Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud

Huangshi Tian
HKUST

Suyi Li
HKUST

Ao Wang
George Mason University, Alibaba Group

Wei Wang
HKUST

Tianlong Wu
Alibaba Group

Haoran Yang
Alibaba Group

ABSTRACT

This work documents our experience of improving the scheduler in Alibaba Function Compute, a public FaaS platform. It commences with our observation that memory and CPU are under-utilized in most FaaS sandboxes. A natural solution is to overcommit VM resources when allocating sandboxes, whereas the ensuing contention may cause performance degradation and compromise user experience. To complicate matters, the degradation in FaaS can arise from external factors, such as failed dependencies of user functions.

We design Owl to achieve both high utilization and performance stability. It introduces a customizable rule system for users to specify their toleration of degradation, and overcommits resources with a dual approach. (1) For less-invoked functions, it allocates resources to the sandboxes with usage-based heuristic, keeps monitoring their performance, and remedies any detected degradation. It differentiates whether a degraded sandbox is affected externally by separating a contention-free environment and migrating the affected sandbox into there as a comparison baseline. (2) For frequently-invoked functions, Owl profiles the interference patterns among collocated sandboxes and place the sandboxes under the guidance of profiles. The collocation profiling is designed to tackle the constraints that profiling has to be conducted in production. Owl further consolidates idle sandboxes to reduce resource waste. We prototype Owl in our production system and implement a representative benchmark suite to evaluate it. The results demonstrate that the prototype could reduce VM cost by 43.80% and effectively mitigate latency degradation, with negligible overhead incurred.

1 INTRODUCTION

Function-as-a-Service (FaaS) is gaining increasing popularity in cloud computing. All major cloud providers have FaaS platforms [1–4] that allow users to write scalable, event-driven applications as a set of stateless, short-running functions. The FaaS platforms overtake the burden of resource management on behalf of users, including function provisioning, autoscaling, logging, and fault-tolerance. The platforms charge users only when their functions are running.

As the FaaS platform bears the cost of resource idling, reducing the resource provisioning cost (i.e., using fewer machines and less machine-time) is of paramount importance for the provider. Largely affecting the cost is the function *scheduling*. In a common FaaS platform, it launches a number of *host* machines (virtual or bare-metal) in a cloud as needed and run function instances as *sandboxes* (containers or secure sandboxes [24]) in hosts. When a function invocation request arrives, the scheduler *routes* it to an available function sandbox for execution. If no sandbox is currently available, the scheduler selects a host and *places* a new sandbox onto it to execute that request. The goal of scheduling is to dynamically pack function sandboxes to *as few hosts as possible*, without causing noticeable performance degradation.

To achieve this goal, we first conduct extensive trace analysis (§2.2) in our FaaS platform, Alibaba Function Compute [1], that serves a large number of users from various business domains. Our analysis reveals that most function sandboxes can only utilize 20–60% of their memory allocations. Furthermore, to avoid frequently cold-starting an instance, the platform retains an idle sandbox for a short period of time before terminating it, a technique known as *keep-alive* [36]. In our platform, keep-alive results in 25% of sandboxes idling for over half of their lifetimes.

Intuitively, the resource waste can be reduced with overcommitment, i.e., placing more sandboxes on a host than the allocation limit allows. However, the existing techniques of overcommitment are not directly applicable to FaaS. For instance, the overcommitment of container workload [8] and analytics tasks [29] allocates resources to containers by their real usage. But our experiment (§3.1) shows that the allocation based simply on actual usage could easily lead to performance degradation of the functions. It results from the unique characteristics of FaaS scenario.

The scheduling in FaaS differs from other scenarios mainly in two aspects. ❶ The function executed in each sandbox is highly *heterogeneous* and difficult to profile. Allowed to upload arbitrary code, the customers of FaaS span various industries and application domains. Hence unlike the analytics tasks that are often compute-intensive, a function could utilize multiple resources. The complex resource usage explains the degradation in our overcommitment experiment (§3.1). Due to function heterogeneity, the profiling has to be conducted over multiple resource types. Some of their usage (e.g., CPU and I/O devices) are dynamic and volatile on a short scale, and cannot be accurately measured with a single value. Many task scheduling techniques [13, 23, 25, 26] thus fall short in FaaS scheduling because they presume the availability of a per-task profile. ❷ As FaaS provides no functionality of storage or coordination, the functions usually involve some external services, such as the message queue, database or file system provided by the cloud. They could unexpectedly become a potential source of performance problems in FaaS platform. For example, their failure could degrade the request latency and the provider has to distinguish it from the scheduling-related degradation. The above two issues have shaped our design of the scheduling system.

Challenged by the difficulty of profiling in FaaS, we rethink how the resource usage can be described and develop a new technique—*collocation profiling* (§5.1). It does not measure the fine-grained usage of each resource, but counts how many sandboxes can be collocated on a host without causing contention, even when they all have saturated load. An example profile could be *function A can have at most ten sandboxes placed on a single host*. The upper bound of ten quantifies the highest extent of overcommitment this function can tolerate, which is a useful piece of information for sandbox placement. We further extends the profile to include two types of functions, e.g., *function A and B can have at most five and seven sandboxes collocated on a host respectively*. It enables the system to explore which two functions can be suitably collocated to achieve high utilization and low contention. The choice of two avoids the exponentially many combinations of functions and mixes function sufficiently well in practice.

We develop the idea of collocation into a placement system (§5.2). It mainly consists of two modules—a profiler that collocates functions and measures performance, and a placer that determines the sandbox location following the profiles. The challenge in designing the profiler is that it cannot invoke any function without user requests. Otherwise the invocation may cause unexpected side effect such as tainting users' database or disturbing users' devices. We thus design an in-production profiling procedure that can be conducted during serving the real user requests. The procedure calls the scheduling primitives to incrementally place more sandboxes onto a separated clean host and route requests to saturate them. As the sandboxes increase, the profiler observes how performance changes and selects the optimal quantity for the profile. However, it only applies to popular functions, because only popular ones can have sufficient requests to be profiled. We address the limitation by designing a complementary scheduling approach, which will be described shortly.

How to integrate the profiles in placement is the core problem we tackle in placer design. We first formulate the offline scenario—given a fixed number of different sandboxes, how to collocate them together—and design an algorithm that iteratively packs sandboxes following the most resource-efficient collocation. Then we extend it to the online setting by introducing a periodic update scheme that can partially update the cluster with the offline algorithm. Our profile-guided placer ensures that resources are properly overcommitted so as to avoid contention-resulted degradation in the first place.

For less popular functions that can not be profiled, the placer falls back to usage-based overcommitment, i.e., allocates resource according to their historical usage. It mitigates the ensuing degradation problem with latency monitoring and degradation remedy (e.g., sandbox isolation, migration). The challenge of monitoring in FaaS is that sometimes the detected performance drop is not caused by overcommitment but other issues unrelated to scheduling. For example, a user function may depend on an external gateway service whose timeout causes the latency surge. As this problem has nothing to do with resource contention, the scheduler should be able to differentiate it and stop a futile attempt to improve sandbox placement. We design a *comparative validation* technique to achieve that. It places the affected sandboxes in a contention-free environment and compares the resulting performance with the previous measurement. If the degradation persists, it confirms that the cause is from an external source that cannot be addressed by scheduling.

We have implemented a prototype scheduler, Owl, in our FaaS platform (§6). It has a hybrid design that differentiates between the popular and less popular functions. For the popular functions, Owl actively maintains their collocation profiles and uses them to guide the sandbox placement. It further consolidates the idle sandboxes to reduce resource waste whenever possible (§5.3). For the less popular functions, Owl overcommits resources when placing their sandboxes and reactively mitigates the detected performance degradation following the pre-specified rules.

We evaluate Owl in a cluster up to the scale of 200 VMs, against a benchmark suite that reflects the complexity of FaaS workloads in our production platform. The FaaS applications included in the suite span various business domains, are implemented in multiple programming languages, and have diverse external service dependencies. The benchmark suite has been released as open-source. The evaluation shows that Owl reduces the VM provisioning cost by 44% without degrading the function performance. The scheduling

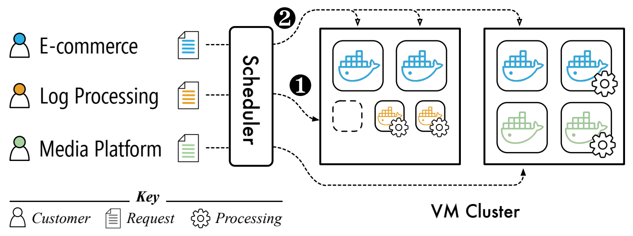

Figure 1: An overview of FaaS scheduling.

overhead, including both collocation profiling and reactive contention mitigation, is negligible.

