Defuse: A Dependency-Guided Function Scheduler to Mitigate Cold Starts on FaaS Platforms

Jiacheng Shen*, Tianyi Yang*, Yuxin Su*, Yangfan Zhou†‡, and Michael R. Lyu*
*Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China.
Email: {jcshen, tyyang, yxsu, lyu}@cse.cuhk.edu.hk
†School of Computer Science, Fudan University, Shanghai, China. Email: zyf@fudan.edu.cn
‡Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China.

***Abstract*—Function-as-a-Service (FaaS) is becoming a prevalent paradigm in developing cloud applications. With FaaS, clients can develop applications as serverless functions, leaving the burden of resource management to cloud providers. However, FaaS platforms suffer from the performance degradation caused by the cold starts of serverless functions. Cold starts happen when serverless functions are invoked before they have been loaded into the memory. The problem is unavoidable because the memory in datacenters is typically too limited to hold all serverless functions simultaneously. The latency of cold function invocations will greatly degenerate the performance of FaaS platforms. Currently, FaaS platforms employ various scheduling methods to reduce the occurrences of cold starts. However, they do not consider the ubiquitous dependencies between serverless functions. Observing the potential of using dependencies to mitigate cold starts, we propose Defuse, a Dependency-guided Function Scheduler on FaaS platforms. Specifically, Defuse identifies two types of dependencies between serverless functions, i.e., strong dependencies and weak ones. It uses frequent pattern mining and positive point-wise mutual information to mine such dependencies respectively from function invocation histories. In this way, Defuse constructs a function dependency graph. The connected components (i.e., dependent functions) on the graph can be scheduled to diminish the occurrences of cold starts. We evaluate the effectiveness of Defuse by applying it to an industrial serverless dataset. The experimental results show that Defuse can reduce 22% of memory usage while having a 35% decrease in function cold-start rates compared with the state-of-the-art method.**

***Index Terms*—FaaS, Cold Start, Serverless, Cloud Computing, Service Dependency**

I. Introduction

Function-as-a-Service (FaaS) is a promising programming model for cloud computing [1]. Many cloud providers have launched their FaaS products, such as Google Cloud Functions, AWS Lambda, Azure Functions, and so on. Compared with traditional cloud products, the adoption of FaaS benefits both cloud providers and their clients. Specifically, by providing FaaS services, cloud providers can maximize the utilization of their infrastructures, i.e., host machines and VMs [2]. For example, computing resources can be allocated on-demand instead of idling and waiting to be used. Meanwhile, by using FaaS platforms, clients can focus exclusively on their application logic [3]. They only need to develop applications as serverless functions and leave all the labor-intensive management and administration tasks to cloud providers. Besides, clients' expenses can be saved since they only need to pay for what they actually use [4].

However, providing FaaS services to clients raises new challenges to cloud providers. By using FaaS, clients expect their serverless functions to be "always ready". As memory is one of the most limited resources in datacenters [5], there exists a trade-off between the readiness of serverless functions and the memory consumption. Typically, FaaS platforms put each serverless function into a container to execute them safely. When a serverless function is invoked, if a container with the function has been initialized in memory, the function can be executed instantly. However, if the container is not loaded, the latency of the function invocation will degenerate greatly because container initialization is time-consuming [6]. This is known as the cold-start problem on FaaS platforms. Since not all serverless functions are frequently invoked, keeping all these initialized containers loaded wastes the precious memory of cloud providers. Besides, clients of FaaS platforms are not billed on the resource consumption when their functions are not executed. Keeping idle functions loaded also wastes cloud providers' money [4]. The increased latency of cold starts will cause client losses and finally leads to economic losses.

Current methods on cold start mitigation can be classified into two categories. One is based on system-level optimizations [6]. These methods generally focus on reducing the time spent on executing a cold start. However, cold starts still lead to the increased latency which will affect user experiences. Another is to schedule serverless functions according to their invocation histories [4]. These methods focus on reducing the occurrences of cold starts. In this paper, by extending scheduling methods with higher accuracy, we first analyze the current scheduling methods and identify two issues. The first is the coarse granularity of scheduling. Current scheduling methods schedule all the serverless functions in an application as a whole. The idle functions, i.e, those that are loaded but not invoked, inevitably waste memory in FaaS platforms. Second, current scheduling methods cannot cope with applications without clear invocation patterns. The unpredictable invocation behaviors of these applications deteriorate the overall performance of FaaS platforms. Therefore, it is critical to attack these two issues, in order to improve the cold-start performance.

We suggest that the ubiquitous dependencies among serverless functions can be leveraged to improve current scheduling methods. The dependencies are the patterns when clients invoke serverless functions. In clients' perspective, serverless functions are APIs that can be invoked remotely [7]. Clients compose these APIs into complex applications. In this regard, there exists two types of dependencies, i.e., (1) some functions may be frequently invoked together and (2) some functions may be invoked by others. We can utilize the dependencies among serverless functions to guide the memory allocation when scheduling on FaaS platforms. Specifically, we can (1) schedule dependent functions as a whole, and (2) relate unpredictable functions to predictable ones. By scheduling dependent functions as a whole, cold starts can be reduced because these functions will naturally be invoked together. By relating unpredictable functions to predictable ones, the cold starts incurred by unpredictable functions can be diminished with the help of the predictable invocation patterns. Furthermore, scheduling in this finer granularity reduces the memory consumption.

However, discovering the dependencies among serverless functions is a non-trivial task. The dependencies are defined solely by the usage patterns of clients. FaaS platforms cannot acquire this information directly from the uploaded serverless functions. Observing the fact that clients exploit serverless functions by invoking them remotely, the usage patterns are recorded in the traces of function invocations. We conduct dependency mining by analyzing the invocation histories of serverless functions. Specifically, we divide the dependencies into strong and weak dependencies. The strong dependencies describe the relationships between predictable functions. We apply frequent pattern mining to discover it. The weak dependencies relate unpredictable functions to predictable ones. We exploit positive point-wise mutual information (PPMI) to reveal it from the data.

In this paper, we propose Defuse, a dependency-guided function scheduler on FaaS platforms. There are three steps in Defuse, i.e., dependency mining, dependency set generation, and scheduling. Dependency mining reveals the dependencies among serverless functions from their invocation histories. The output of it forms a function dependency graph. Then dependency sets are generated as connected components on the graph. Finally, Defuse schedules functions in the dependency sets with their invocation histories to decrease the occurrences of cold starts.

We evaluate Defuse with the Azure Public Dataset [8]. We measure the function cold-start rates and memory usages of different scheduling methods. The experimental results show that compared with the baseline methods, Defuse reduces 35% of function cold-start rate while saving 22% memory consumption.

