Serverless Cold Starts and Where to Find Them

Artjom Joosen, Ahmed Hassan, Martin Asenov,
Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, Adam Barker
sirlab@huawei.com
Central Software Institute, Huawei

Abstract

This paper analyzes a month-long trace of 85 billion user requests and 11.9 million cold starts from Huawei's serverless cloud platform. Our analysis spans workloads from five data centers. We focus on cold starts and provide a comprehensive examination of the underlying factors influencing the number and duration of cold starts. These factors include trigger types, request synchronicity, runtime languages, and function resource allocations. We investigate components of cold starts, including pod allocation time, code and dependency deployment time, and scheduling delays, and examine their relationships with runtime languages, trigger types, and resource allocation. We introduce pod utility ratio to measure the pod's useful lifetime relative to its cold start time, giving a more complete picture of cold starts, and see that some pods with long cold start times have longer useful lifetimes. Our findings reveal the complexity and multifaceted origins of the number, duration, and characteristics of cold starts, driven by differences in trigger types, runtime languages, and function resource allocations. For example, cold starts in Region 1 take up to 7 seconds, dominated by dependency deployment time and scheduling. In Region 2, cold starts take up to 3 seconds and are dominated by pod allocation time. Based on this, we identify opportunities to reduce the number and duration of cold starts using strategies for multi-region scheduling. Finally, we suggest directions for future research to address these challenges and enhance the performance of serverless cloud platforms.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

Keywords: cloud, serverless, datasets, cold starts, time series

1 Introduction

Serverless computing has become a widely used computing paradigm, with all major cloud providers expanding and improving their serverless offerings. Serverless platforms rely on systems such as Knative [19], YuanRong [6], and Nuclio [16] running on top of container management systems such as Kubernetes [22] to manage the infrastructure. This allows developers to focus on application code, written as functions, while the cloud service provider manages the underlying infrastructure.

For each function, users specify triggers, such as timers, that activate the function. When a function is triggered, a container with that function's code processes the request. If a new container needs to be started either because there are no active containers for this function or because existing containers are overloaded, this causes a *cold start*. This cold start adds significant latency, degrading application performance. Hence, there has been considerable effort to optimize serverless applications [21, 27], frameworks [1, 46], and their resource usage [24, 45].

Cold starts are a significant challenge in serverless systems [5, 23, 28, 40]. While an important problem, there is a lack of open-source data on cold starts from production systems. Currently, all available data from production serverless systems contain only aggregate statistics [18, 34]. While it has been suggested that cold starts are affected by trigger types, package sizes, and language runtimes, among other factors [18], to the best of our knowledge, there exists no detailed analysis of the relationship between these factors and cold starts in production deployments. To solve this problem, some studies have tried to measure cold starts from a user perspective and relate them to their sources [39]. However, these measurements may not be representative of workloads run by real users. Additionally, previous research lacks the provider's view of long-term trends across multiple data centers, making it difficult to suggest optimizations that exploit workload patterns observed in production workloads.

In this paper, we present a thorough analysis of 31 days of detailed metrics from serverless deployments in five of Huawei Cloud's data center regions. The data includes 85 billion requests from over 12 million pods, including over 11 million cold starts[1]. Our analysis highlights differences in peak time effects, resource usage, and cold start distributions between and within our data centers. Furthermore, we analyze the distribution and number of cold starts for different runtimes, trigger types, and resource allocations to identify which workloads tend to cause cold starts. Finally, the period studied in this paper contains a week-long holiday, enabling us to analyze workload changes before, during, and after a holiday.

This paper provides the results of our in-depth analysis as primary contributions. Specifically, we:

1. Analyze cold starts in production serverless deployments, including how cold starts relate to trigger types, synchronicity, language runtimes, and pod sizes.
2. Characterize factors that impact how cold starts differ across five different data center regions, providing

[1]Huawei traces available at https://github.com/sir-lab/data-release

insights into how different regions within the same cloud provider can exhibit varying performance.
3. Provide a first-of-its-kind component analysis on the different latency components in cold starts, providing insights into the causes of cold starts, and insights about where and how cold starts should be optimized.
4. We study long-term time-varying effects of cold starts and components, and identify the potential for spatial peak shaving between regions due to differing peak times and temporal peak shaving, which is enabled by substantial delayed pod allocations.
5. Based on our insights, we identify areas for improvement in data center scheduling and design, and provide several directions for future research, including resource pool prediction, concurrency adjustment and call chain prediction.

2 Background

Serverless computing allows developers to deploy event-driven functions without provisioning or managing servers or backend infrastructure [1]. This section starts by describing our platform and dataset. We then provide a brief background on cold starts, followed by a description of the analysis we perform on the dataset, and the shortcomings of recent works analysing serverless workloads.

2.1 Our Platform and Dataset

Huawei's serverless offerings rely on an in-house platform, YuanRong, a general-purpose serverless platform with a unified programming interface. YuanRong has been deployed for over three years across nearly 20 datacenter regions, processing up to 30 billion requests per day [6]. Each region is divided into four clusters. Clusters provide virtual and physical separations within a region, improving availability and fault tolerance.

Huawei's public cloud lets users upload function code and assign a resource limit for the function. These resource limits are grouped into CPU-memory configurations, such as '300-128', representing 300 millicores and 128MB of memory. Pools of pods with different resource configurations are maintained in case one is required by a cold start. If the autoscaler determines that additional pods are required to address incoming requests, pods are taken from the appropriate pool, the code of that function is loaded into it, and it is ready to process requests.

Requests for a given function may be balanced between clusters or routed to a single cluster. Several load-balancers receive requests and dispatch them to the nodes running function instances. The load-balancers keep track of the number of requests dispatched but not yet returned for each function. If there is a certain cluster with increased load, the system will balance traffic between the clusters within the region, starting pods in a new cluster if they do not exist, or shifting the load to existing pods. If there are no hot-spots in the clusters, a hash mechanism is used to dispatch requests to only one cluster.

Request level table, Regions: 5, Duration: 31 days		
Name	Description	Res
Timestamp	timestamp at worker	ms
Pod ID	hashed pod ID	-
Cluster name	cluster name	-
Function name	hashed function name	-
User ID	hashed user ID	-
Request ID	hashed request ID	-
Execution time	execution time	μs
CPU usage	CPU usage	millicores
Memory usage	memory usage	Bytes
Pod level table, Regions: 5, Duration: 31 days		
Timestamp	timestamp	ms
Pod ID	hashed pod ID	-
Cluster name	cluster name	-
Function name	hashed function name	-
User ID	hashed user ID	-
Cold start time	total cold start time	μs
Pod alloc. time	time to get pod from pool	μs
Deploy code time	time to deploy code	μs
Deploy dep. time	deploy dependency time	μs
Scheduling time	scheduling overhead time	μs
Function level table, Regions: 1, Duration: 31 days		
Function name	hashed function name	-
Runtime	runtime	-
Trigger type	trigger type	-
CPU-MEM	CPU-MEM config	-

Table 1. Summary of our dataset fields with each field's associated timestamp granularity.

Our dataset. In this paper, we analyze detailed metrics from five different regions collected from a total of 20 clusters. The data does not include all functions from all regions in our data centers, but provides a good representation of how serverless production systems operate across multiple regions. The dataset comes from three different monitoring streams: request level monitoring, pod level monitoring, and function level monitoring. From the request level monitoring we analyze and release data from five regions including: timestamp, pod ID, function name, user ID of the function owner, request ID, request execution time, as well as CPU and memory usage of the request. From pod level monitoring, we analyse data logged during cold start events, including timestamp, pod ID, cluster ID hosting the pod, function name, user ID, total cold start time, and components of different parts of the cold start including pod allocation time, code deployment time, and dependency deployment time. We also analyze trigger types of different functions along with their CPU and memory configuration, and the runtime of these functions. For privacy reasons, all IDs are hashed. We summarize the fields we use in our analysis in Table 1.

To the best of our knowledge, this is the most detailed serverless dataset released by any cloud provider. The dataset has event-level metrics for a total of 85 billion requests and

over 11 million cold starts over a period of 31 days with a week-long major holiday period. Figure 1 gives an overview of the size of the data for each region, numbered R1, R2, R3, R4, and R5. Most functions run on multiple clusters in a region. Finally, some functions have no load-balancing in a single region and are deployed only on a single cluster. The dataset hence aims to capture most of the possible dynamics for serverless production systems.

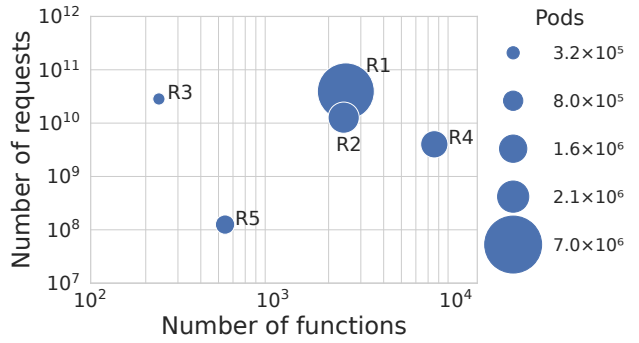


Figure 1. Plot showing the number of requests, functions, and pods for all five regions.