To summarize, this paper makes following contributions:

- We identify the problem of FaaS scheduling and characterize its issues in a production environment.
- We propose several placement techniques that can differentiate the degradation cause, improve the resource utilization without affecting the performance.
- We implement a benchmark suite for evaluating the scheduling algorithm in FaaS platform. The code is released at https://github.com/All-less/faas-scheduling-benchmark.
- We prototype Owl in our production system and evaluate its effectiveness and overhead.

2 BACKGROUND AND MOTIVATION

2.1 FaaS Scheduling from Provider's View

Function-as-a-Service (FaaS) emerges as a new computing paradigm that greatly simplifies cloud programming: users simply package their application code as a set of functions, define the events that trigger these functions, and let the FaaS platform manage function provisioning and scheduling. Figure 1 gives an overview of FaaS scheduling in our platform. A user request may trigger two types of scheduling decisions: *placement* or *routing*. ❶ If the requested function has no idle instance, the scheduler chooses a VM and places a new instance on it, which runs in a sandbox (a secure container similar to gVisor [24]). Note that the chosen VM must have enough memory to accommodate the function's allocation demand. ❷ If the function has multiple sandboxes available for the request, the scheduler chooses one and *routes* the request to it for execution.

By default, our scheduler uses first-fit bin-packing for sandbox placement and the most-recently-used (MRU) request routing algorithm. The scheduler maintains an inventory of running VMs and places a new sandbox onto the first VM that can accommodate it. (AWS Lambda is reported to adopt a similar strategy [42].) When performing request routing, the scheduler prioritizes the most recently invoked sandbox. This strategy is designed in accordance with the *keep-alive* mechanism, in which a sandbox can stay idle for up to five minutes before terminating by the platform. The MRU routing algorithm results in a longer idle time for the least invoked sandboxes so that their resources can be reclaimed sooner.

From the provider's perspective, the goal of FaaS scheduling is two-fold: (1) packing the function sandboxes to *as few VMs as possible* to minimize the platform's resource provisioning cost, and (2) in the meantime protecting the performance of user functions. However, our current scheduler falls short in achieving the two objectives, as we explain below.

2.2 Workload Characterization

To understand the challenges of FaaS scheduling, we collect an one-day trace (on the scale of $O(10^8)$ invocations) from our production platform that serves a large number of users running diverse applications, such as e-commerce, online education, manufacturing, and SaaS providers. We analyze the trace and have the following findings.

Underutilized Resources By default, a function sandbox receives an full allocation of memory as specified in the user configuration, which is usually more than the function can actually use. Figure 2 (red dashed line) depicts the cumulative distribution of memory utilization (in proportion of the allocation). Most sandboxes can only use 20–60% of their allocated memory. There are two main reasons of memory under-utilization. First, at the moment, the minimum memory size one can configure for a function is 128 MiB [14]. However, many functions perform rather simple computations with memory footprint much smaller than 128 MiB. Second, we find that many users intentionally configure a larger memory for the sake of obtaining more CPU cores for their applications, as in current FaaS platforms memory and CPUs are allocated in a fixed proportion (e.g., 1.5 GiB per core in our platform [15]).

Like in other FaaS platforms, a function sandbox is not terminated immediately once becoming idle, but it will be kept alive for a configurable short period of time (5 minutes in our platform): if no request is received during that interval, the sandbox is then terminated. This *keep-alive* mechanism [36] is used to avoid frequent tear-and-wear cost due to sandbox creation and destroying. However, we find that it results in a long idling time for a large number of sandboxes. Figure 2 (blue solid line) shows the distribution of the keep-alive idling period relative to the sandbox's lifetime. There are 25% of sandboxes kept alive in idle for over 50% of their lifetimes.

Skewed Function Popularity Just like other cloud services, such as storage and data analytics [11, 27, 46, 48], FaaS functions have highly skewed popularity. A small number of extremely popular functions spawn a large number of sandboxes in the system. In our traces, 61% of sandboxes belong

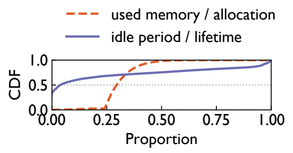

Figure 2: For many sandboxes, memory utilization is low and the keep-alive mechanism leaves CPU idle.

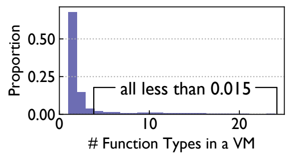

Figure 3: Owing to skewed function popularity, most VMs only host one or two types of sandboxes.

to 0.65% of functions that are of top popularity. Moreover, the top 5% (20%) functions create 88% (97%) of sandboxes. Our observation is in line with that reported by Microsoft Azure [36]. The skewed popularity results in an undesirable scheduling consequence that sandboxes belonging to the same popular function are *concentrated* on VMs. Figure 3 shows the number of the belonging functions of sandboxes running on each VM. Around 65% of VMs exclusively run sandboxes of one function. As these sandboxes run the same function code, they contend on the same type of resource (e.g., I/O) while leaving the other resources (e.g., memory and CPU) underutilized.

Diverse Performance Requirements While in general a FaaS platform should provide low latency for functions, users have different performance expectations and requirements for their applications. We have interviewed our service operators [7] about the top clients and their service requirements. The feedback suggests that the requirements vary depending on the business domains. For example, the payment function from an e-commerce service is sensitive to latency fluctuation as it is in the critical path of transaction processing. In comparison, the data compression functions deployed by a log processing company are more tolerant to latency variations. In fact, some user applications invoke functions asynchronously, queuing their requests for batch processing. These functions are inherently tolerant of a long latency.

Drawing on the findings above, it is imperative to design a new FaaS scheduler that can (1) tightly pack functions to VMs to achieve high cluster utilization, (2) be aware of the diverse latency requirements of user functions and strive to meet them, and (3) efficiently make scheduling decisions with low overhead.

3 THE CHALLENGES OF OVERCOMMITMENT

Given the significant memory under-utilization of function sandboxes (§2.2), a mitigatory scheduling approach is to overcommit more sandboxes on a VM beyond its memory limit. However, this may result in prolonged function executions due to the increased resource contention (§3.1). To make the problem even more complicated, the latency increase may also be caused by other unrelated issues beyond the scheduler's control (§3.2). It is thus necessary for the scheduler to differentiate between different sources of degradation and judiciously perform overcommitment.

3.1 The Risk of Overcommitment

Usage-Based Overcommitment We design a heuristic scheduling algorithm that overcommits resources by allocating memory to sandboxes based on their past usage (which is typically smaller than the configured size). To prevent the out-of-memory error, the algorithm also allocates some "slack" memory. Specifically, for each function, we maintain a running estimate of memory usage u by averaging over the past 1000 requests. Supposing its configured runtime size is c, the algorithm allocates $pu + (1 - p)c$ memory to its sandbox at creation, where p is a tunable percentage controlling the slack space. In this paper, we set p as 50% as it strikes a good balance between resource saving and slowed function execution.

Problems of Usage-Based Overcommitment The usage-based overcommitment scheme can effectively improve resource utilization. Yet, it has several drawbacks that may harm the function performance. First, its fixed allocation may slow down some functions with fluctuating memory usage, because their memory operations could be clogged by the reduced allocation. Second, the scheme only considers the sandbox memory but ignores its CPU utilization. Though memory-based placement is a common practice in public FaaS platforms, it may result in an unsuitably high placement density for CPU-intensive functions. Third, the scheme does not account for the contention between collocated sandboxes.

To demonstrate the impact of these drawbacks, we compare the performance of the same functions in two FaaS clusters with and without overcommitment scheduling, respectively. We run the same workloads (i.e., identical functions and request sequences) in the two clusters.[1] Figure 4 cites the two representative functions whose P95 latency surges by 207% and 107%, respectively. Such a significant performance drop is unacceptable for latency-sensitive applications.