The contribution of this paper is summarized as follows:

- To the best of our knowledge, this is the first work that utilizes the dependencies among serverless functions during the process of scheduling on FaaS platforms.
- We employ frequent pattern mining and PPMI to reveal dependencies from the data. We further propose Defuse, a dependency-guided scheduler, to mitigate the cold-start problem on FaaS platforms leveraging the dependencies among serverless functions.
- The experimental results show that compared with the baseline methods, Defuse reduces 35% of function cold-start rate while cutting 22% of memory usage.

This paper is organized as follows. Section II introduces basic concepts about FaaS platforms and the cold-start problem on them. In Section III we present the motivations of our work. Then in Section IV we elaborate the design and implementation of Defuse. We present our experimental study in Section V. We introduce some related works in Section VI and discusses about the limitations in Section VII. Finally, we conclude the paper in Section VIII.

II. Background

In this section, we provide background information on serverless computing, highlight the cold start problem in FaaS platforms, and introduce some basic solutions.

A. Serverless Computing and FaaS

In a FaaS platform, clients implement serverless functions instead of monolithic applications. These serverless functions appear as APIs that can be invoked with different triggers (e.g., events from the back-end, HTTP end-points) provided by FaaS platforms [9]. There are two main benefits FaaS provides to their clients. The first is that their time spent on environmental management is saved. Clients no longer need to manage virtual machines (VMs) or containers on their own. Besides, FaaS platforms take the responsibility of scaling clients' functions when a burst of requests emerges. By using FaaS, clients expect their serverless functions to be "always-ready" regardless of the load of concurrent invocations. Thus, one of the common Service Level Agreements (SLA) is the latency of function execution [10].

To meet the latency SLA, cloud providers should execute the serverless function as soon as the client's request arrives. Besides, FaaS platforms are responsible for container management, i.e., instance selection, scaling, deployment, and fault tolerance [1].

B. The Cold Start Problem in FaaS

As serverless function invocations exhibit unique characteristics such as burstiness, variable execution times, and statelessness [11], meeting clients' SLAs under such a dynamic workload is challenging to all FaaS platforms. Typically, FaaS platforms exploit sandbox mechanisms like containers to host the serverless functions [12]. When a serverless function is invoked, a platform normally takes three steps to deal with the request. First, it will choose an available worker and create an instance of the invoked function. An instance of the function is a container having all the required code loaded. Then it will execute the function in the container under restricted resources. Finally, it will return the execution results to the client according to the trigger type of the function. The whole process is known as the critical path of a client's request.

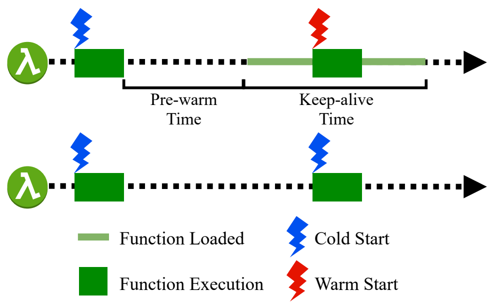
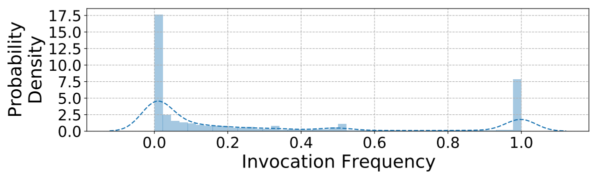


Fig. 1. The cold-start problem.

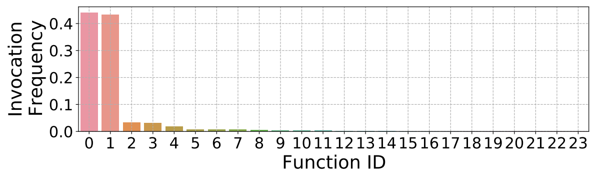
However, the existence of the critical path deteriorates the latency of cold starts in FaaS platforms. A cold start happens when a serverless function is requested without any instance of it in memory. The problem is unavoidable [13], [14] because the memory in datacenters is too scarce to keep all instances of serverless functions loaded. During a cold start, the whole critical path must be executed before clients can get their results. Since the time-consuming container initialization is on the critical path, the latency of cold function invocations will increase greatly. Meanwhile, the execution time of serverless functions is short and bursty [11]. The increased latency of cold function invocations will deviate from clients' expectations of their function execution time, which may lead to client losses and financial damage [15]. To maintain an "always-ready" illusion to clients, cloud providers have to (1) execute the critical path as fast as possible and (2) reduce the occurrences of function cold starts.

Currently, there are two types of methods to mitigate the problem of cold starts on FaaS platforms. One focuses on cutting the time spent on the critical path. The other targets at decreasing the number of cold starts. Here we briefly introduce the latter since Defuse also aims to reduce the occurrences of cold starts. The problem of serverless function scheduling can be reduced to deciding three parameters, i.e., the scheduling granularity, when to load into memory (pre-warm time), and how long should it be kept in memory (keep-alive time). Given the granularity of the scheduling, the decision of pre-warm and keep-alive time is crucial. As shown in Figure 1, cold starts can be reduced if functions can be loaded and kept in memory before their invocations.

However, the trade-off between memory usage and cold-start performance makes scheduling difficult. Employing an aggressive scheduling policy that uses a long keep-alive time will result in huge memory waste, while the reduced memory usage will inevitably lead to an increased cold-start rate. For example, the most aggressive scheduling policy is to set the pre-warm time to be 0 and the keep-alive time to be ∞. There will be no cold starts under this setting because all functions are loaded. But such a policy is impractical because it will waste the precious memory in FaaS platforms. An ideal scheduler should achieve an optimal balance between minimizing the occurrences of cold starts and keeping low memory consumption.



(a) Histogram of Function Invocation Frequency



(b) Invocation Frequencies of Functions in an Application

Fig. 2. Invocation frequencies of (a) all the serverless functions and (b) functions in a single application.

III. Motivation

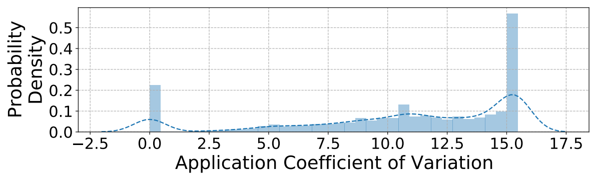
In this section, we will first show our observations of current scheduling methods by conducting data analysis on an industrial dataset [8]. Then we will demonstrate the opportunity of using dependencies of serverless functions to reduce the occurrences of cold starts in serverless platforms. Finally, we conclude this section with the challenges we need to solve.