2.2 Cold starts in our System

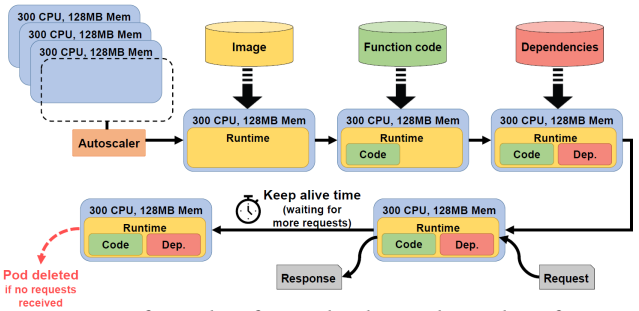


Figure 2. Life cycle of a pod. The pod is taken from its resource pool, loaded with a runtime, function code, and dependencies. After serving requests, the pod waits for additional requests for a designated keep alive time. If the pod receives no more requests during this time, it is deleted.

One of the main bottlenecks of serverless systems is cold starts, where the system must start a new pod if it does not have sufficient resources to process an incoming request [18, 23, 39]. The number and time of cold starts vary widely across different runtimes, trigger types, and resource allocations. We analyze how cold starts are affected by these factors. In addition, we analyze the times spent in the different parts of our serverless system during a cold start, as shown in Figure 2, including pod allocation, deploying the compressed function file, deploying dependencies, and scheduling.

As shown in Figure 2, our system maintains resource pools of inactive pods with different CPU and memory allocations, e.g. 300 millicores with 128MB of memory. When a function is cold-started, the scheduler selects a pod from the pool with that CPU and memory configuration. Pods have several popular runtimes and libraries preinstalled. Users can also provide a custom container image for other runtimes not supported by default, which is downloaded into the pod when needed. When the runtime is ready, the function code is pulled. Additional dependencies are downloaded if required. Once the pod is prepared, it serves the request that spawned it and returns the result. The pod then waits for additional requests for a designated keep alive time. In our system, this time is set to one minute by default. If the pod receives no requests during this time, it is deleted. Otherwise, the keep-alive time is reset back to one minute with each new request.

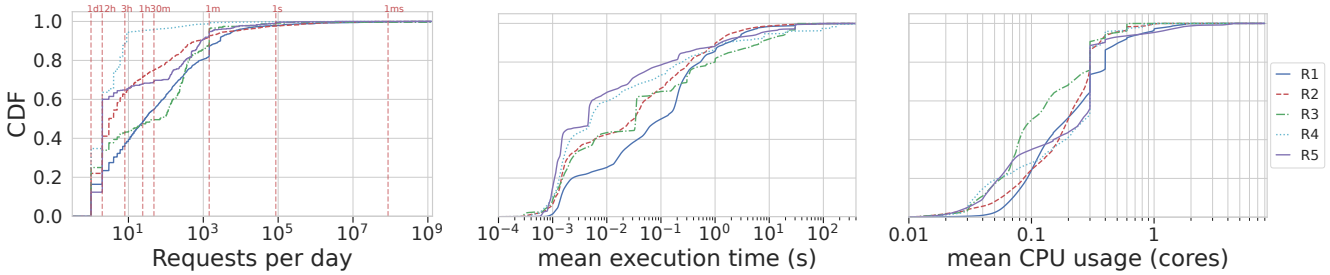
3 Multi-region Serverless in Production

In this section, we provide a high-level overview of similarities and differences between our regions. Figure 1 shows the size of each region by their number of requests, functions, and pods, showing differences of several orders of magnitude between regions. We observe that a larger number of functions does not necessarily mean more requests or pods. We start our analysis in this section by examining differences between regions in their resource usage, latency, peak times, peak-to-trough ratio, and holiday effects.

3.1 Region Statistics

Figures 3a, shows CDFs of the number of requests per function per day in each region, with dashed vertical lines representing the equivalent mean request inter-arrival time per function. A large majority of functions have very few requests per day on average. However, a small minority of functions have a very large number of requests and dominate resource usage. There are also large differences between regions in the number of functions with high load. Approximately 20% of functions in Region 1 have at least one request per minute on average, while this is approximately 1% for Region 4. We note that the average hides the per function patterns. Many of our functions have a large variation in their inter-arrival rates throughout the day.

Figures 3b, and 3c show the mean execution time per minute and mean CPU usage per minute. Similar to the variations in number of arrivals per function, we also see significant variation in execution times and the CPU usage per region. Median execution time varies between 4ms in Region 5 to 100ms in Region 1. Execution time can be as long as several tens or hundreds of seconds. Additionally, the median CPU usage varies between 0.1 cores in Region 3 to 0.3 cores in Region 3. These significant differences in CPU usage and execution time between regions present opportunities for cross-region scheduling.

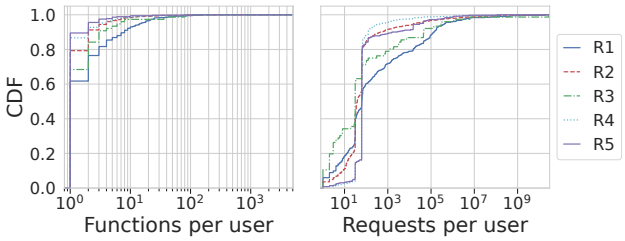


(a) CDF of sum of requests on median day per function for each region.

(b) CDF of mean execution times per minute for each region.

(c) CDF of mean CPU usage in cores per minute for each region.

Figure 3. CDF of invocations, execution time, and CPU usage for all regions.



(a) Functions per user for all regions. **(b)** Requests per user for all regions.

Figure 4. Number of functions per user and number of requests per user for all regions over the duration of the trace.

Functions per user. Figure 4a shows the CDF of number of functions per user while Figure 4b shows number of requests per user in each region. Figure 4a shows that 60% to 90% of users have only a single function, depending on the region, with almost all users having fewer than 20 functions. The two figures show that the number of functions and requests tend to be more concentrated in fewer users in smaller regions, and less concentrated in larger regions. For example, 30% of users in Region 1 have more than 1000 requests, while less than 5% of users have 1000 requests in Region 4. This means that, for many users, migrating their functions to a different region may be feasible due to a small number of functions.

Previous work has shown that users understand their own resource requirements poorly, and tend to choose regions close to their end users, not considering cheaper or faster options [35]. This is partly because users do not know the behaviours of other users, unlike the cloud provider, which can see user behaviours and schedule more efficiently than users can. In addition, our regions are spread out geographically, with power prices and carbon costs varying accordingly. As we show later, the latency between regions can be insignificant compared to the longer cold starts and execution times in the more popular regions. By moving some requests to other regions, it may be possible to improve latency, cost, or carbon footprint [2, 3]. Such approaches have been previously suggested for batch workloads [14]. However, we believe that it can also be used for serverless systems with minimal effect on execution time.

Cross-region scheduling potential

Regions can have very different profiles. For example, median invocations per function, function execution time, and CPU usage between regions vary by factors of up to 50, 25, and 3, respectively, which presents opportunities for cross-region load balancing. Enabling such scheduling has the potential to reduce latency, cold starts, and costs in serverless public clouds.

3.2 Peak Time Analysis

Peak times are a well-known phenomenon in serverless systems, with several earlier works concerned with predicting and mitigating such surges [7, 18, 34]. Peak times present challenges such as large fluctuations in resource allocation and network congestion, causing increased latency. However, to the best of our knowledge, peak workloads have not been studied across multiple regions. In this section, we examine the changes that occur in the number of requests during peaks and troughs. Other prior work has shown that different peak times present opportunities for peak load shaving and can be exploited for scheduling and load balancing, as has been done in some private serverless cloud platforms [32]. Such solutions are particularly viable in scenarios consisting of many non-latency critical workloads.

Peak time lags. Peak times occur in each one of our regions. However, they often occur at different times of the day. Figure 5 shows the normalized request patterns for each of our regions for a three day period. The largest peak every day is marked with a red line. We see clear periodic behaviour in all of our regions, consistent with existing literature [18, 34]. The largest peaks tend to occur at a different time of day in every region, with smaller secondary peaks also present in

many regions. Some regions have more prominent periodic behavior compared to others.