3.2 Causes of Degradation

One unique difficulty of tackling performance drops in FaaS is that *not* all of them are caused by overcommitment, meaning, the occurrence of degradation does not necessarily indicate that the scheduling is problematic. Possible causes include failures of an external dependency or changes in

[1]The results are extracted from Workload-B with details given in §7.2.
[2]Table 2 lists the function abbreviations.

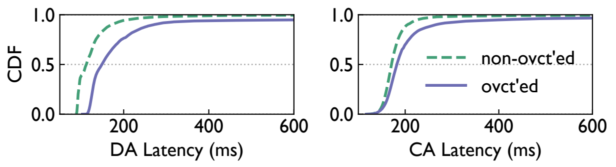

Figure 4: The latency of function DA (detect-anomaly) and CA (convert-audio)[2] degrades because of overcommitment.

request parameters. We found many anecdotal evidences in our production platform. To name a few, one data processing function was reported to exhibit degraded performance, and the investigation discovered that the root cause was the timeout of its external gateway; another IoT function was reported to significantly slow down in face of many requests, and the reason turned out to be the rate limit imposed by its external proxy. In fact, over 40% of sandboxes have CoV (coefficient of variance) larger than one in the request latency. The inherently varying latency could be mis-detected as degradation.

We find that the performance degradation caused by different sources cannot be easily differentiated, as they usually exhibit similar problems. For instance, the data processing function mentioned above got slowed down due to an external gateway. Similar processing slowdown can also be caused by an overly condensed sandbox placement. Only the latter cause should (and could) be addressed by the scheduler. We thus propose the following principle to scope the responsibility of FaaS scheduling:

> **Scoped Responsibility:** *The service provider (and the scheduler) is held responsible only for the degradation caused by the execution environment.*

Following this principle, our scheduler only addresses the performance degradation caused by resource insufficiency or uncontrolled interference.

4 OWL OVERVIEW

In face of the degradation, the scheduler could either apply mitigation after it occurs, or carefully plan the placement beforehand to prevent it from happening. However, for popular functions with many sandboxes, their degradation could be too wide spread to mitigate. For the long-tailed unpopular functions, their large quantity makes it costly to construct detailed model about their behavior and adjust the placement beforehand. We thus design a hybrid approach that can automatically classify functions based on popularity and schedule them accordingly.

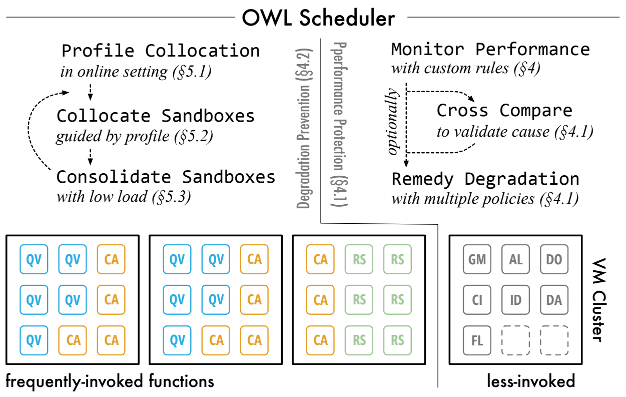

Figure 5: A high-level overview of Owl.

For less popular functions, Owl employs usage-based allocation (§3.1) and tackles the ensuing degradation by monitoring function performance and remedying the detected slowdown (§4.1); for popular functions, it applies a degradation-preventive measure by profiling the interference patterns of their sandboxes and overcommitting them in a contention-free manner (§4.2). As the profiling requires a function to have sufficient number of requests for a stable period of time, it becomes a natural criteria of popularity for classifying the functions. Hence Owl can automatically decide if the function is popular based on its incoming requests. Figure 5 shows a high-level overview of Owl.

Performance Rules In view of the diverse performance requirements of our users, we design a customizable rule format for users to specify their performance objectives. Inspired by SLOs (Service-Level Objectives [9]), the rules accept several choices of metrics measured at a configurable checking frequency. Below shows the format and a concrete example.

Format: Every `[interval]`, check if
`[metric] [op] [threshold]` within `[window]`.
Example: Every `30 seconds`, check if
`P95 latency < 300ms` within `past 1000 requests`.

Our current implementation supports the `metric` of latency or throughput (i.e., requests per second). The former targets online services, whereas the latter suits batch processing applications. The `op` is a common mathematical comparator. Both the `interval` and `window` can be a time interval or a number of requests. The checking `interval` captures the users' requirement on detection timeliness. The examination `window` specifies the sensitivity of detection. Users could demand higher performance stability by setting a smaller `window` size.

4.1 Performance Protection

To achieve high utilization, Owl places function sandboxes with the usage-based heuristic. It addresses the potential degradation in three steps. (1) As the basis, Owl monitors the sandbox performance and detects the degradation according to the *user-defined rules*. Owl provides a flexible rule format to allow users to express their diverse performance requirements. (2) When the degradation occurs, Owl remedies it with a combination of policies. (3) As the degradation may result from external causes beyond the scheduler's control, Owl uses a *comparative validation* technique to differentiate them.

Degradation Remedy When Owl detects a violation to the performance rule, it attempts to remedy the affected sandbox using the following policies:

(1) Migrating the affected sandbox to another VM that has the sandboxes of the same function. It presumes that the target VM induce no interference against the function.
(2) Migrating the degraded sandbox to another VM with lower utilization. It assumes the target VM has a less degree of contention.
(3) Moving away the last created sandbox(es) from the host VM. It aims at relieving the resource pressure on the VM.
(4) Migrating the sandbox to a non-overcommitted VM.

By default, Owl applies the first policy and, if the function has no other instances, switches to the second policy. When multiple sandbox degradations are detected on a machine, Owl deems it as a VM-level problem and applies the third policy. If the degradation persists after all these trials, Owl turns to the fourth policy as the last resort. That being said, functions are allowed to set their own remedy workflows. For example, a high-priority function can choose the fourth policy as the primary way to address degradation.

Comparative Validation In a public FaaS platform, the performance degradation does not always mean the problem of execution environment. As the function invocation accepts custom parameters, the degradation may simply reflect a parameter change. Moreover, many user functions depend on external database, network services, and even other functions. The external degradation can also cause the performance issue. As a concrete example, we have once received customer complaint about slowed latency, which is found to be caused by the timeout in their CDN gateways. The scheduler is thus challenged to accurately differentiate the cause, mitigate those caused by execution environment, and preclude the degradation from unrelated factors.

We design a technique, *comparative validation*, to verify the cause of degradation. Its underlying assumption is that, if the sandbox is running in a non-overcommitted environment, then its degradation results from the function itself. Hence the core step in validation is to set up a clean environment as the baseline for comparison. Specifically, the scheduler selects a non-overcommitted host machine and migrates the degraded sandbox onto it. Then the scheduler monitors its performance and compares with that before the migration. If the degradation shows no sign of alleviation, Owl confirms that the cause is external. Otherwise, the degradation is concluded to be caused by the execution environment.

We further design a variant of the technique for the functions with a large number of sandboxes. We term the procedure described in the last paragraph as *temporal comparison*, because it compares the performance before and after migration. If the function has many sandboxes, it is possible to always keep one of its sandbox on a non-overcommitted machine and let it serve as the baseline. The comparison no longer requires migration and can be done in real time. We refer to such setting as the *canary comparison*.

The technique may encounter the corner case where the external problem is transient (i.e., recovering after a short period) and the scheduler misattributes it to the overcommitment. But such error actually has limited and manageable impact. Between the two validation variants, the transient problems don't affect the canary comparison as it is conducted in real-time. Since it covers most popular functions, that means most sandboxes are immune to the transient problems. As for temporal comparison, the system will record the relevant logs each time the validation is conducted. If the problem becomes widespread and causes system-wide abnormalities, the operator would manually check the logs and correct the mistake.

4.2 Degradation Prevention

For popular functions, Owl establishes a *sandbox collocation profile*, i.e., what function sandboxes and how many of them could be collocated together without incurring detectable contention. Based on this profile, the scheduler performs contention-free overcommitment. We choose to profile popular functions because they have sufficient requests for performance evaluation in various collocation combinations. Besides, popular functions spawn the majority of sandboxes as illustrated in §2.2.