A. *Observations of Current Methods*

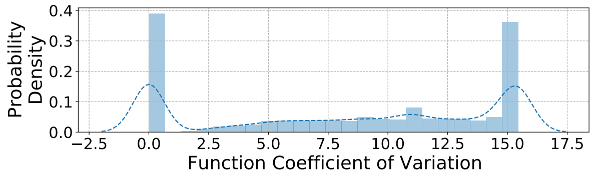
There are two main issues with existing scheduling methods.

1) Coarse Scheduling Granularity: The hybrid histogram scheduling method [4] schedules function invocations and allocates resources at the granularity of applications. An application is a set of serverless functions developed by clients to accomplish a complete business logic, e.g., a train ticket selling system [16]. The coarse scheduling granularity leads to two problems on FaaS platforms.

The first problem is the memory waste. Memory is wasted because not all serverless functions in an application are frequently invoked. Figure 2a shows the histogram of function invocation frequencies in the applications they belong to. 64.7% of functions have an invocation frequency less than 0.25. Figure 2b shows the histogram of function invocation frequency in a single application. In this specific application, only 2 out of 23 functions have invocation frequencies of more than 40%. Considering the amount of infrequent functions, the problem is universal on FaaS platforms. The skewed distribution of function invocation frequencies shows that memory will be wasted if a whole application is loaded into the memory during its cold start. This further implies that scheduling at the application level will waste a huge amount of memory on loading these infrequent functions. Having these wasted memories freed, FaaS platforms can employ more aggressive caching policies to further reduce the occurrences of cold starts.



(a) CV Histogram of Applications



(b) CV Histogram of Functions

Fig. 3. The histogram of coefficient of variations of IT histograms of (a) applications and (b) serverless functions.

The second problem is the increased cold-start overhead. The time and computation resources required to deal with a cold start is related to the number of functions to be loaded. When scheduling at the application level, the platform needs to load all the serverless functions in an application to deal with a cold start. However, as indicated in Figure 2a and Figure 2b, only a few functions in an application is needed. This means that resources and time will be wasted on loading functions that are unimportant to the cold start. Having the overhead of a cold start reduced, FaaS platforms can allocate more computing resources to deal with the potential burst of requests.

2) Unpredictable Functions/Applications: The second issue is the existence of unpredictable functions or applications, i.e., functions or applications that do not exhibit clear invocation patterns. Unpredictable applications/functions can be distinguished with the coefficient of variation (CV) of their binned idle time (IT), i.e., the time between two adjacent invocations. Functions/applications with small CVs are considered to be unpredictable as discussed in [4]. The problem of unpredictable functions is nonnegligible. Figure 3a shows the distribution of application CVs. There are 14% unpredictable applications with CV ≤ 5. The problem becomes even worse when it comes to the granularity of functions. Figure 3b shows the distribution of CV of function invocation interval histograms. There are 32% unpredictable functions with CV ≤ 5. The existence of unpredictable functions/applications incurs two problems to existing scheduling methods.

The first problem is that they incur a huge amount of cold starts. Normally, FaaS platforms apply a 10-minute fixed keep-alive policy to schedule these unpredictable applications/functions[1]. However, the invocation intervals of these applications/functions are likely to exceed the fixed keep-alive time [4]. This implies that the cold-start performance can be improved if these functions/applications can be properly coped with.

[1]Hybrid histogram scheduling method [4] also uses a 10-minute fixed keep-alive policy to schedule applications with small CVs, i.e., unpredictable applications.

The second problem is that the enormous number of unpredictable serverless functions makes it difficult to employ a fine-grained scheduling method. As shown in Figure 3b there are 32% of unpredictable functions with CV ≤ 5. The existence of these unpredictable functions leads to a poor result if current scheduling methods are directly applied at the function level. (See Section V).

To find a better solution to the cold-start problem, we need to figure out the following two questions:

- What caused the skewed distribution of function invocation frequencies?
- How to reduce the negative impact that unpredictable functions/applications cause on FaaS platforms?

B. Dependencies of Serverless Functions

We suggest that the ubiquitous dependencies among serverless functions can explain the skewed distribution of their invocation frequencies. To clients, serverless functions are APIs that jointly serve for a complex application. During the process of composition, serverless functions will exhibit some usage patterns, which can be viewed as dependencies among serverless functions. Specifically, taking serverless-trainticket [16], a train ticket selling system implemented as serverless functions on OpenFaaS [17], as an example, when a user books a ticket, the function `preserve-ticket` will invoke function `dispatch-seats` and function `create-order`. This simple case implies that whenever a user books a ticket, the three functions `preserve-ticket`, `dispatch-seats` and `create-order` will be invoked together. Further, the function `dispatch-seats` and function `create-order` are common services that will be jointly invoked by other functions, which means they will be frequently invoked together.

The dependencies among serverless functions can be exploited to reduce memory waste and decrease the occurrences of cold starts. Specifically, in the case of the serverless-trainticket, when a user books a ticket, we just need to load three functions `preserve-ticket`, `dispatch-seats`, and `create-order`. Compared with loading the whole application, memory can be saved by only loading those necessary functions. Generally, if a FaaS platform tracks all the dependencies of serverless functions, more memory can be saved by scheduling these dependency sets. The overhead of a cold start can also be reduced since fewer functions are required.

However, revealing dependencies among serverless functions is a non-trivial task for FaaS platforms. In special settings like AWS step functions, FaaS platforms can acquire the dependencies by requiring clients to provide them. In most cases, the dependencies cannot be explicitly acquired because they are defined solely by the usage patterns of clients. Finding that the usage patterns of functions are recorded in their invocation histories, we decide to reveal the dependencies from this data.

In summary, the challenges in leveraging the dependencies to schedule serverless functions are listed as follows.

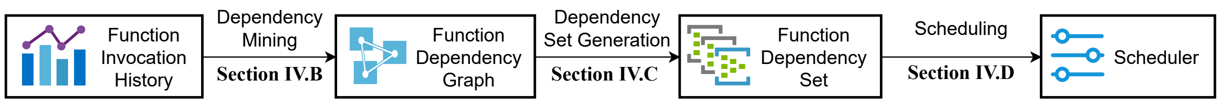


Fig. 4. Overview of Defuse

- How to reveal the dependencies among serverless functions?
- How to cope with unpredictable functions?
- How to have comparable cold start performance while having less memory usage compared with coarse-grained scheduling methods?

IV. Methodology

In this section, we will introduce how Defuse solves the above challenges. We will first demonstrate the workflow of Defuse. Then we will present each step in detail, i.e., dependency mining, dependency set generation, and scheduling.

