preventing functions from overloading downstream services through two real-world incidents that happened in production.

In the first incident, a write-through cache (WTCache) situated in front of the social-graph database, called TAO [9], exhibited a significant degradation in read and write availability, dropping 10% and 20% from their respective SLOs. This was due to a bug in the new code release of WTCache that caused high traffic to be sent to WTCache's backing persistent key-value store (KVStore) during cache misses. As a result, KVStore throttled WTCache's requests, and WTCache, in turn, further dropped reads and writes to WTCache.

Under normal circumstances, XFaaS would continuously execute a high volume of calls to *Function A* and *Function B* in Figure 13, which would further call WTCache. However, during the incident, XFaaS's back-pressure mechanism slowed

Figure 13. When a downstream service was overloaded, XFaaS first slowed down function executions and then gradually increased them after the downstream service recovered. The top figure shows the downstream service's read SLO, and the bottom figures show how XFaaS paced the execution of two functions affecting the downstream service. The red vertical lines indicate the beginning and end of the incident.

down the execution of those functions to avoid further overloading WTCache when it was already unhealthy. XFaaS's DurableQs stored a backlog of pending calls to those functions without dropping those calls. As soon as WTCache recovered, XFaaS slowly increased the execution rates of those functions, and the backlog was processed within two hours. This incident demonstrates that XFaaS's back-pressure mechanism is effective in slowing down the execution of functions when the downstream service is overloaded.

The second incident is different from the first incident described above, but it also involved TAO. During the migration of TAO, its capacity was undersized in one of our datacenter regions, which led to TAO becoming overloaded after the migration. During this incident, XFaaS automatically slowed down the execution of as many as 200 unique functions, reducing the total traffic to TAO in the impacted region by about 40%. This successfully limited user impact as a potentially more severe region overload would had to be followed by a region drain and redirection of the traffic to remote regions, significantly increasing latencies or in the worst-case leading to unavailability. Overall, this incident, as depicted in Figure 14, demonstrates that XFaaS's back-pressure mechanism can automatically identify and slow down a massive number of unique functions that affect a downstream service without manual intervention. This would be hard to achieve without a fully automated mechanism.

6 XFaaS and Public Cloud

While certain techniques used in our private cloud may not translate directly to public clouds, this section discusses the broader lessons that might be applicable to public clouds.

In public cloud FaaS platforms, function executions are typically confined to a datacenter region. However, XFaaS

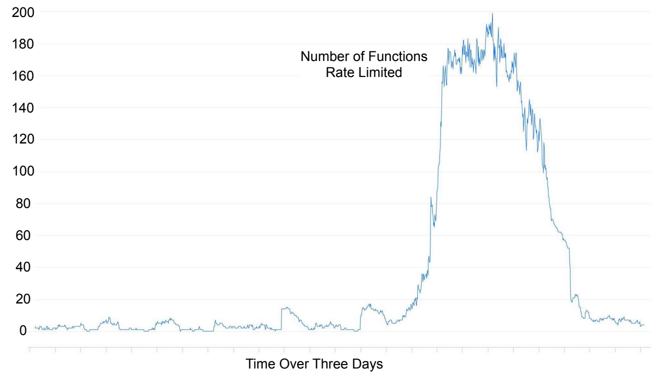

Figure 14. Number of unique functions (not invocations of the same function) whose execution was slowed down by XFaaS when a downstream service was overloaded.

takes a different approach, prioritizing local region execution but having the flexibility to dispatch function calls globally across regions when necessary for load balancing. While this strategy involves added complexity for global coordination, our successful implementation in XFaaS demonstrates its practicality. This approach holds potential relevance for public clouds, particularly as major cloud providers expand to encompass 50 or more regions, offering abundant opportunities for improved load balancing across regions.

Another main insight is that optimizing for resource utilization and the throughput of function calls should be an important focus of a FaaS platform, rather than solely fixating on the latency of function calls. We believe that this principle holds relevance for public clouds as well, although many existing studies tend to focus exclusively on reducing function cold start times, often overlooking considerations related to hardware utilization and throughput.

To reduce latency, a common approach is to keep a VM idle for 10 minutes or longer after a function invocation to allow for potential reuse [45]. In contrast, if a FaaS platform is optimized for hardware utilization and throughput, this waiting time should be reduced by a factor of 10 or more, because starting a VM consumes significantly fewer resources than having a VM idle for 10 minutes.