But still, there are an exponentially large number of sandbox combinations on a single VM, so exhaustively profiling each option is computationally prohibitive. We thus restrict the profiling to *two functions* only. That is, we only allow *two* types of function sandboxes to be collocated on a VM. This restriction substantially reduces the search space. Note that CPU and memory are the two major resource types in FaaS, and the collocation of two functions can well mix sandboxes with complementing dominant resources. Our evaluation confirms the effectiveness of our approach despite such a restriction.

More specifically, the scheduler profiles function performance on VMs with different sandbox collocations in production in search for the contention-free collocation pairs. Figure 5 shows a concrete example, where at most 5 sandboxes of function QV and 4 of CA can be collocated (the lower left VM). This profile can guide Owl to place incoming sandboxes. We will elaborate on how to establish the profile and follow its recommended placement rules in §5. As the collocation profile packs as many sandboxes as possible onto a VM based on the resulting performance, not their configured memory sizes, overcommitment is naturally supported to achieve high utilization. In the mean time, contending sandboxes will not be collocated, thus avoiding the potential performance degradation.

5 PROFILE-GUIDED PLACEMENT

Underpinning the degradation prevention is a profiling technique (§5.1) that discovers contention-free collocations. It guides Owl to maintain contention-free placement among the popular functions (§5.2). However, as we will show in §5.3, the contention is avoided at the expenses of some resources left idle. We thus propose another technique called *sandbox consolidation* to further reduce the waste.

5.1 Collocation Profiling

Owl learns which functions can be collocated together by varying the placement of different collocations and profiling their performance. We propose a data structure, collocation profile, to store the profiling results. We then design a scheme to profile the collocation of two functions in the production environment. Finally we describe how Owl executes the scheme and constructs the profiles for popular functions.

Collocation Profile Essentially, the profile describes, given two functions, how many of their sandboxes can be collocated on a single VM without contention. Table 1 presents an example profile for a function QV in our benchmark suite (details in §7.2 and Table 2). The last three columns profile its collocation with functions CA, GMM and AL, while the second column is a special case where QV is the only function on a VM. The second row lists the maximum number of sandboxes, e.g., "6 / 10" in the CA column meaning at most 6 QV sandboxes and 10 CA sandboxes can be collocated. The third row records the allocation ratio (i.e., the total allocated memory divided by the machine limit) in each collocation. Owl prioritizes a high-ratio collocation during placement because it implies higher overcommitment. That the ratio is usually greater than one is a consequence of prevalent memory underutilization (§2.2).

Table 1: A sample function profile of QV (abbreviations explained in Table 2). The allocation ratio calculates the ratio of the total allocated memory to machine limit. A value larger than one indicates overcommitment.

Collocation	QV (self)	CA	GMM	AL
# Sandbox (self/other)	14 / -	6 / 10	6 / 14	10 / 1
Alloc. Ratio	1.17	1.33	1.08	1.17

Profiling Collocation Performance Given two chosen functions, Owl first singles out a VM and collocates their sandboxes on it.[3] The system then routes requests to keep those sandboxes saturated and observes if any of their performance rule is violated. If not, it fills more sandboxes and repeats the observation. Finally when the performance starts deteriorating, Owl records the sandbox quantity and the allocation ratio in the collocation profile. The example profile in Table 1 is obtained by repeating the procedure for four times (i.e., collocating QV with CA, GMM, AL and itself respectively). The profiling is conducted in the production environment and §6 will explain how the scheduling primitives can complete the involved operations.

The profiling does not affect the service quality in production because Owl is configured with conservative termination criteria. It detects the degradation with a more strict version of the user-defined rule. Specifically, suppose the rule sets the threshold value as t, and the metric calculated from the historical execution is h. The terminating rule will use $\alpha \cdot t + (1 - \alpha) \cdot h$ as the threshold where $\alpha \in (0, 1)$ is a tunable parameters for controlling the conservativeness. In practice, we find an α of 50% can strike a good balance.

Constructing Collocation Profiles For a newly added popular function, Owl builds its profile by first profiling its collocation with itself (cf. the second column in Table 1), and then the collocation with others. Within the building process exist several design considerations.

- *What functions are counted as "popular"?* We define the *popular* function based on the arrival rate of its requests. As the sandboxes are continuously triggered (i.e., saturated load) during profiling, not every function has adequate incoming requests. Hence a natural criterion is that the requests of a *popular* function can keep saturating, for a period of time, so many sandboxes that they can fill up one VM. In practice, when the requests of a function stay above this threshold, Owl will start collecting its collocation profile. If the arrival rate drops during the profiling, then the process will be interrupted.

[3]How the number of sandboxes is chosen and how Owl adjusts them are described in Appendix A (Algorithm 3).

• *How many collocation pairs does a profile usually have?* The guiding rule is, for each function, to have collocation with another β of the popular functions, where β is a tunable percentage. It ensures the scheduler could have enough possible pairs to explore when placing the sandboxes. In practice, we find a β of 10% is enough for well-mixed collocation. But the actual number in each profile may vary because the profiling may be interrupted. So we design the placement algorithm that works with no requirement on the pair number.

• *How does Owl choose which function to collocate?* When constructing the profile for a function, Owl chooses its collocation based on a resource heuristic. It tries to find the functions with the complementary resource patterns, which are measured by the ratio between memory usage and CPU utilization. For a high-ratio function, Owl attempts to match it with low-ratio ones, and vice versa.

5.2 Profile-Guided Placement

With the profiles collected, Owl follows their guidance for a degradation-avoidant placement. For the ease of presentation, we first consider an offline version of the problem, that is, given a *fixed* set of sandboxes, how to collocate them to achieve high utilization. We design an algorithm that iteratively pairs up the sandboxes according to the profiles. Then we extend the algorithm to the dynamic setting where the sandboxes can be created and terminated.

Offline Problem Formulation At a given moment, suppose the system has a set of functions $F = [f_1, f_2, \cdots]$. The corresponding number of their sandboxes are $N = [n_1, n_2, \cdots]$. Let $P = [\cdots, p_{ij}, \cdots]$ denote the collocation pairs extracted from the function profiles of F. That is, if the collocation of f_i and f_j has been profiled, P will include an entry p_{ij} in the form of (s_i, s_j, a_{ij}), representing the sandbox quantities and the allocation ratio (as shown in Figure 1). The problem of collocation-based placement can be formulated as, given F, N and P, how to find a contention-free and high-utilization arrangement M, where M is a list of key-value pairs $\{f_i : n_i, f_j : n_j\}$. Each pair corresponds to one VM and means that n_i sandboxes of the function f_i and n_j sandboxes of f_j are collocated.

Offline Algorithm The intuition behind Algorithm 1 is to start with the collocation pair of the highest allocation ratio and pack as many sandboxes according to the pair. Then it iterates along the pairs and repeats the packing until no sufficient sandboxes remain. The algorithm is guaranteed to terminate because the collocation profile has a self-collocation entry. If any sandboxes haven't been packed with others, they can still be placed with themselves. The generated placement plan can be implemented in the cluster by migrating the sandboxes, which will be described in §6.

Algorithm 1 Place sandboxes according to function profiles.

```
1: $M = [\ ]$: The placeholder for resulting placement.
2: function PLACE($F, N, M, P$)
3:     Sort $P$ by allocation ratio in descending order.
4:     for each $p_{ij} = (s_i, s_j, a_{ij})$ in $P$ do
5:         $t \leftarrow \min\{\lfloor n_i/s_i \rfloor, \lfloor n_j/s_j \rfloor\}$
6:         Append $\{f_i : s_i,\ f_j : s_j\}$ to $M$ for $t$ times.
7:         $n_i \leftarrow n_i - ts_i, n_j \leftarrow n_j - ts_j$
```

Online Updating When some sandboxes are created or terminated, the placement generated by Algorithm 1 may no longer be optimal and thus need an update by recalculating the placement. However, if the recalculation involves all sandboxes, the update would be time-consuming. We introduce a *marking mechanism* that helps reduce the VMs in each update. Specifically, every time a new arrangement is generated, the VMs whose sandboxes exactly matches a collocation pair (Line 6) are marked as "full".[4] They will be excluded from the next update. When any sandbox gets removed from a "full" VM, its mark will be revoked. When over γ (a tunable percentage) of the VMs no longer have the mark, a placement update will be launched. Since the placement is no longer updated after each sandbox creation, the new sandboxes are placed onto separate VMs first. The separation guarantees those sandboxes can not interfere with each other. Those VMs will be included in the next placement update. In practice, we set γ to be 10% to avoid too large-scale update.