A. Overview

Figure 4 shows an overview of Defuse. There are three steps, dependency mining, dependency set generation, and scheduling. First, Defuse takes the invocation histories of serverless functions as the input and conducts dependency mining on these invocation records. The dependencies are divided into two categories, i.e. strong dependencies and weak dependencies. A function dependency graph is constructed based on the mined dependencies. Then we generate dependency sets of serverless functions according to the graph. All these dependency sets are the input of the dependency-guided scheduling policy, in which all serverless functions in a dependency set are scheduled as a whole. FaaS platforms can decide when to load a dependency set and how long to keep it in the memory based on the scheduling policy.

B. Dependency Mining

The first challenge to solve is how to discover the dependencies among serverless functions. We will start by defining the dependency we want to find and the intuition behind it. Then we will present how to acquire the dependencies from the invocation histories of serverless functions and how to generate dependency sets.

1) Definition of the Dependency: Two aspects need to be considered when defining the dependencies among serverless functions. The first is how to precisely describe the dependencies demonstrated in Section III. There are two properties of the dependencies. First, dependent functions are likely to be invoked together. Second, dependencies only exist among serverless functions of the same client because the dependencies result from the usage pattern of the clients. Typically, clients only have access to their own serverless functions. It would be meaningless to define dependencies across clients.

The second aspect is the cold starts incurred by the unpredictable functions. If these ubiquitous unpredictable functions cannot be properly dealt with, the latency of FaaS platforms will degenerate greatly. The dependencies between predictable and unpredictable functions could be leveraged to solve the problem. Here is a motivating example. In serverless-trainticket, users may book tickets at any time. This will lead to the unpredictable invocation of the function `preserve-ticket`. During the execution of `preserve-ticket`, it will invoke `dispatch-seat`, which is a common service that is invoked frequently and is predictable. The above dependencies help us relate unpredictable functions with predictable ones. The unpredictable functions could be scheduled according to the invocation patterns of predictable ones, which will reduce the cold starts.

Therefore, we define the strong dependencies and weak dependencies among serverless functions.

- Strong Dependency: Function f_a and function f_b have strong dependency iff. 1) they belong to the same client and 2) there is high probability of them being simultaneously invoked in a small time window. It is a bi-directional relationship ($f_a \leftrightarrow f_b$).
- Weak Dependency: Function f_a have weak dependency on function f_b iff. 1) they belong to the same client and 2) there is high probability that f_a is invoked under the condition that f_b is invoked. It is a single-directional relationship ($f_a \rightarrow f_b$).

Both strong and weak dependencies should satisfy two conditions, i.e., (1) the ownership condition, and (2) the probability condition. The strong dependencies describe the relationship between globally frequently invoked functions, which are likely to be predictable. The weak dependencies describe the relationship between unpredictable and predictable functions.

2) Strong Dependency Mining: The purpose of strong dependency mining is to find the relationships among functions that are frequent and predictable. We need to find combinations of a client's functions that have a high probability of being invoked together. Frequent pattern mining [18] naturally fits this requirement. Given a set of transactions, frequent pattern mining can find all the itemsets with frequency greater than a given threshold. Hence, Defuse adopts frequent pattern mining to uncover the strong dependencies among serverless functions.

Specifically, for each client, the invocation records of all her functions can be represented as a set $R = \{r_i | i = 1, 2, \ldots, m\}$, where m is the number of functions of the client, $r_i = (t_1, t_2, \ldots, t_n)$ is the invocation records of function f_i, and t_j is the timestamp of its j^{th} invocation. Defuse first divides the period where the records are sampled into small non-overlapping time windows and counts the number of invocations of each function in the time window. Then we get the invocation matrix I for the client, where $I_{i,j}$ is the number of invocations of function i in time window j. After that, for each column of I, Defuse gathers all the functions with non-zero invocation count into a single transaction and gets all the transactions of the client. Finally, Defuse employs FP-Growth [19] to conduct frequent pattern mining on the generated transactions. The outputs of frequent pattern mining are frequent itemsets. All functions in a frequent itemset have a high probability of being invoked in the given small time window, which satisfies the probability condition. Additionally, since Defuse conducts frequent pattern mining only on functions that belong to the same client, the ownership condition is satisfied as well. The strong dependencies of all serverless functions in FaaS platforms can be retrieved by repeating the above steps to each client on the platform.

3) Weak Dependency Mining: The goal of weak dependency mining is to find the dependencies between unpredictable and predictable functions. Since the coefficient of variations (CV) of idle time (IT) histograms of unpredictable functions are small. Defuse distinguishes unpredictable functions with predictable ones with the CV of function IT histograms. Particularly, Defuse generates the IT histogram of each function from its corresponding vector in the invocation matrix I. Then Defuse calculates the CV for each function and discriminates them by a threshold.

Defuse mines weak dependencies by positive point-wise mutual information (PPMI) [20]. Suppose the possibilities of an unpredictable function f_u and a predictable function f_p being invoked individually are P_u and P_p, respectively. Let $P_{u,p}$ indicate the probability of them being invoked together. The PPMI of the invocation of f_u and f_p can be represented as:

$$PPMI(f_u, f_p) = max(0, PMI(f_u, f_p))$$

Where $PMI(f_u, f_p)$ is the point-wise mutual information (PMI) [21] between f_u and f_p. It can be represented as:

$$PMI(f_u, f_p) = log_2 \frac{P_{u,p}}{P_u \cdot P_p}$$

Intuitively, if f_u and f_p are dependent, the probability of them being invoked together will be higher than they are each invoked independently. As a result, $P_{u,p}$ will be greater than $P_u \cdot P_p$, which means PMI will be positive. The higher the $PMI(f_u, f_p)$ the stronger the dependency. Since $PPMI(f_u, f_p)$ is the maximum of $PMI(f_u, f_p)$ and 0, it is also positively related to the degree of dependency.

To get PPMIs, Defuse first constructs a co-occurrence matrix C based on the function invocation matrix of predictable and unpredictable functions. Each row of C represents an unpredictable function and each column of C represents a predictable one. $C_{i,j}$ represents the number of co-invocations of function f_i and f_j in a small time window. Then we estimate the probability by invocation frequencies and calculate $PPMI$ based on C. For each unpredictable function f_{u_i}, a vector $v_{u_i} = (PPMI(f_{u_i}, f_{p_1}), \ldots, PPMI(f_{u_i}, f_{p_w}))$ is generated by Defuse. After sorting each vector in descending order, Defuse assigns the top k predictable functions to be weakly dependent on the unpredictable function f_{u_i}, where k can be defined by users.