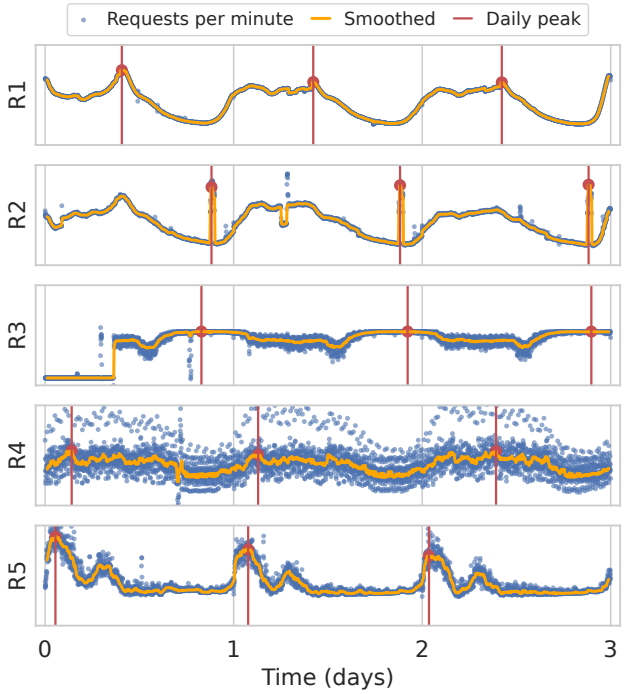
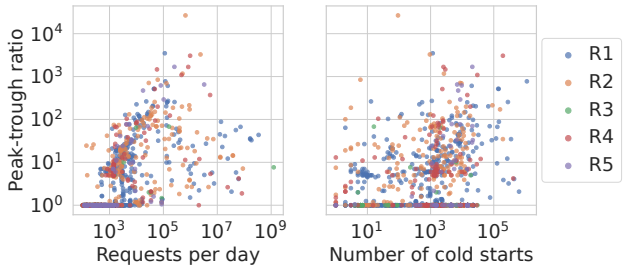


Figure 5. Plot showing peaks in normalized number of requests per region. Peaks detected on a smoothed version of the signal. The largest peak in 24 hours is highlighted.

Peak-to-trough ratios for function invocations. One way of analyzing workload periodicity and burstiness is to measure their peak-to-trough ratios, which is the ratio of the largest peak in a periodic pattern to its lowest trough. This is a measure of the strength of periodic oscillation. Large peak-to-trough ratios indicate a function with large bursts of invocations.

Figure 6a shows a scatter plot of each function's peak-to-trough ratio against its median invocations per day. Peak-to-trough ratios tend to be lower for low-request functions, high for moderately popular functions, and lower again for very popular functions. We see a wide variety of peak-to-trough ratios, from less than 2 to over 1000. Huge variations are less common in workloads with larger number of requests, and the largest workloads experience peak-to-trough ratios less than 60. Additionally, we see a cluster of functions with fewer than 1440 requests per day (corresponding to 1 request per minute) with peak-to-trough ratios close or equal to 1. These functions are invoked on average once per minute and do not have enough requests to have identifiable peaks. Other works have previously reported peak-to-trough ratios of similar magnitude in production serverless public clouds (over 500) [38].

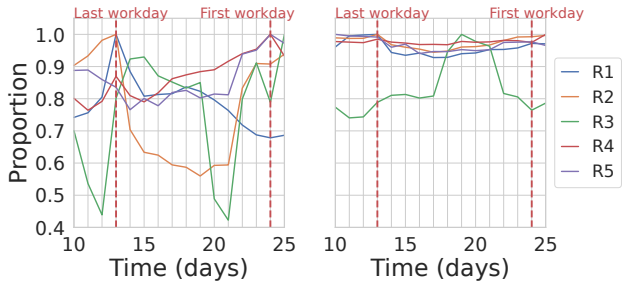
Figure 6b shows the peak-to-trough ratio of each function against the total number of cold starts for that function



(a) Scatter plot of functions by their median requests per day and peak-to-trough ratio.

(b) Scatter plot of number of cold starts over 31 days for a function against its peak-to-trough ratio.

Figure 6. Scatter plots showing a function's requests per day and number of cold starts against its peak to trough ratio. Functions with a constant value of requests per minute, or no identifiable peaks have a peak to trough ratio of one.



(a) Change in allocated pods.

(b) Change in mean CPU usage.

Figure 7. Normalized number of pods and allocated CPU during holiday period. Day 13 is the last working day before the holiday. Day 24 is the first working day after the holiday.

over the duration of our dataset. We observe a concentration of functions with larger peak-to-trough ratios and a higher number of cold starts. Additionally, we see a cluster of functions with peak-to-trough ratios close (or equal) to 1, that tend to have fewer than 44640 cold starts (equivalent to one cold start per minute for 31 days), which corresponds to the same group of infrequently invoked functions described in Figure 6a. Hence, we see that high numbers of cold starts are a result of workloads with large fluctuations in their invocation patterns leading to frequent autoscaling decisions, or a large number of functions that are all invoked at most once per minute, falling just outside of the pod keep alive time. A successful policy to minimize the number of cold starts will address both of these sources.

Holidays. Holidays and other special occasions may have a significant impact on cloud traffic. Certain workloads may increase or decrease during and around holiday periods [15], while other workload types (e.g. associated with shopping and commercial loads) increase before holidays [4]. Our dataset includes a week-long holiday period. To the best

of our knowledge, this is the first serverless dataset that includes such patterns.

Figure 7 shows the number of pods and mean CPU usage, normalized to their maximum value during the same number of days before the holiday. The last working day before the holiday is on day 13, and the first working day after the holiday is day 24. Both Figures 7a and 7b show a similar pattern. Regions 1, 2, 4, and 5 all peak on day 13, decrease during the holiday, and increase to another peak on day 24. This suggests a pre-holiday 'rush' and post-holiday 'catch-up'. Region 3 experiences a different pattern, where its workload increases substantially at the start of the holiday, and reduces again towards the end.

Complex origin of cold starts

High numbers of cold starts can be contributed both by workloads with large fluctuations in invocation patterns, or simply by a large number of functions that are invoked with a periodicity greater than one minute, falling just outside the pod keep alive time. Some functions have peak-to-trough ratios greater than 1000. A successful policy to minimize the number of cold starts will address both of these sources. Holidays usually decrease resource allocation, with some regional variations.

3.3 Runtimes and trigger types

We now describe the distributions of trigger types, runtimes, and their combinations focusing on Region 2. A function can use one of preinstalled runtimes or use a custom image. Preinstalled runtimes include *C#*, *Go*1.*x*, *Java*, *NodeJS*, *PHP*7.3, *Python*2 (for legacy reasons), *Python*3, and *HTTP*. When deployed, a function is invoked using one of many triggers. Some trigger types can only be invoked asynchronously, while others can be invoked synchronously. For synchronous (S) requests, the invoking program waits for a response, while for asynchronous (A) requests the invoking program does not wait for a response, typically checking the result later. Our platform supports the following trigger types:

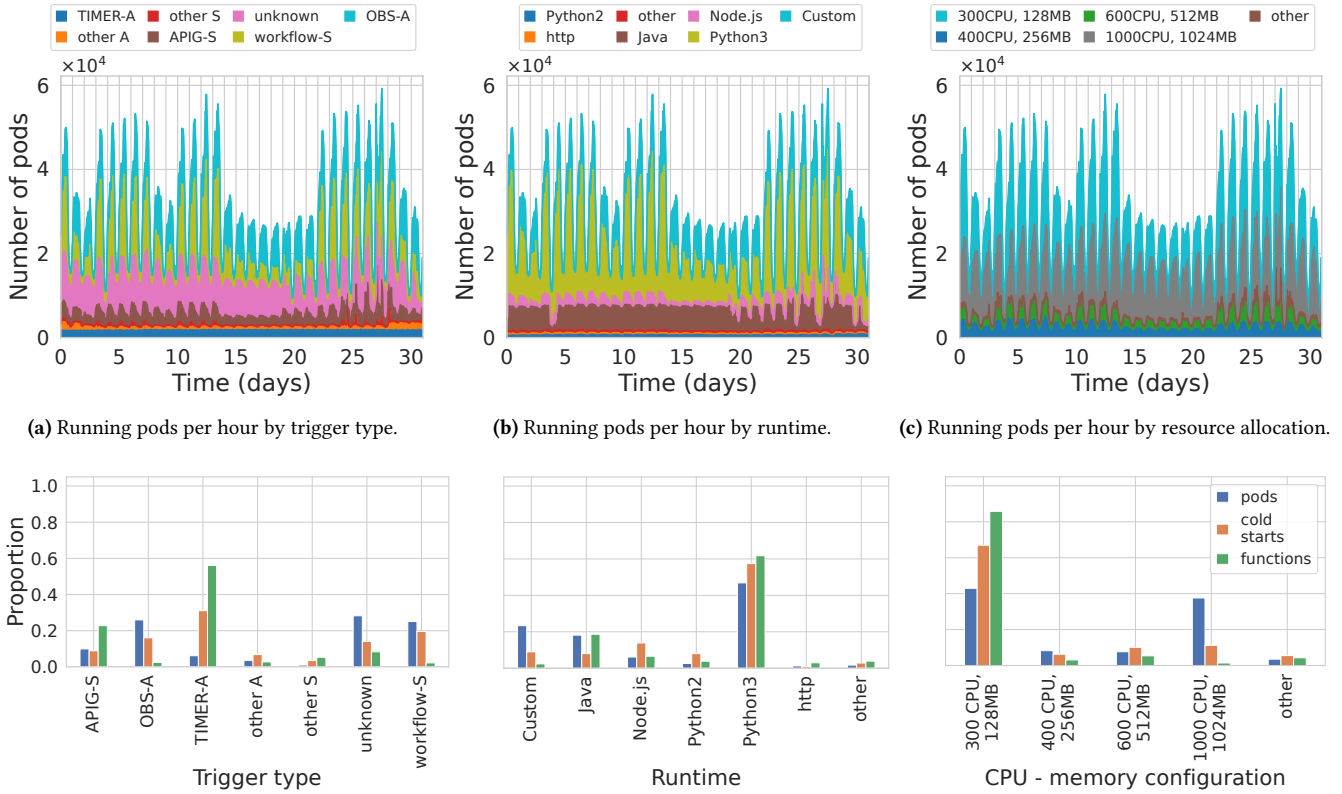
1. **API gateway (APIG) (S and A)** is an API hosting service. The function can be invoked through HTTPS using a custom REST API and a specified backend;
2. **Timer** invokes functions based on a cron-style timer;
3. **Cloud Trace Service (CTS) (A only)** is a platform monitoring service;
4. **Data Ingestion Service (DIS) (A only)** triggers functions by a data stream, e.g. to process updated records;
5. **Log Tank Service (LTS) (A only)** triggers functions by logging events;
6. **Object Storage Service (OBS) (A only)** invokes functions by storage events, e.g. object creation or deletion;
7. **Simple Message Notification (SMN) (A only)** triggers functions using messages posted under topics;
8. **Kafka** triggers functions using a Kafka queue;
9. **Workflow (S and A)** lets functions directly trigger other functions.