To further reduce the placement latency of a new sandbox, we maintain a per-function *placement cache* to memorize the positions of the terminated sandboxes. For instance, when a sandbox of function QV is removed from VM1, we insert a placeholder of VM1 into the cache of QV. Next time a sandbox of QV is to be created, it can be directly created on VM1. It is guaranteed to abide by the collocation rules. If the VM1 has a "full" mark before the removal and a new sandbox is refilled onto it, then its "full" mark will be restored.

5.3 Consolidating Sandboxes

The placement in §5.2 prevents contention even under the saturated load, but causes resource waste at a lower load. It is because the collocation profile is obtained under the condition of *saturated load*, a worst-case assumption about the runtime environment. However in reality, the function load may fluctuate and some of the sandboxes may become idle, thus incurring resource waste.

Algorithm 2 mitigates the problem by consolidating the mostly idle sandboxes. The intuition is to migrate those sandboxes from less utilized VMs (i.e., with more resource slack)

[4]In comparison, some sandboxes are left over and not paired with other in Algorithm 1, so their host VMs are not "full".

Algorithm 2 Consolidate idle sandboxes.

```
1: function Consolidate
2:     for $f \in$ profile-enabled functions do
3:         $C \leftarrow$ sandboxes of $f$ that idle for 80% of the past period
4:         $M \leftarrow$ VMs that host sandboxes in $C$
5:         for $m \in M$ do
6:             Collect CPU share and memory of $m$ in past period.
7:             Calculate the resource slack of $m$.
8:         for $c \in C$ do
9:             Calculate the average CPU share of $c$ in past period
10:            $m \leftarrow$ VM with smallest sufficient slack for $c$
11:            Migrate $c$ to $m$ if $m$ exists and $c$ is not on $m$.
```

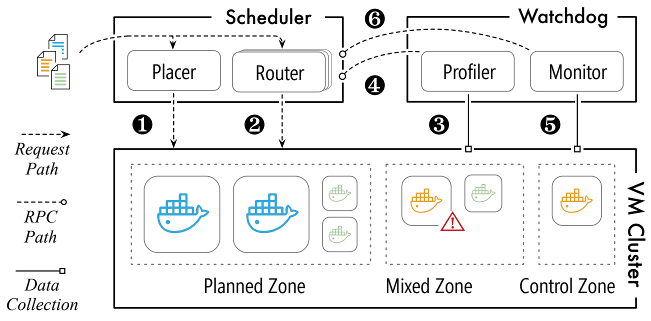

Figure 6: System Architecture

to more utilized ones, so that the latter can consolidate more resources. The source VMs of migration are revoked of its "full" mark and they are included in the next update to increase utilization. Note that the consolidation in Algorithm 2 is conducted within each functions, i.e., the source and target VM of the migrated sandbox both already have other function instances. Hence the migration will not cause any functions to be collocated unexpectedly.

However, if the migrated idle sandbox starts to receive more requests, it may contend with other sandboxes. For instance, suppose a VM can have at most 10 sandboxes being concurrently invoked. If we migrate two more idle sandboxes onto it, we have to prevent more than 10 sandboxes from being triggered. We thus introduce *grouped routing* whose purpose is to keep the overall resource usage of a VM within the same level as before migration. Specifically, the scheduler creates a `group` structure to logically bind the two migrated sandboxes with two existing sandboxes. The router ensures that at most two sandboxes in the `group` will be simultaneously invoked, thus keeping the overall resource usage in check.

6 OWL SCHEDULING SYSTEM

This section describes how the components in Owl fulfills the functionalities of performance protection and degradation prevention.

6.1 System Design

System Architecture Owl mainly consists of two components, the *scheduler* and the *watchdog* (Figure 6). The former performs the scheduling decisions during request processing, including the placement and routing. To shorten the latency, we separate other operations into the watchdog, a background component that monitors and profiles the sandbox performance. Within the scheduler are a *centralized placer* and multiple *separate routers*. The routers are created on a per-function basis so that they can leverage multiple CPU cores and sustain high throughput. The placer is centralized because it requires a consistent view of cluster status to perform placement decision. We discuss how to scale it in §8.

The VM cluster is logically partitioned into three zones. The *planned* zone is primarily for popular functions with guided placement. The *mixed* zone is for the rest of the functions. Their resources are overcommited and their performance is protected by the remedy mechanism. The *control* zone is a non-overcommited environment which would serve as the comparison baseline in comparative validation and the last resort for degradation remedy.

Scheduling Primitives The design of Owl presumes the availability of several sandbox operations, such as adjusting sandboxes in profiling, migrating them during remedy, and grouped routing for consolidated sandboxes. The scheduler provides several RPC *primitives* for those operations.

- *Route Diversion*: Lower the priority of a given sandbox during routing. It is mostly triggered after a degradation is detected.
- *Route Prioritization*: Increase the priority as opposite to route diversion. It can prioritize a sandbox and keep it being saturated by requests, which is a required condition for collocation profiling.
- *Sandbox Migration*: Create a new sandbox, replace the original one in the router, and then terminates the original one. The termination waits until the original sandbox finishes the request and the new one starts processing to ensure the processing is not interrupted.
- *Sandbox Grouping/Ungrouping*: Group together/ungroup a set of sandboxes so that their requests will stay within the allocation limit.

Scheduling Workflow As shown in Figure 6, ❶ when a request requires a new sandbox, the *placer* searches for its appropriate placement. If the request belongs to a profile-enabled popular function, the placer will conduct the two-phase placement as described in §5.2. If otherwise, the placer allocates the sandbox in the mixed zone with usage-based overcommitment (§3.1). ❷ After creation, the sandbox is inserted into the record of *router*. It dispatches the requests with the MRU policy in general, and adjusts the routing when the diversion, prioritization and sandbox grouping is in effect. ❸ If the function has sufficiently high request load for profiling, the *profiler* coordinates the collection of its collocation profiles. ❹ The profiles are communicated to the placer to be consulted in collocation-based placement. ❺ When the function performance violates the performance rule, the degradation will be detected by the *monitor*. ❻ It issues RPC calls to the scheduler to attempt a remedy. The monitor also checks the utilization of sandboxes and initiates the consolidation (§5.3).

6.2 System Implementation

We prototype Owl in the production system with roughly 7000 lines of Go code. The prototype extends the original scheduler, which has a similar architecture as OpenWhisk: in our system, the *scheduler* schedules *requests* to *host machines*; in OpenWhisk, the *Controller* schedules *Invocations* to *Invokers*. For the readers interested in the original scheduler, we refer them to OpenWhisk for reference. Below we describe how the major components in the prototype are implemented.

6.2.1 Scheduler. The scheduler exposes an interface as an RPC server. Its interacts with the rest of the FaaS platform by receiving incoming requests and returning which sandboxes it selects to execute them. If no available sandbox exists, the scheduler creates new ones and returns them as result. Within the scheduler are two major types of modules: the routers are created on a per-function basis, and a centralized placer for determining placement.

Router For each function, a router maintains a list of busy sandboxes and an *ordered* list of available sandboxes. The order follows how recently the sandbox has been invoked. The router tends to pick the first sandbox so that the less invoked sandboxes can stay idle for longer and get recycled sooner (due to the keep-alive mechanism). The scheduling primitives are implemented as the operations on the sandbox list. The *route diversion* marks a sandbox as unschedulable and the router will skip it during routing. We add a separate list for *route prioritization* so that the prioritized sandboxes can be picked first. The *sandbox grouping* associates a `group` data structure with all the involved sandboxes. The `group` records how many sandboxes in it are currently being invoked. Every time a router selects the sandbox within the `group`, it has to update the record. Whenever the recorded requests consumes more than the resource limit, the grouped sandboxes will be skipped during routing.