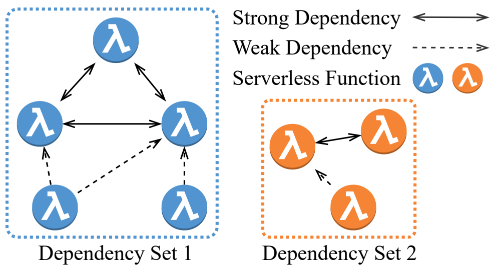


Fig. 5. The Dependency Graph and Dependency Sets of Serverless Functions

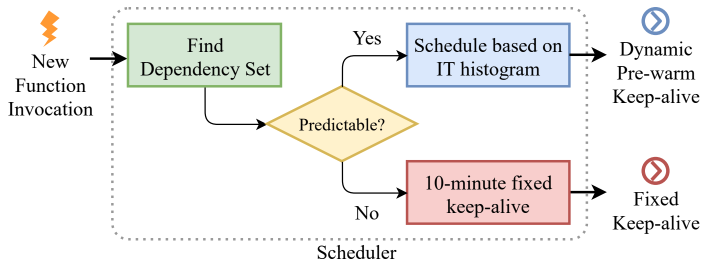
C. Dependency Set Generation

The second step of Defuse is to generate dependency sets based on the mined dependencies. As mentioned in Section II a scheduler needs to decide the granularity of scheduling and how long a function should stay in memory. However, the mined dependencies are relationships among serverless functions, which cannot be directly exploited. To facilitate the scheduling step, Defuse conducts dependency set generation to convert the relationships into function sets. First, Defuse constructs a function dependency graph as shown in Figure 5. Each vertex in the graph is a serverless function and each edge represents either strong or weak dependency. The dependency sets are defined as connected components on the graph. Then Defuse uses union-find to extract all these connected components and group them as dependency sets. Implied by the definition of dependencies, functions connected with each other have a high probability of being jointly invoked. Scheduling functions in a dependency set together reduces the occurrences of cold starts.

D. Scheduling

The last step of Defuse is to schedule serverless functions based on the generated dependency sets. As discussed in Section II, for each dependency set, the scheduler needs to decide 1) when to pre-warm it by loading it into the memory (pre-warm time) and 2) how long to keep it in memory after it is invoked (keep-alive time). As the IT histogram is proved to be effective in scheduling functions in [4], we adopt the same policy to determine the pre-warm time and keep-alive time of each dependency set. Figure 6b illustrates the cumulative distribution function (CDF) of an IT histogram of a dependency set. We set the 5^{th} percentile of the IT histogram as the pre-warm time and load the set of functions into the memory this period after its invocation. Then we set the time between the 5^{th} and 95^{th} percentile as the keep-alive time, which means a dependency set will be reserved in the memory without being invoked for this period.

Besides, there exist some dependency sets without clear invocation patterns. We distinguish dependency sets as predictable sets and unpredictable sets based on their CVs. The



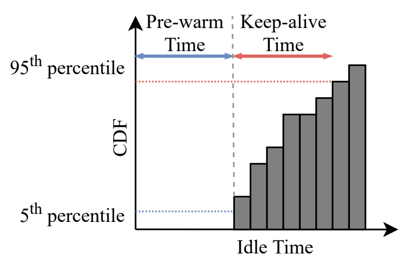


Fig. 6. The Scheduling Policy. (a) The scheduling procedure when a new invocation is triggered. (b) Deciding keep-alive time and pre-warm time by the CDF of idle time between function invocations.

scheduling policy is shown in Figure 6a. For predictable sets, Defuse decides the pre-warm time and keep-alive time based on their IT histograms. For unpredictable sets, Defuse steps back to a fixed-timeout policy. Since the granularity of scheduling is finer, Defuse can employ more aggressive timeout settings and further reduce the negative effect brought about by these unpredictable sets. In addition, it is possible to conduct scheduling using different policies and we will discuss this in Section VII.

V. Experiments

In this section, we focus on the following research questions to evaluate Defuse.

- RQ1: How effective is Defuse compared with other scheduling methods?
- RQ2: What is the overhead of Defuse on FaaS platforms compared with other scheduling methods?
- RQ3: What is the performance of Defuse under different memory usage?
- RQ4: What are the contributions of weak and strong dependency mining to Defuse?

A. *Experiment Settings*

To evaluate the cold-start performance of Defuse, we conduct simulations on the Azure Public Dataset [8]. The dataset released by [4] contains a 14-day function invocation record on the Microsoft Azure Functions with 83,137 functions, 24,964 applications, and 15,097 users. For each serverless function, the dataset records its invocation times in the granularity of minutes. We mine dependencies on the invocation data of the first 12 days and conduct simulation with the data in the last 2 days. All the simulations are conducted on a workstation with two 12-core Intel Xeon E5-2620 CPUs and 128 GB memory. Since the traces of functions are recorded minute-wise, we set the time window to be 1 minute for simplicity. For strong dependency, we conduct frequent pattern mining with pyfpgrowth[2]. Since the memory consumption of FP-Growth explodes when the length of transactions exceeds a limit, we shuffle the functions of a user and split them into small windows of functions. In order to reserve dependencies across windows, the window size is set to be 20 and the stride is set to be 10. There are two parameters in the dependency mining step, i.e. the support θ of frequent pattern mining and the top-k in the weak dependency mining. To set proper parameters, we conduct line-search on both θ and k and discover that Defuse performs best when the support is set to be 0.2 and the top-k is set to be top-1.

We choose two baseline methods, namely Hybrid-Application and Hybrid-Function. Hybrid-Application adopts the hybrid histogram scheduling policy [4] at the application level. We reproduce the method strictly following the descriptions in the paper. Hybrid-Function simply employs the hybrid histogram scheduling policy at the function level. We implement the method by directly setting the granularity of scheduling to be the function level. The reproduced scheduling policy has a 0.25 application cold-start rate during the simulation on the traces of the last 2 days, which is about the same as the result described in the original paper [4]. There are two main sources of the little difference. The first is the randomness in the ARIMA model and the second is the differences in the dataset[3].

There are three parameters in the hybrid histogram scheduling policy, i.e., cv_{thresh}, mem_{thresh}, and $hist_{thresh}$. cv_{thresh} distinguishes predictable functions from unpredictable ones. mem_{thresh} controls the keep-alive time of unpredictable functions. It decides how long an unpredictable serverless function will stay in the memory after its former invocation. Finally, $hist_{thresh}$ controls the pre-warm time and the keep-alive time of predictable functions according to its IT histogram. All these three parameters will affect the memory usage. For all the baseline methods, we set $mem_{thresh} = 10$ minutes and $hist_{thresh} = 0.05$ following the previous work [4]. We set $cv_{thresh} = 5$ which performs better than the default setting ($cv_{thresh} = 2$) as reported in [4].