In addition, our system supports a function to have multiple trigger types. Since some of these trigger types are seldom used, we aggregate them in our analysis. We aggregate all trigger types except for timers, OBS-A, APIG-S, and workflow-S. We split the aggregation into other synchronous (other S) or other asynchronous (other A). Timers account for 42% of all triggers, followed by APIG-S (23%), APIG-S and TIMER-A combination (13%), with the remaining triggers and trigger type combinations representing less than 5% of functions each. The majority of functions only have one trigger type, with only a handful having two or more trigger types.

Trigger types by runtime. To analyze the relationship between runtime and trigger type, Figure 9 shows a stacked bar chart of trigger types as proportions of the total number of functions for a given runtime. We note that the prevalence of different trigger types varies considerably between runtimes. For example, *Python*3, *PHP*7.3, and *Node.js* functions are mostly triggered by timers, while *Java* and *HTTP* runtimes tend to use APIG-S triggers. Asynchronous triggers other than OBS and timers are most significant in *Python*2, and even then only account for a minority of functions. We note that a small proportion of our data does not have the runtime or the trigger types logged.

Pods by runtime and trigger type over time. Since invocations have periodic patterns, it is worth investigating if a similar periodicity can be observed in the number of running pods when grouped by runtime language or trigger type. Such patterns may be useful in workload collocation, pre-warming, and scheduling. Figures 8a and 8b shows the average number of pods per hour in Region 2 by trigger type and runtime for the full duration of our dataset. While some runtimes and trigger types have a stable number of pods, others show significant variability. For example, *Python*3 runtimes and workflow-S trigger types show large variability that correlates with time of day. Both peak during working hours, and decrease during nighttime. Notably, Figure 8a shows that the number of pods allocated for timers does not vary much with time, even though they represent almost 60% of functions as shown in Figure 8d. Many timer functions tend to be invoked infrequently (at most once per minute) and are only allocated a single pod.

Daily periodicities in public cloud workloads have previously been attributed to timers and diurnal user patterns [18]. Our analysis suggests that timer workloads do not contribute significantly to the total platform load periodicity, but that it is user-driven diurnal behaviour driven by APIG calls or



(a) Running pods per hour by trigger type. **(b)** Running pods per hour by runtime. **(c)** Running pods per hour by resource allocation.

(d) Proportions grouped by trigger type. **(e)** Proportions grouped by runtime. **(f)** Proportions grouped by resource allocation.

Figure 8. Proportions of running pods, cold starts, and functions by trigger, runtime, and resource allocation in Region 2.

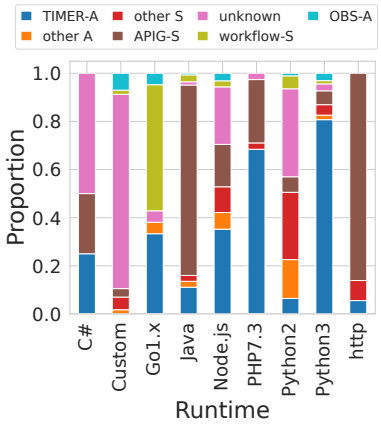
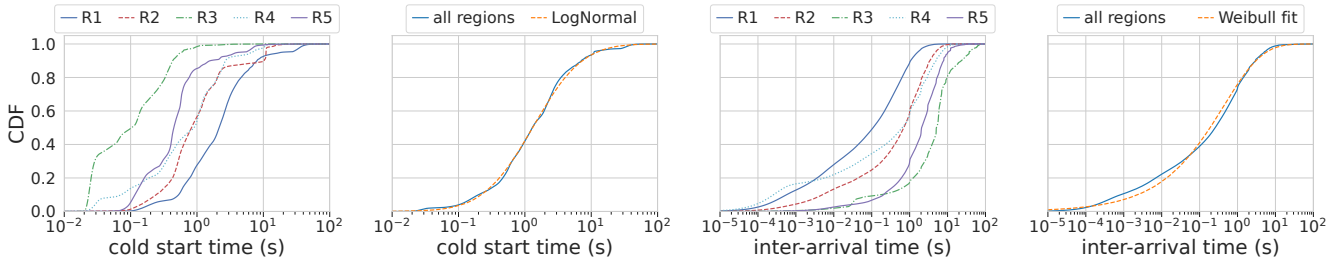


Figure 9. Region 2 trigger types by runtime.

workflows that contribute more to daily periodicity. Such predictable patterns may be exploited by serverless management systems and cluster managers to reduce the number and duration of cold starts by prewarming instances of these runtimes according to these patterns. We also note that most trigger types and runtimes have strong weekly periodicity, with approximately 30% more pods allocated during weekdays compared to weekends. This weekly periodicity is interrupted during the holiday period beginning on day 14, with day 13 being the last working day, with workload level similar to weekend levels. Timer-triggered functions are almost completely unaffected by the holiday period.

Synchronous trigger types, such as workflow-S and APIG-S, have very strong daily periodicity and contribute significantly to peaks in the overall number of allocated pods on the platform. Given that these requests are synchronous, there may not be significant potential for flexible scheduling. However, OBS triggered functions, which are asynchronous, also have very strong daily oscillations and contribute significantly to the peak time pod allocations, and account for almost 30% of running pods, as seen in Figure 8d. Asynchronous triggers may be used for tasks that are not latency critical, such as log batch analysis triggered by a new logging event (LTS) or the presence of new files (OBS). Hence, if the provider can differentiate between asynchronous triggers that have less latency critical deadlines versus asynchronous requests that are time-critical but where the system does not wait for the response, peak-shaving can be used, whereby the allocation of these pods and execution of these requests is delayed. Given the narrow peak widths, even a short delay could significantly reduce peak pod allocations.



(a) Cold start times per region. **(b)** Fit for cold start times. **(c)** CDF of inter-arrival times. **(d)** Fit for inter-arrival times.

Figure 10. CDFs and distribution fits for cold start times and inter-arrival times between cold starts. LogNormal fit for cold starts has mean 3.24 and standard deviation 7.10. Weibull fit for inter-arrival times has mean 1.25 and standard deviation 3.66.

Finally, we note the changing periodicity characteristics of certain workloads. For example, the number of pods serving *Java* functions varied very little until day 18, after which a strong diurnal periodicity began. This means that a serverless platform must be able to detect these changes in the workload characteristics as some of these changes, if well detected, can present new opportunities for online system optimizations.

Pods by CPU and memory allocation. Figure 8c shows the number of running pods over time grouped by their resource allocation. We can see that these different resource configurations contribute different amounts to the total periodic fluctuations. As described in Section 2.2, our platform maintains pools of inactive pods to be used as demanded by user traffic. The predictability of these patterns may allow the provider to maintain just enough pods to meet expected demand, while avoiding excessive overallocation using online predictors and dynamic resource pre-warming.

Predictive scheduling and peak shaving

Function invocations follow periodic patterns that could be leveraged to pre-warm pods with popular configurations, thus reducing cold starts. Delaying pod allocation for asynchronously invoked functions could reduce peaks if they are not latency critical.

4 Causes of cold starts

We now conduct an in-depth analysis of cold starts. We first study cold starts and their components in all regions. We then study Region 2 in further depth, examining how cold starts vary by trigger type, runtime, and resource allocation.