At Meta, function calls exhibit high levels of spikiness, and the peak-to-trough ratio for function calls is 4.3. Similarly, in the industry, Shahrad et al. reported that Azure Functions' workloads have a peak-to-trough ratio of two [39], and Wang et al. reported that Alibaba Cloud Function Compute experiences a peak-to-trough load ratio of more than 500x for certain functions [43]. If a FaaS platform primarily prioritizes latency, it would have to significantly over-provision hardware resources to meet the peak demand, leading to overall low hardware utilization.

Furthermore, Shahrad et al. reported that 64.1% of calls to Azure Functions are triggered by queues, events, storage, timers, and orchestration workflows [39]. While certain portions of these function calls are indirectly user-triggered [19]

and may have response expectations of a few seconds, the remaining calls possess the potential for delay-tolerant execution. This is because, in general, function calls triggered by non-RPC events like storage changes are less likely to involve an interactive user waiting for an immediate response, unlike functions triggered directly by RPCs like HTTP.

Several techniques in XFaaS for smoothing out the load spike might be applicable to public clouds. First, allowing the caller to specify a future execution start time would provide the caller with the flexibility to explicitly control how to spread out their function execution. Second, allowing a function's owner to choose its SLO in terms of the execution completion deadline would provide the FaaS platform the flexibility to postpone the execution of delay-tolerant functions to off-peak hours. Third, allowing function owners to assign a criticality level to each function ensures that critical functions are executed first when the capacity is low. All of these might be provided as additional offerings to public cloud users at a discounted price to motivate adoption.

Although a public cloud would not be able to run functions from different users in the same Linux process like XFaaS does, we believe that large customers of public clouds like Netflix, Snapchat, and Twitch consume a significant amount of capacity, and those large customers can adopt XFaaS's approach in their virtual private cloud on top of public clouds. Specifically, among thousands of teams using XFaaS, a single team consumes 10% of the total capacity, while 0.4% and 2.6% of the teams consume 50% and 90% of the total capacity, respectively. Similarly, such large customers are likely to exist in public clouds as well and may be able to adopt XFaaS's approach.

7 Related Work

Surveys and applications. Several studies have provided surveys of the serverless computing landscape [20, 21, 26, 31], with some specifically focusing on the survey of serverless applications [13]. Moreover, specific types of serverless applications have also been studied before [25, 30, 48].

Scheduling functions and tackling cold start. Prior work illustrated several approaches for efficient scheduling of functions [27, 28, 39, 41]. XFaaS's deferred execution of delay-tolerant functions is unique and can result in significant capacity savings during peak time. Some works tackle the cloud function start-up latency problem [1, 2, 8, 10, 11, 16, 33–38, 44, 46, 47] using several architectural optimizations at the workers such as using microVMs [1] or microkernels. These can help speed up prewarm time when the execution environment is initialized. AWS Lambda recently introduced Snapstart [6] which stores and loads snapshots of the execution environment to further speed up cold start. In contrast, XFaaS eliminates cold start altogether in the common case since execution environments are pre-compiled using a region-based profiling-guided compilation in tandem with a staged rollout process to pre-provision workers with optimized code for the execution environment and all functions of that environment.

Industry and open-source systems. Several FaaS industry systems [4, 5, 17] and open frameworks [3, 22] already exist. However, none of them consider the interaction between the FaaS system and downstream services. XFaaS's backpressure mechanism leverages overload signals to reduce load on downstream services dynamically and automatically. Moreover, XFaaS's approach to widely push execution runtimes with pre-compiled optimization code can greatly reduce startup latencies; its unique deferred computation model can save capacity at peak periods; and XFaaS leverages locality groups to reduce workers' memory consumption.

Hardware cost of FaaS platforms. Wang et al. [45] benchmarked several public cloud FaaS platforms and found that all of them keep VMs idle for 10 minutes or longer after a function call, which could result in low hardware utilization. Several studies have reported the workload characteristics of public cloud FaaS platforms [39, 43]. However, they did not provide information on the hardware utilization of these platforms. As a rare example of research that focuses on the hardware cost of a FaaS platform, Zhang et al. [49] proposed using harvested resources to run functions.