Placer When a new sandbox is being created, the placer determines its location in the cluster. If all host machines are fully utilized, the placer acquires new machines from the public cloud. The system maintains a buffer pool of vacant machines to accelerate the acquisition process. To logically separate the cluster into three zones, each machine is labelled with `planned`, `mixed` or `control`. For the functions without collocation profiles, the placer searches for its placement on the `mixed` machines, following the naive overcommitment (§3.1) algorithm. For profiled functions, the placer instead follows the profile-guided algorithm (§5.2) and only places them onto the `planned` machines. The `control` machines are reserved for sandbox isolation or collocation profiling, and the sandboxes are usually migrated onto them. The primitive of *sandbox migration* is implemented as three steps: creating a new sandbox, inserting its record into the router and tearing down the original sandbox. Note that it is different from the live migration of a long-running container in that no real data migration is performed. The stateless nature of FaaS sandboxes enables us to use quick recreation as a migration strategy.

6.2.2 Watchdog. The watchdog serves the auxiliary functionality of performance monitoring and function profiling. As it is not directly involved in request serving, we thus implement it as a standalone long-running background component. It also consists of two major modules.

Monitor keeps examining the function performance, applies remedies when the degradation is detected, and validates its cause when the remedy fails. When implementing the monitor, we consider it unscalable to monitor each function separately. Hence the monitoring is carried out by function batches which are aggregated based on the interval of their rules (§4). As a concrete example, a batch may include all functions that are set to check latency every 15 seconds. In our platform, the performance rules and the request records are stored as database tables. Hence the rule-checking is implemented as a SQL computation that joins the two tables, groups the requests by functions, calculates the latency metrics specified by the users and compares them with the threshold values. To accelerate the computation, the table of request records is partitioned by generated timestamps (rounded to seconds) and function checking interval. If the comparison indicates that degradation occurs, the monitor takes remedial actions that are outlined in §4.1. If the degradation persists after remedies, the monitor performs comparative validation to verify the potential cause. If the cause is proven external, the monitor records the comparison results so that the system operators could manually inspect them later.

Table 2: Short description of the benchmark suite.

Function	Mem. Size	Usage
QV: **Q**uery **V**acancy state.	256MiB	~70MiB
RS: **R**eserve a parking **S**pot.	256MiB	~70MiB
AL: **A**nonymize **L**og from MQ.	1024MiB	~20MiB
FL: **F**ilter **L**og from MQ.	1024MiB	~20MiB
DO: **D**etect **O**bject in image.	3072MiB	~1700MiB
CI: **C**lassify an **I**mage.	2560MiB	~500MiB
GMM: **G**et **M**edia **M**etadata.	128MiB	~20MiB
CA: **C**onvert **A**udio encoding.	256MiB	~100MiB
ID: **I**ngest **D**ata from IoT.	768MiB	~10MiB
DA: **D**etect **A**nomaly in data.	768MiB	~10MiB

Profiler manages the profiling procedures of popular functions. It periodically scans the recent request rate of functions and the completeness of their collocation profiles. When it finds a function with sufficient requests but incomplete profile, it pushes the function into a queue to be profiled. During the procedure, the adjustment of sandbox quantity is implemented with migration. If the incoming requests decreases during the profiling, or the monitor discovers an external problem, the profiling will be immediately aborted.

7 EVALUATION

We prototype OWL in our production system and evaluate its effectiveness and overhead. The evaluation highlights include:

- OWL reduces VM cost by 43.80% on a representative benchmark without violating the performance rule of tail latency (§7.2 and §7.6).
- OWL has comparable scheduling latency and scalability to the existing scheduler (§7.3).
- OWL can effectively distinguish the degradation caused by different factors (§7.4).
- OWL incurs little overhead from comparative validation and placement migration (§7.5).

7.1 Benchmark Suite

We design a benchmark suite for evaluating the effectiveness of our scheduler design. We cannot borrow user functions as they may have unexpected side effects on users' data. Although several benchmarks [31, 47] exist for the serverless scenario, they are not suitable for scheduler evaluation because the majority of them are either single-resource test programs or data processing applications. The former has too simplistic resource behavior to emulate the normal user functions. The latter is not sufficiently representative of our customer base. Hence in our benchmark suite (Table 2 and more details in Appendix §B), the applications span the three most popular domains in our production environment: backend hosting, multi-media processing and IoT data analytics. We select five real-world applications from existing customers and implement part of their functionality. The functions therein covers different memory size[5], external dependencies and programming languages. The source code of the benchmark can be accessed via https://github.com/All-less/faas-scheduling-benchmark.

7.2 End-to-End Comparison

We first evaluate how OWL and each of its techniques affect the resource utilization and function performance.

Testbed The system is deployed with each component running on a VM instance with 8 CPU cores, 16GiB memory and 2.5Gbps network. The VMs for hosting function sandboxes have 2 CPU cores, 4GiB memory (where 3GiB is allocatable and 1GiB is reserved for system components) and 1Gbps network. The host VMs are dynamically acquired over the course of experiment and they can reach hundreds of VMs.

Baselines Four variants of the scheduler are included in the comparison.

- Baseline (BASE): The scheduler in the production system without overcommitment and performance awareness.
- Overcommitted (BASE-OC): The scheduler that only implements naive overcommitment (§3.1).
- Profile-Guided (OWL-PG): The BASE-OC scheduler that adds profile-guided placement (§5.2).
- OWL: The OWL-PG scheduler with sandbox consolidation (§5.3) enabled.

BASE and BASE-OC represent two extremes in the design space: the former does not cause degradation yet is resource-inefficient; the latter saves the most resources but its induced degradation is unacceptable in production.

Workload Each workload consists of a set of benchmark functions and request traces. The trace is a record of function invocations, including their quantity and time. We expect the workload to possess a similar degree of complexity as in the production environment. Hence we sample the traces directly from production logs. We have collected three workloads (labeled as Workload-A, B, and C), each with the same benchmark suite and ten different request traces. Their scale is chosen to occupy 100, 120, and 80 VMs respectively.

Methodology For each scheduler variant, we clear the VMs, deploy the scheduler, execute the benchmark workload and collect the metrics. We test all the variants on all three workloads. At the beginning of each trial, we warm up some sandboxes because a scheduler in production rarely encounters an empty cluster and the warm-up could avoid a surge of sandbox creation. The prewarming creates for each

[5]Following the common practice [44], the memory size is configured such that the cost of a single request is the lowest.

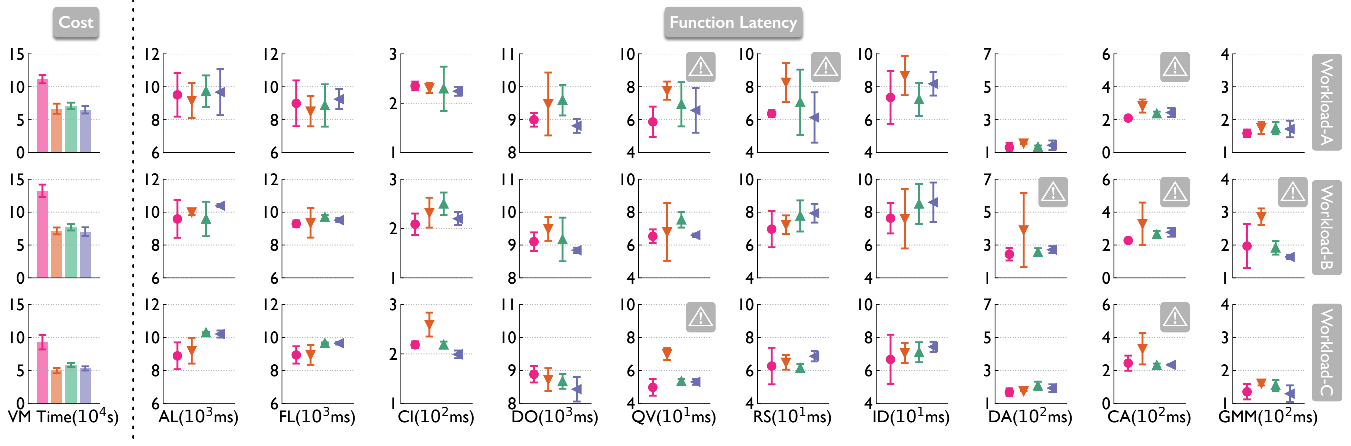

Figure 7: Each row reports the VM time and the tail latency of ten functions of a workload. The functions with a warning sign on the upper right corner show degradation in the Base-OC trial; they are marked as profile-enabled in the Owl-PG and Owl trial. (The symbols in the order of Base, Base-OC, Owl-PG, Owl)

function 60% of its peak number of sandboxes. The peak is estimated from the maximum request rate in the trace.