4.1 Cold start distributions

We start our analysis by plotting CDFs of the duration and inter-arrival times of cold starts. Figure 10a shows the distribution of cold start times for different regions. We see large variations in cold start times between different regions, with medians between 0.1 seconds and 2 seconds. We see that cold start times in all regions have a long tail. Figure 10c shows the distribution of inter-arrival times (IAT) between cold starts for different regions. Median inter-arrival times range from 0.1 seconds in R1 to several seconds in R3.

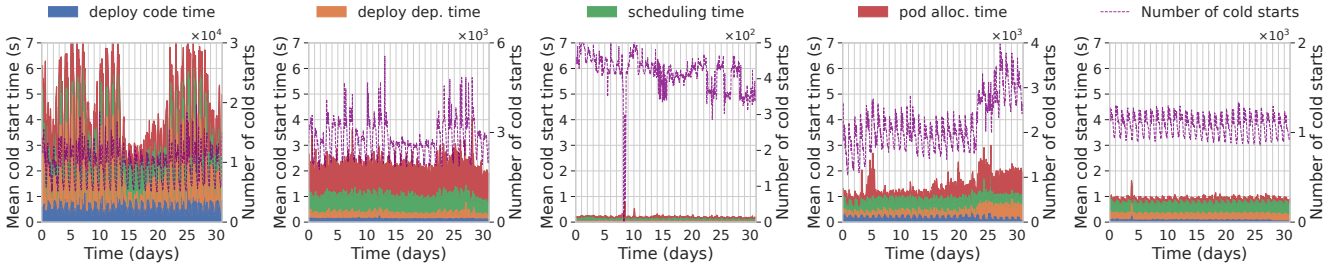
To capture the average behavior across regions, e.g. for simulation purposes, we fit a distribution to all cold start times and another distribution for cold start inter-arrival times. Cold start times can be approximated with a LogNormal distribution, while inter-arrival times can be approximated with a Weibull distribution, a common distribution for modelling event inter-arrival times [20]. Figure 10b shows the distribution of cold start times across all regions with a LogNormal fit with mean 3.24 and standard deviation 7.10, and Figure 10d shows inter-arrival times for all regions with a Weibull fit with mean 1.25 and standard deviation 3.66. These fits are very close to the measured data from our system. We believe that researchers working on problems related to cold start time optimizations can use these fits to run simulation experiments for cold start optimizations.

4.2 Components of cold start times

When a cold start occurs, several steps are required until the new pod is operational. We log the time taken for each step taken during a cold start, namely; the time taken to start a pod if no free pods exist or to select a pod from the existing pool to be used by the newly started function (referred to as *pod allocation time*); the time taken to download, extract, and deploy function code (referred to as *deploy code time*); the time to fetch and load dependencies (referred to as *deploy dependency time*); and the time for networking, routing, and scheduling overheads (referred to as *scheduling time*).

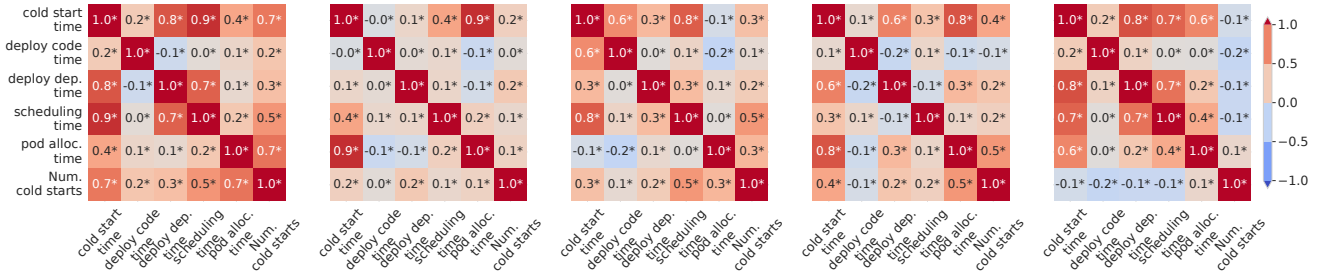
Figure 11 shows the mean cold start time per hour for each region, with stacked areas representing component times along with the total number of cold starts per hour. We plot the time taken to allocate a pod, deploy code, deploy dependencies, and scheduling, which together add up to the total mean cold start time on the left-hand axis. We see that the relative proportion taken up by each component varies over time as well as between regions.

In absolute and relative terms, pod allocation time in Region 2 is much higher than in Region 1. Mean total cold start



(a) Region 1 **(b)** Region 2 **(c)** Region 3 **(d)** Region 4 **(e)** Region 5

Figure 11. Mean cold start time per hour split by components on left axis. Number of cold starts per hour on right axis.



(a) Region 1 **(b)** Region 2 **(c)** Region 3 **(d)** Region 4 **(e)** Region 5

Figure 12. Spearman correlations of mean cold start time components per minute aggregated over all functions for each region. Correlation values where $p < 0.05$ are marked with an asterisk.

times vary between 3 seconds in Region 1 to less than 0.3 seconds in Region 3. Periodicity of components varies. For example, pod allocation time in Region 2 has the largest oscillations of all components in that region and is in phase with the number of cold starts. Time to deploy dependencies and scheduling time oscillate less. Meanwhile, time to deploy code remains almost constant. We can see a significantly different pattern in Region 1, where the time to deploy code and dependencies both oscillate significantly.

Day 23 is the first working day after the holiday, and all regions show an increase in number and duration of cold starts then. This effect is especially pronounced in Regions 1, 2, and 4. These regions in particular also see a strong increase in time to deploy dependencies and pod allocation time. These may be caused by first-time function code deployments following a prolonged period of inactivity, as well as competition for a small number of reserved pods.

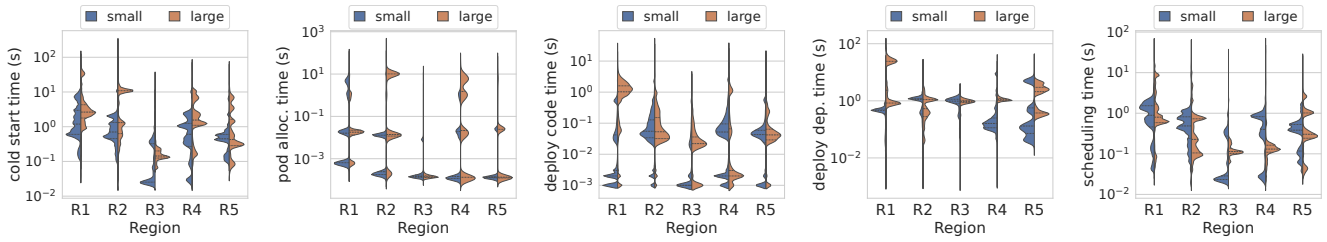
The different composition of cold start times in different regions at different times of day may occur due to differing workloads between regions or due to architectural differences between data centers. Different data centers may use different architectures, network topologies, or other structural differences that cause bottlenecks in different parts of the system when scaling. Identifying which component to optimize must be done within the context of the data center's architecture as well as in-depth workload analysis.

Correlations between cold start time components. Figure 12 shows Spearman correlation values between cold start time and its components, as well as the number of cold starts, averaged over all functions in that region. We can observe several trends that generalize across regions. Cold start time tends to be positively correlated with the number of cold starts, although the strength of this correlation varies across regions. Scheduling time and pod allocation time are positively correlated with the number of cold starts, especially in Regions 1, 3, and 4, suggesting that increasing demand for cold starts may cause delays in scheduling and pod allocation. There are also several more isolated, stronger correlations, such as those between scheduling and deploying dependencies in Regions 1 and 5, pointing to region-specific bottlenecks.

Cold start components across regions and time

Cold start times and the components that dominate them vary significantly between regions, pointing to workload differences and potential effects of hardware setups. Mean cold start time tends to correlate positively with number of cold starts.

Resource allocation and cold start time. Our system maintains pools of pods of different resource configurations,



(a) Total cold start time. **(b)** Pod allocation time. **(c)** Time to deploy code. **(d)** Deploy dependency time. **(e)** Scheduling time.

Figure 13. Violin plot of cold start time and components by pool size for all regions with normalized width. For functions without layers, deploy dependency time is zero and excluded from plots. Dashed lines in the density plots represent quartiles.

ranging from 300 millicores of CPU and 128MB of memory to 26 cores and 32GB of memory. We aggregate these pools into two groups: smaller pods with at most 400 millicores of CPU and 256MB of memory, and larger pods with allocations larger than that. In order to observe if there is a difference in the distribution of cold start times between small and large pods, Figure 13 shows a violin plot of the distributions of cold start times and their components for small and large resource pools. We observe that larger resource pools tend to have longer median cold start times. The ratio of median cold start times between the high and low resource pods ranges from approximately 1:1 (Region 5) to 5:1 (Region 3).