8 Conclusion

We have introduced XFaaS, a serverless platform in our private cloud. XFaaS employs several techniques to enable workers to serve requests for any function almost instantly without cold start time, including proactive code deployment, cooperative JIT, cross-function runtime sharing, and locality groups for reducing memory consumption. XFaaS also leverages the insight that, although function invocations exhibit high spikes, not all functions require near-real-time latency. This insight allows for the postponement of computation for delay-tolerant functions to off-peak hours, resulting in significant capacity savings. Lastly, we advocate that workload management for an FaaS platform should be contextualized within a broader ecosystem, specifically considering the impact of spiky function executions on downstream services. A main piece of our ongoing work is to transition most functions using guaranteed quota to utilize opportunistic quota for additional capacity savings.

Acknowledgments

This paper presents a decade of work by several teams at Meta, including Async, FOQS, Web Foundation, Serverless Workflow, HHVM, and CEA. We thank these teams for their contributions and Shie Erlich for the support. We also thank all reviewers, especially our shepherd Yiying Zhang, for their insightful feedback which has helped significantly improve the quality of the paper.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *NSDI*, volume 20, pages 419–434, 2020.

[2] 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*, pages 923–935, 2018.

[3] Apache OpenWhisk. Accessed on April 4, 2023.

[4] AWS Lambda. Accessed on April 4, 2023.

[5] Azure Functions. Accessed on April 4, 2023.

[6] Jeff Barr. New - accelerate your lambda functions with lambda snapstart, 2022. Accessed on April 4, 2023.

[7] David Elliott Bell. Looking back at the bell-la padula model. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 15–pp. IEEE, 2005.

[8] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference*, pages 645–650, 2018.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 49–60, 2013.

[10] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand Container Loading in AWS Lambda. In *2023 USENIX Annual Technical Conference*, pages 315–328, 2023.

[11] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[12] Guoqiang Jerry Chen, Janet L Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1087–1098, 2016.

[13] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. Serverless applications: Why, when, and how? *IEEE Software*, 38(1):32–39, 2020.

[14] Marius Eriksen, Kaushik Veeraraghavan, Yusuf Abdulghani, Andrew Birchall, Po-Yen Chou, Richard Cornew, Adela Kabiljo, Ranjith Kumar S, Maroo Lieuw, Justin Meza, Scott Michelson, Thomas Rohloff, Hayley Russell, Jeff Qin, and Chunqiang Tang. Global Capacity Management with Flux. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[15] Jason Flinn, Xianzheng Dou, Arushi Aggarwal, Alex Boyko, Francois Richard, Eric Sun, Wendy Tobagus, Nick Wolchko, and Fang Zhou. Owl: Scale and flexibility in distribution of hot content. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 1–15, 2022.

[16] Alim Ul Gias and Giuliano Casale. Cocoa: Cold start aware capacity planning for function-as-a-service platforms. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8. IEEE, 2020.

[17] Google Cloud Functions. Accessed on April 4, 2023.

[18] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*, 2023.

[19] Katia Gil Guzman. How to Use Azure Queue-Triggered Functions and Why, December 2020. https://medium.com/swlh/how-to-use-azure-queue-triggered-functions-and-why-7f651c9d3f8c.

[20] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. Survey on serverless computing. *Journal of Cloud Computing*, 10(1):1–29, 2021.

[21] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[22] 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.

[23] HHVM. https://hhvm.com.

[24] Hive. https://hive.apache.org.

[25] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.

[26] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[27] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM symposium on cloud computing*, pages 158–164, 2019.

[28] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Practical scheduling for real-world serverless computing. *arXiv preprint arXiv:2111.07226*, 2021.

[29] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)*, 54(10s):1–34, 2022.

[30] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[31] Anupama Mampage, Shanika Karunasekera, and Rajkumar Buyya. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[32] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[33] Anup Mohan, Harshad S Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. *HotCloud*, 2019(10.5555):3357034–3357060, 2019.

[34] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400. IEEE, 2017.

[35] 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*, pages 57–70, 2018.

[36] Guilherme Ottoni. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[37] Guilherme Ottoni and Bin Liu. HHVM jump-start: Boosting both warmup and steady-state performance at scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*,

pages 340–350. IEEE, 2021.

[38] Rohan Basu Roy, Tirthak Patel, and Devesh 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*, pages 753–767, 2022.

[39] 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 *USENIX Annual Technical Conference*, 2020.

[40] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.

[41] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 311–327, 2020.

[42] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1013–1020, 2010.

[43] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *USENIX Annual Technical Conference*, 2021.

[44] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[45] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference*, pages 133–146, 2018.

[46] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The true cost of containing: A gvisor case study. In *HotCloud*, 2019.

[47] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020.

[48] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, 2020.

[49] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.