B. *Evaluation Metrics*

We adopt the following three metrics to evaluate the memory usage, cold-start performance, and the cost of cold starts of different scheduling policies. First, limited by the dataset we approximate the memory usage of different methods with the

[2]https://github.com/evandempsey/fp-growth

[3]The released dataset is a small fraction of the dataset used in [4]

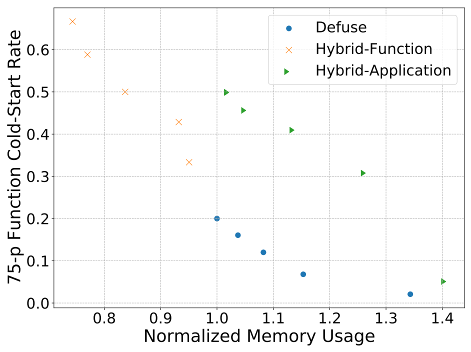


Fig. 7. 75th percentile function cold-start rate under different memory usage

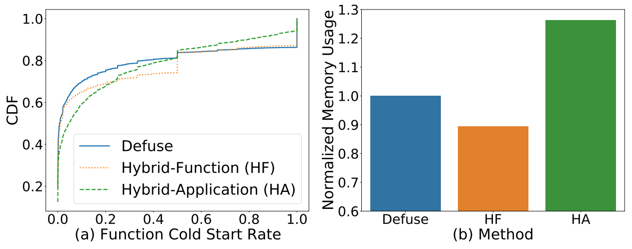


Fig. 8. CDF of function cold start rate of Defuse, Hybrid-Function, and Hybrid-Application

number of functions loaded in memory. The dataset doesn't provide the run-time memory consumption of each serverless function. Considering the fact that serverless functions usually perform simple tasks, the memory consumption of them will be close. We resort to using the number of functions loaded in memory to evaluate the memory usage. Then, similar to [4], we quantify the cold-start performance of different scheduling methods by the function cold-start rate. For Defuse, we calculate the cold-start rates of each function as the cold-start rate of its dependency set. Similarly, for the Hybrid-Application method, the cold-start rate of functions are computed as the cold-start rate of their applications. As for the Hybrid-Function, we simply calculate the cold-start rate of each serverless function during the simulation. Intuitively, the lower the cold-start rate, the fewer functions will encounter high-latency cold starts when they are invoked. This means that the latency of function invocations will also be reduced. Finally, we measure the overhead of scheduling methods by the number of loading functions. The process of loading serverless functions requires FaaS platforms to execute the critical path, which consumes the computation resources of host machines. The fewer the functions being loaded, the more free computation resources can be used to serve clients' requests. Therefore, it is reasonable to evaluate the overhead that a scheduling method will add on FaaS platforms with the number of loading functions.

C. *RQ1: Effectiveness of* Defuse

Figure 7 shows the 75^{th} percentile function cold-start rate of three scheduling methods under different memory usages. To precisely control the memory usage of each method, we add an additional parameter, amplification factor a. Specifically, for each method, we increase or reduce its memory usage by multiplying a to the calculated keep-alive time. Each point on the figure represents the result of a single simulation. The x-axis of the figure means the average memory usage of the experiment. Without further noticing, all the memory usages are normalized by the minimum memory usage of Defuse. The y-axis of the figure means the 75^{th} percentile function cold-start rate. A point (x, y) on the graph indicates that under the memory usage of x, there are 75% serverless functions that have cold-start rates less than y. Hence, the closer a point is to $(0, 0)$, the better the performance of the scheduling method. The Hybrid-Function scheduling method has the least average memory usage, but it has the highest cold-start rate. This is caused by the huge number of unpredictable functions that it cannot properly deal with. On the other hand, Hybrid-Application has more memory usage and lower cold-start rate. However, its cold-start rate will increase greatly when its memory usage is restricted. Compared with two baseline methods, Defuse exhibits the best cold-start performance. Compared with the Hybrid-Application method, the memory usage of Defuse is reduced. However, the memory consumption of Defuse exceeds that of the Hybrid-Function method. This happens because Hybrid-Function schedules at a finer granularity. But the improvement in function cold-start rate outweighs the increase in the memory usage.

Figure 8a shows the cumulative distribution (CDF) of the cold-start rates of serverless functions under different scheduling methods. Figure 8b shows the corresponding memory consumption. We restrict the memory consumption for the fairness of comparison. Defuse has more functions that have a lower cold start rate compared with the other two. Especially, compared with the Hybrid-Application scheduling method, Defuse has a 20% reduction in memory usage while having a 35% decrease in the 75^{th} percentile function cold-start rate. However, the Hybrid-Application method has fewer functions that have high cold-start rates compared with the other two fine-grained methods. The inferior performances of finer-grained scheduling methods are caused by the infrequent unpredictable serverless functions. The same problem also exists in the Hybrid-Application scheduling method, but the negative impact is weaker since these functions are grouped together as applications. The increasing memory usage of the Hybrid-Application contributes to the decrease of cold-start rates in these functions. The problem can be mitigated with a different scheduling policy. For example, by conducting time-series predictions to schedule functions more accurately. We leave this to our future work and will discuss it in Section VII.

D. *RQ2: The Overhead of* Defuse

To evaluate the overhead of Defuse on FaaS platforms, we measure the number of loading functions in every minute

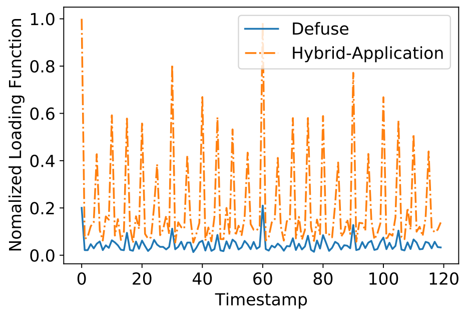


Fig. 9. Normalized number of loading functions in 2 hours. (We don't consider Hybrid-Function here because it sacrifices cold-start rate to have lower memory consumption and load one function at a time.)