Figure 13b shows the time to allocate a pod for large and small resource pools. During a cold start, the first step is to search for a pod of the appropriate configuration in a given resource pool. This search is conducted in stages, with the search expanding if a pod is not found. These stages clearly show themselves in the multimodal distribution in pod allocation time. For small resource pools, the pod allocation time tends to take less time, with a smaller proportion of cases expanding to further stages of the search with longer latency. For larger resource pools, the search tends to expand more frequently to later stages, taking a longer time to allocate a pod in total. This trend is consistent across all five regions. Figure 13c shows the time to deploy function code in small and large pods. We see that function code typically takes longer to deploy in larger pods, which is particularly pronounced in Regions 1, 2, 3, and 5. Figure 13d shows the time to deploy dependencies, if any. We see that time taken to deploy dependencies is longer for larger pods compared to smaller ones. Finally, Figure 13e shows the scheduling time in small and large pods. For regions 1, 3, and 4, smaller resource pods tend to have shorter scheduling times than larger pods. Regions 2 and 5 see the opposite pattern, where large pods have shorter scheduling times. Longer code and dependency deployment time may point to more complex code being deployed in larger pods.

4.3 Which functions cause the most cold starts?

We now focus our analysis on Region 2. We choose Region 2 as it has several interesting characteristics, such as large changes during the holiday period and large variations in different cold start components. Figures 8d, 8e, and 8f show the proportion of pods, cold starts, and number of functions accounted for by different trigger types, runtimes, and resource allocations respectively. The proportion of pods is calculated using the mean number of active pods per minute, while the proportion of cold starts is calculated using the number of newly started pods. Figure 8d shows that different trigger types account for vastly different proportions of pod allocations, cold starts, and functions. For example, timers account for almost 60% of functions and 30% of cold starts, but only 5% of running pods. Similarly, *Python*3 runtimes account for almost 50% of all cold starts. For resource allocations, small CPU-memory allocations account for more than 60% of cold starts. A provider can use the above data to decide the different percentages of pre-warmed runtimes and configurations. For example, since a large proportion of all functions are deployed as *Python*3 and also with a small CPU-Memory configuration, a provider can pre-warm a larger number of pods with *Python*3 deployed in small CPU-memory configuration pods.

To better understand how often cold starts occur, Figure 14 shows the total number of requests for each function against the total number of cold starts in Region 2, with each function colored by its trigger type. The diagonal red line represents the one-to-one case where each request causes a cold start. Most functions are invoked infrequently and therefore tend to be cold started every time they are invoked, with the majority of these being triggered by timers. Functions with more than 1 request per minute on average have fewer cold starts compared to the number of requests due to the pod keep-alive time which is set to one minute.

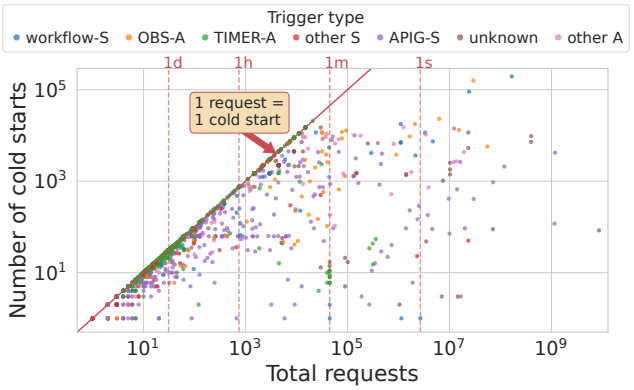


Figure 14. Total number of requests per function vs number of cold starts colored by trigger type in Region 2. Each point is one function. The dashed red lines show the equivalent mean inter-arrival time.

The impact of timer-triggered functions

A significant source of cold starts is the discrepancy between timer intervals triggering functions and the pod keep-alive time. Pre-warming pods for timer-triggered functions could alleviate cold starts for timer functions. For functions with timers less frequent than the pod keep-alive time, releasing resources sooner could improve overall resource usage.

4.4 Runtime languages and cold start time.

Previous work has found significant differences in cold start time between different runtime languages [39]. Figure 15 shows cold start time distributions by runtime language, along with cold start component times. In all distributions, the yellow curve labelled 'all' represents cold start times for all runtimes. We make the following key observations:

- Cold starts in different runtimes are dominated by different components. For example, *HTTP* cold starts are dominated by pod allocation time, while *Node.js* cold starts are dominated by scheduling. *Go* pods have much higher dependency and code deployment times compared to scheduling overheads.
- Scheduling time overheads are on average much higher than all the other overheads.
- For most runtimes, most cold start times are below one second with a long tail, meaning that a small proportion of cold starts takes several seconds regardless of runtime. The only exceptions are *Custom* and *HTTP*, where the median is greater than 10 seconds.
- While a small fraction of running pods (see figure 8e), *Custom* and *HTTP* skew the distribution of total cold start times, increasing the tail of the combined distribution. Long cold start times from *Custom* and *HTTP* come from pod allocation, while other components are negligible by comparison. Slow cold starts for *Custom* runtimes can be explained by the pod allocation process. Resource pools are not maintained for *Custom* runtimes, so these are created from scratch every time they are required. For *HTTP*, cold starts tend to be slow since they require the start of an *HTTP* server.

Trigger type and cold start time. We now perform a similar analysis on the distribution of cold start times for different trigger types in Region 2. Figure 16 shows cold start time and component CDFs by trigger type. Figure 16a shows that functions triggered by Object Storage (OBS) tend to have a slow cold start time with a median of 10 seconds.

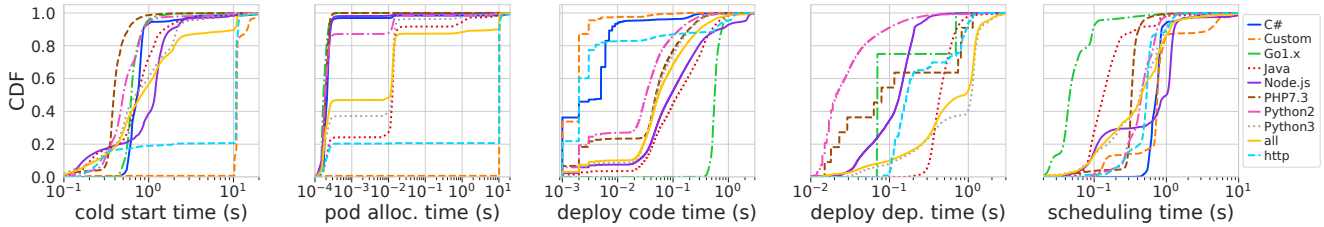
It is difficult to attribute cold start times to a specific factor. However, considering the complexity of FaaS scheduling systems, as shown in Figure 2, it is possible to get an idea of potential bottlenecks affecting cold starts. For example, in Figure 16a, the distribution for OBS has a median of 10 seconds while others have a median less than 1 second. That said, such a disparity is not necessarily caused only by the OBS trigger. Figure 9 shows that the most frequently logged known trigger type for *Custom* runtimes is OBS. In this case, the cause for the longer cold start times for OBS triggers is the fact that these functions tend to have *Custom* runtimes, which do not have reserved resource pools and therefore require pods to be started from scratch.

Causes of cold starts and dominant components

There is large variability in cold start times and components driving them for different trigger types, runtimes, regions, and over time. Functions with larger resource allocations tend to have cold start times between 2 and 5 times longer compared to functions with smaller resource allocations, driven by pod allocation and code and dependency deployment. While cold starts for some trigger types, such as OBS, can be mitigated by improved networking and storage, to the best of our knowledge, there is no single solution that can reduce cold starts for all cases.

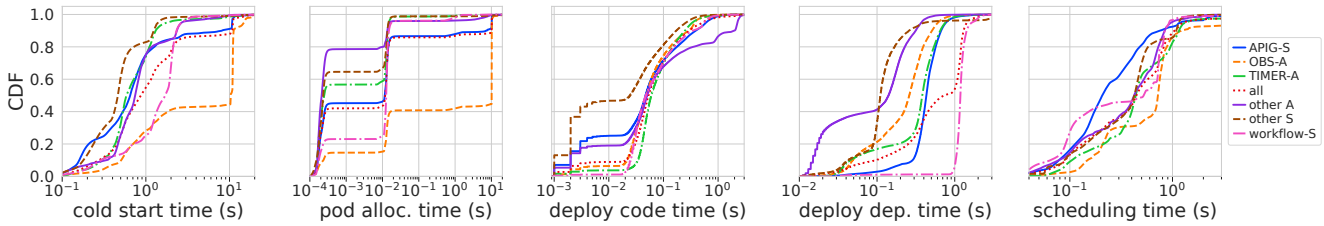
4.5 The real cost of cold starts

Optimizing cold starts has been a popular research topic. In our work, we want to obtain a more complete understanding of cold start costs by analyzing cold starts delays relative to pod lifetime. A pod with a long cold start time is more efficient when that pod lasts longer and serves more requests than if it is deleted immediately after serving the request that spawned it. Hence, we study the ratio of a pod's useful lifetime to its cold start time. Useful pod lifetime is calculated by subtracting keep alive time (1 minute) from total pod lifetime. A 1:1 ratio or less means that the pod is used for a time less than or equal to its cold start time. A higher ratio



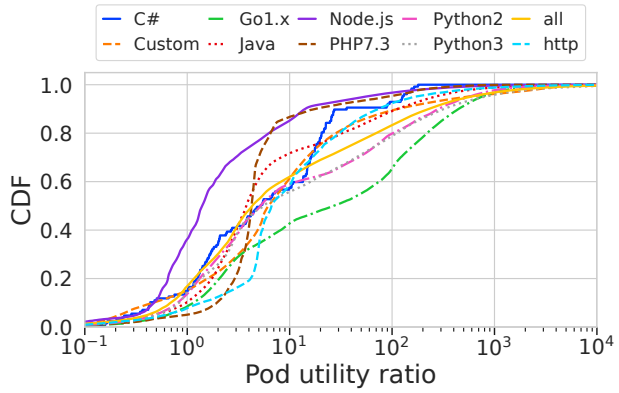
(a) Total cold start time. **(b)** Pod allocation time. **(c)** Time to deploy code. **(d)** Deploy dependency time. **(e)** Scheduling time.