The performance rule for benchmark functions is specified as “P95 latency should not increase by more than 25%”. The figure 25% is chosen because the function latency in the non-overcommited environment can vary as much as 20% (e.g., ID in Workload-A). Across the workloads, we change the functions that Owl-PG and Owl scheduler provide performance protection to. As the request trace differs in each workload, the same function receives a different number of requests and becomes differently susceptible to resource contention across the experiment. If Base-OC causes a function to break the performance rule, we configure it as profile-enabled in Owl-PG and Owl.

Metrics We quantify resource costs with the *VM time*, i.e., the total time of VMs that are occupied by some sandboxes in the benchmark workload. It measures the purchasing cost of those VMs in a pay-as-you-go billing model. The function performance is measured by its *P95 latency* throughout all requests. We repeat each experiment for three times and report the average and the standard deviation of the metrics.

Results Figure 7 presents the results of three workloads. Overall, Owl reduces the VM time by 43.80% without any function breaking the performance rule. As a concrete example, in workload-A, Base acquires 92 VMs in total and Owl 49 VMs. Inspected closely, Base-OC saves 44.26% of VM time over Base, which means overcommitment brings the most of resource saving. Yet it causes the degradation in tail latency, such as QV, RS, CA in Workload-A, DA, CA, GMM in Workload-B, and QV, CA in Workload-C. By enabling profile-guided placement for those functions, Owl-PG manages to mitigate their degradation with an average increase of 9.24% in VM time. The increase mainly comes from a lesser extent of overcommitment for those functions. Finally, the sandbox consolidation in Owl further reduces the VM cost by 8.53%.

Table 3: Example sandbox placement under different scheduler. The last column is the sum of allocated memory.

	VM ID	Function Instances	Alloc.
Base	vm-pf1ane	RS×4, FL, GMM×2, QV, ID	3072M
	vm-t5f9i0	RS×7, GMM×4	2304M
	vm-g3zjov	RS×8, FL	3072M
Base-OC	vm-ignsdv	RS×14, CA×2, AL×1, GMM	5248M
	vm-ldnhoz	RS×9, CA×1, AL×2, GMM×5	4736M
	vm-t6xcmn	RS×16, GMM×3	4480M
Owl	vm-ncdwjl	RS×10, CA×6	4096M
	vm-32h6fo	RS×10, CA×6	4096M
	vm-y9ho0u	RS×7	1792M

We validate how mitigation works by closely inspecting the sandbox placement on VMs. From the experiment logs of Workload-A, we sample several VMs to reveal how different schedulers place the sandboxes of RS function. Table 3 lists the snapshots of three VMs for each variant, which showcase the behavioral traits of the schedulers in comparison. In terms of memory utilization, Base-OC is the highest as its ratio of allocated memory to machine limit (i.e., 3072M) is the largest. Meanwhile Base strictly allocates within the limit. Owl strikes a balance because RS is configured as profile-enabled function and its profile limits the overcommitment. As for function performance, Base-OC disregards the latency and tightly collocates RS with other functions,

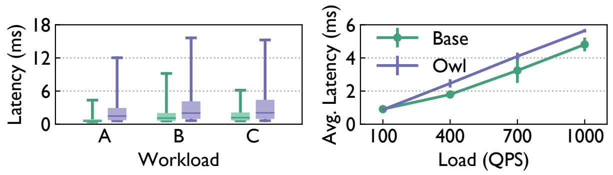

Figure 8: *Left*: The scheduling latency (5, 25, 50, 75 and 95-th percentile) in the end-to-end benchmarks. *Right*: The average latency under different load.

some of them compute-intensive. The uncontrolled overcommitment causes the degradation in the P95 latency. The profile-guided placement in OWL prevents the degradation and sometimes the tail latency is even lower than BASE. The top row in Table 3 explains the reason: BASE may collocates RS with other interfering functions. As a comparison, OWL only selects the functions compatible in performance.

7.3 Scheduling Latency

The fact that FaaS requests are usually short-lived highlights the importance of timely scheduling in FaaS. We thus examine the scheduling latency of OWL in the end-to-end experiments. We define the latency as the time of placement and routing decision and it does not include the sandbox creation. Figure 8 (left) plots the latency percentiles across three workloads. In general, OWL schedules at a similar speed as the BASE, their median latency being 0.959ms and 1.853ms respectively. The difference is much shorter than the creation time of a sandbox, thus having no significant effect. The longer average latency mainly comes from the checking of sandbox group.

We further evaluate how the scheduler scales with the increasing QPS through a controlled experiment. Both BASE and OWL receive a request flow of varying speed and we record how their average latency changes. They demonstrate a similar level of scalability in Figure 8 (right). It is worth mentioning that the system in experimentation is adapted from the production system and it includes various other components, such as billing management, plugin instrumentation, etc. They add extra overhead and constrain the system scalability. Hence a meaningful interpretation is that OWL does not add extra computation complexity and has the similar scalability with the original scheduler.

7.4 Validating Degradation

The comparative validation (§4.1) enables OWL to distinguish the problems of execution environment from those of other factors. We demonstrate its effectiveness with a microbenchmark study which simulates the degradation of a

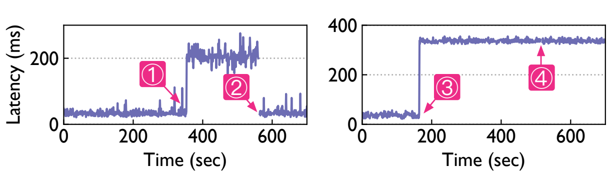

Figure 9: We demonstrate the comparative validation with two types of degradation: collocating CPU thrashing program (left) and inserting delay to external dependency (right).

data-processing function with an external database dependency. In the first case, we launch CPU thrashing programs on the VM to emulate a crowded execution environment. In the second case, we insert 300ms latency to the database service to simulate an external problem. The performance rule is set as "the average of latency should be less than 100ms". Figure 9 records how the system reacts to the latency change.

On the left side, the thrashing program causes the function latency to jump to 200 ms (①). After a period of time, the watchdog detects the degradation and migrates the sandbox to a control VM. Its performance immediately resumes to normal (②), which makes the watchdog conclude that the execution environment causes the degradation. In the right figure, a similar surge happens when we slow down the external service and the latency increases by 300ms (③). Similarly, the watchdog notices the change and migrates the sandbox. However, it finds that the latency is still degraded (④) and thus attributes the problem to the function itself.

7.5 System Overhead

The overhead of OWL comes from two sources: the comparative validation in performance monitoring and the sandbox migration in placement update.

Validation We simulate the validation cost with the production trace used in §2.2. We consider the popular and non-popular functions separately because they have different validation techniques. For popular functions, they are suitable for *canary comparison* (§4.1) and each requires one `control` machine (i.e., non-overcommited) to serve as a comparison baseline. We define popular functions as those with over 10 VMs. We add one extra VM for each function throughout the trace and calculate the total machine time. The result increases by 0.254% from the original time in the trace. The negligible increase has two reasons. First, the popular functions only account for a small proportion, 12.34%, in our platform. Hence the added VMs have a relative small scale. Second, the popular functions already occupy a large number of VMs, with top ones easily taking hundreds or thousands. They eclipse the cost of the added extra VM.

We then calculate the additional cost for non-popular functions, assuming they use *temporal comparison*. Mohammad et al. [36] report that only 20% of applications have invocations more frequent than once per minute. Those less frequent functions require no comparison because it takes a long time to accumulate a number of requests that have statistical significance. Hence we assume the validation is mainly for the top 20% functions. We similarly simulate that we add another one VM for each of them. In our trace, the resulting VM number only rises by 0.646%. Therefore, we argue the comparative validation incurs negligible overhead.

Migration We calculate the migration cost in Workload-A (§7.2) and its ratio to the overall VM cost. In profile-guided placement (§5.2), if no placement cache exists, the scheduler first places the sandbox separately and then migrates it to a more suitable place. During the migration, an extra sandbox is created and coexists with the original sandbox, thus increasing the total resource usage. By adding up all the migration time in Workload-A, we find the total only accounts for 0.106% of the total VM cost. The reason behind could be understood with a close inspection of the sandbox life cycle. The creation of a typical sandbox lasts for hundreds of milliseconds. But the keep-alive mechanism adds a retention period of five minutes to the life cycle. That means a single migration will incur at most 0.167% increase in the lifetime of a sandbox. Therefore we argue that the migration only causes negligible overhead.