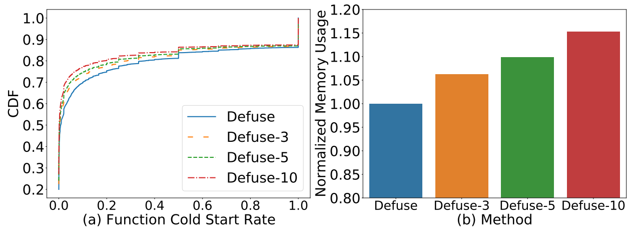


Fig. 10. CDF of function cold start rate under different memory usage.

during the simulation. The number of loading functions reflects the computation resources that FaaS platforms need to spend on the scheduling. We first calculate the average loading functions in every minute during the simulation. Compared with the Hybrid-Application method, Defuse reduces the average number of loading functions by 79%. Figure 9 shows the time series of the normalized number of loading functions in two hours. The number of loading functions is normalized by the maximum number of loading functions of Hybrid-Application in the period. We can also find that the cost of cold starts is reduced sharply by using Defuse. The reduction in the number of loading functions owes to the finer granularity of scheduling that Defuse employs. This implies that the dependency of serverless functions can be effectively used to reduce the pressure on FaaS platforms.

E. RQ3: Performance under Different Memory Consumption

To test the stability of Defuse, we measure the function cold-start rate under different memory consumption. Figure 10a shows the CDF of the function cold-start rate of Defuse under different memory consumption. Similarly, we control the memory usage by changing the amplifying factor a to be 1, 3, 5, and 10, respectively. Figure 10b shows the normalized memory consumption. The results show that more functions have smaller cold-start rates when memory usage increases, which confirms the trade-off between memory consumption and function cold-start rate. This further shows that cloud providers can simply balance the memory consumption and the function cold-start rates by choosing a proper parameter that controls memory usage.

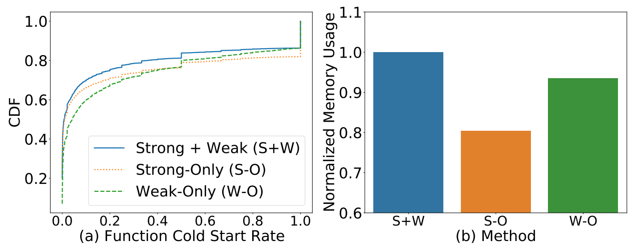


Fig. 11. Left: CDF of function cold-start rate using (1) both strong and weak dependency, (2) strong dependency only, and (3) weak dependency only. Right: Their corresponding memory consumption

F. RQ4: Contribution of Strong and Weak Dependency Mining

We also conduct an ablation study on Defuse. Particularly, we focus on evaluating the contribution of strong and weak dependency mining. We compare the cold-start performance and memory consumption using (1) strong dependency mining only (Strong-Only), (2) weak dependency mining only (Weak-Only), and (3) both (Strong + Weak). Figure 11a shows the CDF of function cold-start rate of the three approaches. Figure 11b shows the corresponding memory consumption. Compared with Weak-Only, more functions exhibit lower cold-start rates in Strong-Only, but there are more functions with high cold-start rates. This is because the strong dependencies only detect functions that are globally invoked together leaving lots of unpredictable functions. However, the weak dependencies can link these unpredictable functions to predictable ones. Linking predictable functions with unpredictable functions and scheduling them as a whole effectively reduces the cold-start rates of these unpredictable functions. Combing both strong and weak dependency mining produces the lowest function cold-start rate. Nevertheless, the memory consumption of Strong+Weak is also the highest. This happens because combining the two dependencies generates bigger connected components in the dependency graph, which results in bigger dependency sets. Combining with the results in Figure 7, Defuse utilizes the memory more efficiently compared with the Hybrid-Application scheduling method. The efficiency comes from the fact that combining strong and weak dependencies can guide memory usage.

VI. Related Work

Cold Start Mitigation. Cold start mitigation [22] on FaaS platforms is an emerging research problem. Approaches for cold-start mitigation can be divided into two categories, i.e., reducing the time spent on a cold start and reducing the occurrences of cold starts.

There are three ways to reduce the time spent on a cold start. The first one is to speedup container initialization. SOCK [6], and HotC [23] leverage lightweight containers to reduce the latency of creating a new container, the memory footprint, and the latency of network of containers. Akkus et al. [24] introduce application-level sandboxing and a hierarchical message bus for acceleration. SEUSS [25] leverages the advantage of unikernels to speedup function invocations as well as cache more function in memory. The second is to reduce the runtime initialization latency. Mohan et al. [2] propose to use pre-created networks to reduce the time spent in network initialization. Daw et al. [26] develop Xanadu to mitigate the cascading latency overheads in triggering a sequence of serverless functions according to a workflow specification. The third way is to reuse containers. Silva et al. [27] propose to restore snapshots from previously executed function processes to reduce cold starts. Ustiugov et al. [28] propose REAP, a record-and-prefetch mechanism to speedup the function invocation.

Many approaches have been proposed to reduce the occurrences of cold starts. OpenWhisk [29] employs pre-warming to launch containers before their invocations. AWS Lambda employs a fixed-time "keep-alive" policy to keep resources in memory after function executions [13]. ENSURE [30] prevents cold-starts incurred by different workloads leveraging the concepts of operations research and the characteristics of different functions. To overcome the lack of flexibility of fixed-time "keep-alive" policy, Thundra [31] periodically makes warm-up calls to serverless functions. The closest related work to ours is [4]. It focuses on deciding when and how long to keep a function execution environment alive. It characterizes the industrial FaaS workload in a large cloud service provider, and then uses the histogram of the historical idle time between function invocations to select the pre-warm and keep-alive times. However, Defuse leverages the dependencies among serverless functions to schedule them directly, which is a finer granularity. It adds another dimension to the problem of cold-start mitigation, and thus is complimentary to all existing methods that focus on the former two aspects.

Scheduling. In FaaS platforms, scheduling is to allocate resources (memory, CPU, container, etc.) to handle incoming function invocations. Although scheduling methods have been studied for decades, most existing methods fall short in catering to the need of FaaS platforms, i.e., scheduling at a scale of millions of function invocations per second while achieving predictable performance. Suresh and Gandhi [10] propose FnSched, the first work that focuses on scheduling user function requests from single and multiple invokers. FnSched first categorizing functions into different categories and then place them accordingly. Kaffes et al. [11] improve performance by explicitly managing the sharing of individual cores among simultaneously executing functions. They proposed a centralized, cluster-level scheduler that operates at core-granularity and assigns functions directly to individual cores. Nguyen et al. [32] extends the use case of FaaS to real-time scenarios and implemented a prototype for it. Different from the above works, Defuse focuses on scheduling on function level instead of the lower-level resource allocations.

Dependency Mining. While mining the dependencies among distributed services has been studied for years, to the best of our knowledge, we have not found any prior work specific to mining the dependencies of serverless functions. Nandi et al. [33] mine a service dependency graph from runtime logs for anomaly detection. Yin et al. [34] cluster services based on the key performance indicators such as CPU utilization, memory utilization, and disk I/O.