Figure 15. Cold start time and components by runtime for Region 2. 'all' represents all cold start times combined.

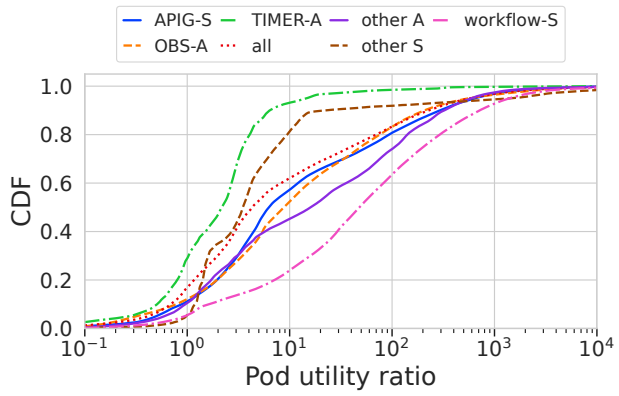


(a) Total cold start time. **(b)** Pod allocation time. **(c)** Time to deploy code. **(d)** Deploy dependency time. **(e)** Scheduling time.

Figure 16. Cold start time and components by trigger type for Region 2. 'all' represents all cold start times combined.



(a) Pod utility ratio by runtime.



(b) Pod utility ratio by trigger type.

Figure 17. Region 2 pod utility ratio.

reflects a greater utility to the function and the system. We hence call this the 'utility ratio'.

Figure 17a shows utility ratios per runtime, while Figure 17b shows utility ratios per trigger type. We see that 20% of all pods have a pod utility ratio less than 1, meaning that their useful lifetime is less than their cold start time. Median utility ratio is approximately 4:1, meaning that 50% of pods last for 4 times their cold start time. In Figure 17a, we see that a significant number of cold starts from *Node.js*, about 40%, had a ratio lower than 1. *Node.js* cold starts represent more than 10% of all the cold-starts in our system (see Figure 8e). *Node.js* is also the third slowest runtime in our system (see Figure 15). The runtimes with the second worst utility ratio are *PHP*7.3 and *Java*, both of which have at least 70% of their cold starts having a utility ratio lower than 10. The highest utility ratio runtime is *Go*1.*x* where 35% of pods have a utility ratio greater than 100. Notably, *Custom* and *HTTP* runtimes have a utility ratio better than several default runtimes, which have much shorter cold start times. Hence, pod utility ratio can be used to obtain a different perspective on the cost of cold starts, particularly for pods with long cold start times, to see how long those pods remain in the system.

Figure 17b shows that timers have the lowest utility ratios, which corresponds with most timer functions being cold started as discussed in previous sections. Timer functions, despite having some of the shortest cold start times, have a low pod utility ratio compared to other trigger types. Meanwhile, workflow-S, which has longer cold start times, tends to have higher utility ratios compared to other trigger types.

Utility ratios

Utility ratios help obtain a more complete picture of the cost of cold starts beyond cold start time by accounting for how long these pods remain useful in the system. We find that runtimes and trigger types with long cold start times tend to have higher pod utility ratios.

5 Discussion and open problems

Serverless systems research has gained momentum, with many projects aimed at reducing cold starts. Recently, Liu et al. [25] identified five research problems for future work on serverless systems. In this section, we leverage our analysis from previous sections to identify several areas important for serverless system optimization.

Cross-region workload scheduling. The average latency between data centers in developed regions is relatively small, in the orders of tens to a few hundred milliseconds [11, 44] with newer cross data center networking solutions expected to reduce this latency further [9]. In our analysis, the most popular regions consistently have much longer average, median, and tail cold-start times. While some of these differences can be attributed to differences in networking, our analysis shows significant differences in cold start time, CPU usage, and memory usage exist between regions, and less congested regions may offer cheaper and faster options for running workloads. It is therefore important for the research community to study cross-region serverless platforms and load-balancing. Cross-region scheduling could assume the form of a global or fleet-wide control plane accounting for peak time shifts, overall system load, latency from the user to the data center, and resources in different regions.

Scheduling delays are significant. Liu et al. [25] discussed how a large component of cold starts comes from scheduling delays. Our analysis shows that in our five data centers, except for a brief period in Region 1, scheduling delays are either the most or the second most significant component in cold start times. While one can think of optimizations to reduce these delays, it might be that a fundamental redesign of serverless platforms would be required. Currently, most serverless platforms run in a multi-layered stack, e.g. on top of Kubernetes, Ray, firecracker, KVM, or a combination of these systems. This adds multiple layers of complexity during a cold start, including setup, configurations, initialization, and scheduling. We believe that novel serverless platforms that reduce layered complexity would result in more optimized serverless performance.

Synchronous vs. asynchronous calls. In our workloads, roughly 60% of functions are asynchronous, in line with Liu et al. [25]. However, when studying the time series of running pods in Figure 8a and the bar chart in Figure 8d, we see that cold starts of functions with synchronous triggers sometimes dominate the total number of cold starts. This points to the fact that optimizations should consider both asynchronous and synchronous functions. Such optimizations can include peak shaving of asynchronous functions if these are not latency critical, starting them for example when there are much fewer synchronous functions.

Predicting cold starts. In Figure 6, we show that cold starts may be caused by functions with small or large peak-to-trough ratios. These tend to be low-request functions that are cold started every time they are invoked, such as those along the diagonal line in Figure 14, or high-request functions with larger peak-to-trough ratios. These two types of functions may require different solutions for predicting resource allocation needs, with the highly requested ones potentially benefiting from periodic time series predictions [18]. Functions running on timer triggers could be pre-warmed before their next invocation. Similarly, for functions running on timers less frequent than 1 minute, a keep alive time of 1 minute is unnecessary and wasteful. Cloud providers may consider a dynamic keep-alive time for such functions.

Workflow function calls can be predicted using previous function calls. As our analysis show, a significant number of cold starts occur due to synchronous workflow functions which can be predicted using function calls earlier in the chain. Resources for downstream functions could be allocated based on the invocations (and maybe even resource usage) of function calls that will invoke it later. This may alleviate the relatively slow cold start time of workflow-triggered functions. Currently, workflows account for 20% of cold starts. Additionally, the synchronous nature of these requests demands a strict latency SLO, which can be vastly improved with predictive autoscaling. Collection and analysis of function call chains may show opportunities for improvement.

Concurrency adjustment. Each function has a user-set concurrency value that determines how many function requests can be executed at the same time. For many functions, the resource utilization can be improved by increasing concurrency as long as the total execution time remains acceptable. This is especially useful given the strong oscillations in some asynchronous triggers, such as OBS.

Resource pool prediction. Our system maintains pools of inactive pods to be used as demanded by user traffic. If demand exceeds the capacity of the resource pool, a pod will be started from scratch, causing significant delays. However, keeping an unnecessarily large number of pods in the resource pool may be wasteful. Due to predictable time-varying patterns of various pod configurations, such as pods of different resource configurations, it may be possible to predict the required number of reserved pods so that user demand

is met without unnecessary overallocation. This differs from function invocation prediction [8, 18, 34], which requires additional steps from number of invocations to a scaling decision, and instead directly predicts required resources.

6 Related work

Analyzing and characterizing cloud and systems workloads has enabled the systems community to perform research on improving computer systems performance. Google released multiple traces from their internal Borg system [31, 36, 37, 43]. This data has been extensively analyzed by the research community over the years [17, 30, 33, 36]. Since then, there have been other data releases from multiple cloud providers for different internal systems, including Alibaba [12, 26, 41, 42], Azure [7, 13, 34], and Huawei [18]. Earlier research has shown that one of the main bottlenecks of serverless systems is cold starts, where the system does not have sufficient resources to process an incoming request and must start a new pod from scratch [18, 41]. There have been significant efforts to optimize cold starts, but few of these are informed by insights from production data.

Prior work with specific mention of cold start statistics [18, 34] tends to offer high-level metrics from a single region with little discussion of components and the effect of factors such as runtime language, resource allocation, and trigger type on the number of cold starts and their component times. Our work analyzes granular event-level metrics with detailed component times of cold starts from five regions, and examines the effect of function characteristics such as resource allocation, runtime language, and trigger type.