VII. Discussion and Future Work

Dependency Mining. Restricted by the dataset, in this work, we can only reveal dependencies from the function invocation histories. That's why we apply frequent pattern mining and PPMI to discover dependent serverless functions. If provided with sufficient data, the dependencies can also be extracted with other data mining or machine learning methods. For example, natural language processing techniques may be applicable to improve dependency mining if the names and contents of these serverless functions are provided.

Scheduling Policies. In this work, we focus on improving the granularity of scheduling with the dependencies among serverless functions. The scheduling policy we employ is similar to the hybrid histogram method proposed in [4]. In fact, our method is compatible with arbitrary scheduling policies. For example, time-series prediction methods can be applied to predict when a function will be invoked. By using a more sophisticated scheduling policy, the memory usage can be further reduced while the cold start rate can be further decreased. Developing accurate scheduling policies based on the function invocation histories is also worth exploring.

Adaptive Scheduling. In the evaluation part, we conduct dependency mining on the data of the first 12 days and conduct simulation with the data of the last 2 days. In fact, Defuse is an adaptive scheduling method. The dependency mining module can be implemented as a daemon process and update the function dependency graph periodically, i.e., every day. It takes 15 minutes to generate all the dependencies in a one-day trace with 50,334 distinct functions with the same machine in Section V. Dependency sets can be updated with the latest dependency graph. The scheduler can be implemented to schedule serverless functions with the updated dependency sets.

Resource Consumption. Another important factor is the resource consumption of the scheduler. During the scheduling process, the scheduler only needs to find the dependency set of the invoked function and update its histogram. The requirements for computation resources are low. On the other hand, the scheduler needs to store all the dependency sets and their corresponding histograms. Since the histograms are of fixed length, the demand for memory resources is also low. In addition, the scheduler can be implemented in a stand-alone service to eliminate the interference with client functions.

VIII. Conclusion

The cold start of serverless functions is a severe problem on FaaS platforms. In this work, we propose Defuse, a dependency-guided scheduler for serverless functions, to mitigate the problem. Unlike the current coarse-grained scheduling methods, Defuse takes the dependencies among serverless functions into consideration in the scheduling process. The utilization of dependencies makes it possible to schedule functions in a finer granularity while reducing the cold-start rates. Besides, Defuse is complementary to existing cold-start mitigation approaches and is compatible with arbitrary scheduling policies as well as system improvements that reduce the cold-start time. Experiments show that Defuse can effectively reduce memory consumption while having smaller function cold-start rates compared with the baseline methods.

ACKNOWLEDGMENTS

The work was supported by Key-Area Research and Development Program of Guangdong Province (No. 2020B010165002) and the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 14210717). Yangfan Zhou is the corresponding author.

REFERENCES

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," EECS Department, UC Berkeley, Tech. Rep., 2019.

[2] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Jul. 2019.

[3] R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. Netto *et al.*, "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM computing surveys (CSUR)*, vol. 51, no. 5, pp. 1–38, 2018.

[4] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, 2020, pp. 205–218.

[5] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "AIFM: High-performance, application-integrated far memory," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020, pp. 315–332.

[6] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018, pp. 57–70.

[7] "Azure functions documentation." [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/

[8] "Azure public dataset." [Online]. Available: https://github.com/Azure/AzurePublicDataset

[9] "Aws lambda." [Online]. Available: https://aws.amazon.com/lambda/

[10] A. Suresh and A. Gandhi, "Fnsched: An efficient scheduler for serverless functions," in *Proceedings of the 5th International Workshop on Serverless Computing, WOSC@Middleware 2019, Davis, CA, USA, December 09-13, 2019*. ACM, 2019, pp. 19–24.

[11] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 2019, pp. 158–164.

[12] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, H. S. Gunawi and B. Reed, Eds. USENIX Association, 2018, pp. 133–146.

[13] M. Shilkov, "Cold starts in aws lambda." [Online]. Available: https://mikhail.io/serverless/coldstarts/aws/

[14] M. Shilkov, "Cold starts in azure functions." [Online]. Available: https://mikhail.io/serverless/coldstarts/azure

[15] B. L. R. Erwan Alliaume, "Cold start / warm start with aws lambda." [Online]. Available: https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/

[16] "Serverless trainticket." [Online]. Available: https://github.com/FudanSELab/serverless-trainticket

[17] "Openfaas." [Online]. Available: https://github.com/openfaas/faas

[18] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993, pp. 207–216.

[19] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data mining and knowledge discovery*, vol. 8, no. 1, pp. 53–87, 2004.

[20] Y. Niwa and Y. Nitta, "Co-occurrence vectors from corpora vs. distance vectors from dictionaries," *arXiv preprint cmp-lg/9503025*, 1995.

[21] K. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Computational linguistics*, vol. 16, no. 1, pp. 22–29, 1990.

[22] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems, COINS 2020, Barcelona, Spain, August 31 - September 2, 2020*. IEEE, 2020, pp. 1–7.

[23] K. Suo, Y. Shi, X. Xu, D. Cheng, and W. Chen, "Tackling cold start in serverless computing with container runtime reusing," in *Proceedings of the 2020 Workshop on Network Application Integration/CoDesign, NAI@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*. ACM, 2020, pp. 54–55.

[24] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018, pp. 923–935.

[25] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "Seuss: Skip redundant paths to make serverless fast," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3392698

[26] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, D. D. Silva and R. Kapitza, Eds. ACM, 2020, pp. 356–370.

[27] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*, D. D. Silva and R. Kapitza, Eds. ACM, 2020, pp. 1–13.

[28] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," *arXiv preprint arXiv:2101.09355*, 2021.

[29] "Squeezing the milliseconds: How to make serverless platforms blazing fast!" [Online]. Available: https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing\-fast-aea0e9951bd0

[30] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 1–10.

[31] E. Şamdan, "Dealing with cold starts in aws lambda." [Online]. Available: https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532

[32] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien, "Real-time serverless: Enabling application performance guarantees," ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–6. [Online]. Available: https://doi.org/10.1145/3366623.3368133

[33] A. Nandi, A. Mandal, S. Atreja, G. B. Dasgupta, and S. Bhattacharya, "Anomaly detection using program control flow graph mining from execution logs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. ACM, 2016, pp. 215–224.

[34] J. Yin, X. Zhao, Y. Tang, C. Zhi, Z. Chen, and Z. Wu, "Cloudscout: A non-intrusive approach to service dependency discovery," *IEEE Trans. Parallel Distributed Syst.*, vol. 28, no. 5, pp. 1271–1284, 2017.