Cold starts have previously been found to be affected by function memory allocation, runtime language, and network latency [39]. Reducing cold starts using novel techniques to calculate the keep-alive time of a container is one active research topic [10]. Another interesting research direction is optimizing container deployments for serverless functions [29, 38]. Our work enables systems researchers to better understand some of the bottlenecks in production serverless systems including some of the root causes of cold starts.

7 Conclusion

This paper conducts an analysis of a multi-region production serverless cloud platform, focusing on factors affecting cold start times and their components. We have examined these factors in the context of long-term, evolving workloads as well as a week long holiday period in our month long dataset.

Our study reveals significant variations in function execution time, resource usage, and cold start time between regions, which may point to benefits of cross-region load balancing to reduce overall latency and cost. In all of our regions, cold start time is positively correlated with the number of cold starts. The dominant component of cold start time tends to vary between regions, which may point to workload differences or bottlenecks in different parts of the architecture. In all regions, the number of cold starts tends to decrease during the holiday period, with a 'catch-up' period afterwards where the number of cold starts and cold start time increase significantly. Additionally, cold start time for larger resource allocations tends to be longer than for smaller resource allocations, with pod allocation and deployment time for code and dependencies being contributors.

Furthermore, we have determined that significant numbers of cold starts come from several types of functions, such as low-request timer-triggered functions or high-request functions with large peak-to-trough ratios that require frequent autoscaling. We find that factors such as trigger type and runtime language affect the number and duration of cold starts differently. For example, pods using *Custom* runtimes experience significantly longer cold starts than those using default runtimes due to the absence of a reserved pool, with pod allocation time accounting for nearly the entire cold start duration. We introduce pod utility ratio, which can be used to measure a pod's usefulness by computing the ratio between a pod's useful lifetime (excluding keep-alive time) and its cold start time. We find that, in some cases, pod configurations with long cold start times tend to last longer, such as for *Custom* runtimes. Finally, we leverage our analysis to identify several bottlenecks in serverless public cloud platforms that contribute to cold starts and highlight opportunities for future research to address these issues and improve performance.

Acknowledgments

We would like to thank the YuanRong team at Huawei for their valuable collaboration and helping to collect the data. We would also like to thank Wei Wei for his feedback. Lastly, we would like to thank the reviewers at EuroSys for their insightful comments and Laiping Zhao for shepherding.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[2] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. 2023. Treehouse: A case for carbon-aware datacenter software. *ACM SIGENERGY Energy Informatics Review* 3, 3 (2023), 64–70.

[3] Noman Bashir, Tian Guo, Mohammad Hajiesmaili, David Irwin, Prashant Shenoy, Ramesh Sitaraman, Abel Souza, and Adam Wierman. 2021. Enabling sustainable clouds: The case for virtualizing the energy system. In *Proceedings of the ACM Symposium on Cloud Computing*. 350–358.

[4] Peter Bodik, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. 2010. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st ACM symposium on Cloud computing*. 241–252.

[5] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. 2022. Starlight: Fast container provisioning on the edge and over the WAN.

In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 35–50.

[6] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, et al. 2024. YuanRong: A production general-purpose serverless system for distributed applications in the cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 843–859.

[7] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[8] Luke Nicholas Darlow, Artjom Joosen, Martin Asenov, Qiwen Deng, Jianfeng Wang, and Adam Barker. 2023. FoldFormer: sequence folding and seasonal attention for fine-grained long-term FaaS forecasting. In *Proceedings of the 3rd Workshop on Machine Learning and Systems* (Rome, Italy) *(EuroMLSys '23)*. Association for Computing Machinery, New York, NY, USA, 71–77. https://doi.org/10.1145/3578356.3592582

[9] Vojislav Dukic, Ginni Khanna, Christos Gkantsidis, Thomas Karagiannis, Francesca Parmigiani, Ankit Singla, Mark Filer, Jeffrey L Cox, Anna Ptasznik, Nick Harland, et al. 2020. Beyond the mega-data center: Networking multi-data center regions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 765–781.

[10] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 386–400.

[11] Yantao Geng, Han Zhang, Xingang Shi, Jilong Wang, Xia Yin, Dongbiao He, and Yahui Li. 2023. Delay Based Congestion Control for Cross-Datacenter Networks. In *2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS)*. 1–4. https://doi.org/10.1109/IWQoS57198.2023.10188700

[12] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *Proceedings of the international symposium on quality of service*. 1–10.

[13] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, et al. 2020. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 845–861.

[14] Walid A Hanafy, Qianlin Liang, Noman Bashir, David Irwin, and Prashant Shenoy. 2023. CarbonScaler: Leveraging Cloud Workload Elasticity for Optimizing Carbon-Efficiency. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 3 (2023), 1–28.

[15] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 73–90.

[16] iguazio. 2024. Nuclio. https://nuclio.io/ Accessed: May, 2024.

[17] Akshay Jajoo, Y. Charlie Hu, Xiaojun Lin, and Nan Deng. 2022. A Case for Task Sampling based Learning for Cluster Job Scheduling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, USA. https://www.usenix.org/conference/nsdi22/presentation/jajoo

[18] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How Does It Function? Characterizing Long-term Trends in Production Serverless Workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*. ACM. https://doi.org/10.1145/3620678.3624783

[19] Knative Team. 2024. Knative: Concepts. https://knative.dev/docs/concepts/ Accessed: May, 2024.

[20] Ayşe Kızılersü, Markus Kreer, and Anthony W. Thomas. 2018. The Weibull Distribution. *Significance* 15, 2 (04 2018), 10–11. https://doi.org/10.1111/j.1740-9713.2018.01123.x arXiv:https://academic.oup.com/jrssig/article-pdf/15/2/10/49185942/sign_15_2_10.pdf

[21] Feifei Li. 2023. Modernization of databases in the cloud era: Building databases that run like Legos. *Proceedings of the VLDB Endowment* 16, 12 (2023), 4140–4151.

[22] Junfeng Li, Sameer G Kulkarni, KK Ramakrishnan, and Dan Li. 2021. Analyzing open-source serverless platforms: Characteristics and performance. *arXiv preprint arXiv:2106.03601* (2021).

[23] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 69–84.

[24] David H Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing more with less: orchestrating serverless applications without an orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1505–1519.

[25] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. 2023. The Gap Between Serverless Research and Real-world Systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (Santa Cruz, CA, USA)(SoCC'23). Association for Computing Machinery, New York, NY, USA*. 475–485.

[26] Qixiao Liu and Zhibin Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*. 347–360.

[27] Ashraf Mahgoub, Li Wang, Karthick Shankar, Yiming Zhang, Huangshi Tian, Subrata Mitra, Yuxing Peng, Hongqi Wang, Ana Klimovic, Haoran Yang, et al. 2021. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 285–301.

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

[29] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX annual technical conference (USENIX ATC 18)*. 57–70.

[30] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*. San Jose, CA, USA. http://www.pdl.cmu.edu/PDL-FTP/CloudComputing/googletrace-socc2012.pdf

[31] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2012. Obfuscatory obscanturism: making workload traces of commercially-sensitive systems safe to release. In *3rd International Workshop on Cloud Management (CLOUDMAN)*. IEEE, Maui, HI, USA, 1279–1286. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6212064

[32] 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 *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 231–246. https://doi.org/10.1145/3600006.3613155

[33] Stefano Sebastio, Kishor S. Trivedi, and Javier Alonso. 2018. Characterizing machines lifecycle in Google data centers. *Performance Evaluation* 126 (2018), 39 – 63. https://doi.org/10.1016/j.peva.2018.08.001

[34] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218.

[35] Jiuchen Shi, Kaihua Fu, Quan Chen, Changpeng Yang, Pengfei Huang, Mosong Zhou, Jieru Zhao, Chen Chen, and Minyi Guo. 2022. Characterizing and orchestrating VM reservation in geo-distributed clouds to improve the resource efficiency. In *Proceedings of the 13th Symposium on Cloud Computing*. 94–109.

[36] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the Next Generation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. ACM, Heraklion, Greece, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[37] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*. 1–17.

[38] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[39] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[40] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 497–517.

[41] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.

[42] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 995–1008.

[43] John Wilkes. 2020. *Google cluster-usage traces v3.* Technical Report. Google Inc., Mountain View, CA, USA. Posted at https://github.com/google/cluster-data/blob/master/ClusterData2019.md.

[44] Gaoxiong Zeng, Wei Bai, Ge Chen, Kai Chen, Dongsu Han, Yibo Zhu, and Lei Cui. 2022. Congestion control for cross-datacenter networks. *IEEE/ACM Transactions on Networking* 30, 5 (2022), 2074–2089.

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

[46] Siyuan Zhuang, Stephanie Wang, Eric Liang, Yi Cheng, and Ion Stoica. 2023. ExoFlow: A universal workflow system for Exactly-Once DAGs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 269–286.