7.6 Large-Scale Validation

We finally evaluate our prototype with a longer time scale. Two systems (Base and Owl) are deployed side by side in separate clusters. From the production logs, we sample ten eight-hour traces from ten different users. Then we match them with the functions in our benchmark suite (Table 2). We simulate the scenario where CA and GMM are two latency-critical functions and require their performance to be unaffected by overcommitment. They are configured as profile-enabled in Owl. Figure 10 shows the experiment results. As this experiment skips prewarming, the VM number surges at the beginning. After that, Owl utilizes consistently fewer VMs than Base and their VM time differs by 37.50%. The latency distribution shows that CA and GMM are well protected by the profile-guided placement and exhibit no sign of degradation.

8 DISCUSSION

System Deployment The centralized placer does not restrict scalability because we have a replication mechanism in production deployment. It ensures scalability and fault tolerance by letting each replica handle a portion of user functions. Their underlying VMs are not overlapped to avoid potential conflict. The functions are dynamically partitioned among replicas and they will be readjusted when one replica fails. The details on the replication is beyond the scope of this paper.

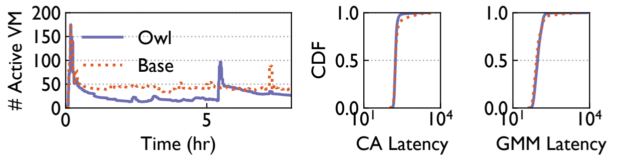

Figure 10: We validate Owl in a production-like environment for eight hours. The CA and GMM are treated as latency-critical functions.

Isolation Techniques The FaaS platform can be deployed over bare-metal or virtual machines, isolating functions with secure container [21] or micro VM [5]. The isolation techniques barely affect our design as long as they support resource limiting and dynamic sharing, which is the basic requirement of resource overcommitment. But Owl is not directly applicable to the bare-metal machines due to their large size. The collocation profile is impractical to fill up a 128GB machine with a single type of functions. When a host has thousands of functions, it is challenging to manage them.

System Scalability Owl can scale with the number of functions thanks to some of its special design. (1) The profile-guided placement only requires each function to be paired with several others, instead of all the rest. Hence the complexity of profiling will not increase with the functions. In fact, Algorithm 1 can operate with an arbitrary number of collocation pairs in the profile. (2) The place-then-update scheme and placement cache (§5.2) in the placer ensure a quick initial decision for each sandbox and prevents the cold start latency from increasing with more functions.

Design Limitations Targeting a commercial platform instead of an open-source one, Owl has embraced several limitations. First, it has to respect the resource quota set by the users. The schedulers such as ENSURE [39] cannot work in our setting because it redistributes resources among functions. The overcommitment in Owl is feasible because it does not factually reduce the resource allocation for any function. The performance guarantee ensures that the functions can get sufficient resources whenever they need. Hence the overcommitment is equivalent to the functions sharing intermittently idle resources. Second, Owl cannot arbitrarily execute user functions during the profiling period. Hence it cannot adopt more advanced techniques [18, 19] that profile how applications react to interference or collocation. Those scenarios could be difficult to set up or lead to performance degradation in our setting.

9 RELATED WORK

Scheduling Serverless Functions §2.1 summarizes the FaaS scheduling into two operations: sandbox placement and request routing. Owl mainly optimizes the former, so do ENSURE [39] and its subproject FnSched [38]. However, they apply to general instead of commercial FaaS platform and their approach violates the constraint of a public platform. They redistribute the CPU share among sandboxes on a single VM to enhance utilization, but public FaaS has to respect the resource quota set by the users. Hermes [30] designs hybrid policies for request routing, but it also cannot apply to our platform. Its policy allows requests to be queued while our platform has to process them immediately by either assigning to an idle sandbox or creating a new one.

FaaS scheduling has been optimized from other aspects besides placement and routing. One of them is the cold start during function boot-up. FaaSNet [41] accelerates it for functions with custom images; SEUSS [12] proposes a unikernel-based architecture for rapid boot-up; Shahrad et al. [36] prewarms containers to avoid cold start. Owl is orthogonal to their optimization and can be deployed with them. Another optimization approach is to search for more resource-efficient configuration for functions. Sizeless [20] and OFC [34] predict the optimal memory allocation with machine learning, and COSE [6] with Bayesian optimization. Their function profiles can be integrated into the profile-guided placement of Owl.

Scheduling Serverless DAGs Beyond the granularity of individual functions, many works tackle the problem of scheduling a DAG of functions. To optimize execution time, Xanadu [16] finds the most likely path in each DAG and prewarms the containers to avoid cold start; Atoll [37] partitions the cluster to shorten the scheduling latency for DAG applications; Sequoia [40] prevents unexpected platform bugs during the execution; Pheromone [45] improves the data locality during DAG execution using a data-centric function orchestration approach. To enhance resource utilization, ORION [33] rightsizes the resource allocation for each DAG; Kraken [10] adapts the container provision within the DAG. The works above mainly target general DAG applications, and there are other works designed for specialized applications. For instance, Caerus [49] balances the completion time and resource cost of analytics workload; SONIC [32] optimizes for the DAGs that require data passing. Owl can be integrated into those DAG-oriented systems to further reduce resource waste on a container level.

Container Scheduling When a FaaS sandbox receives requests continuously, it resembles a long-running applications (LRA). Yet the existing scheduling techniques for LRA are unsuitable for FaaS because of the workload dynamicity and the latency overhead. For instance, Medea [22] presumes each container is associated with a set of manually specified placement constraints. Metis [43] and Paragon [17] learn the placement preference from the simulated and profiled traces respectively. FaaS sandboxes are less controllable than their assumptions. Furthermore, the scheduling latency of the previous three solutions is on the order the minutes, which is intolerable in FaaS platform.

Task Scheduling When the FaaS scheduler decides which sandbox to serve the request, it behaves similarly as the computing task scheduler, such as Sparrow [35] and Quincy [28]. Nevertheless the retention of sandboxes renders the task schedulers problematic in FaaS scenario. If we view each request as a task and directly reuse the task scheduling design, the sandboxes that are retained after each request may cause the resource fragmentation.

10 CONCLUSION

In this paper, we have presented Owl, a performance-aware scheduler for public FaaS platform. It supports a flexible rule format that allows users to specify customized performance rules for different functions. For less popular functions, it places their sandboxes with usage-based allocation, keeps monitoring their performance, and remedies any detected degradation. If the degradation persists, it applies comparative validation to confirm the source of the cause. For popular functions, Owl profiles their collocation by adjusting the sandboxes in the production environment. The collocation profile guides the system to generate a contention-free and resource-efficient placement of the sandboxes. It further consolidates the idle sandboxes within those placement to further reduce resource waste. The usage-based allocation and the resource-tight profile achieve the objective of high utilization; the monitoring mechanism and the contention-free profile satisfy the performance requirements. We prototype Owl in our production system and implement a representative benchmark suite to evaluate it. The results demonstrate the effective of Owl in reducing operation cost and ensuring function performance.

REFERENCES

[1] 2022. Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.
[2] 2022. AWS Lambda. https://aws.amazon.com/lambda/.
[3] 2022. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[4] 2022. Google Cloud Functions. https://cloud.google.com/functions.
[5] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *NSDI*.
[6] Nabeel Akhtar, Ali Raza, Vatche Ishakian, and Ibrahim Matta. 2020. COSE: configuring serverless functions using statistical learning. In *IEEE Conference on Computer Communications (INFOCOM)*.

[7] Anonymized. 2021. Private Communication. Interview.
[8] Noman Bashir, Nan Deng, Krzysztof Rzadca, David Irwin, Sree Kodak, and Rohit Jnagal. 2021. Take it to the limit: peak prediction-driven resource overcommitment in datacenters. In *EuroSys*.
[9] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.".
[10] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In *SoCC*.
[11] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. 2018. Rock you like a hurricane: taming skew in large scale analytics. In *EuroSys*.
[12] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *EuroSys*.
[13] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *NSDI*.
[14] Alibaba Function Compute. 2022. Manage functions. https://www.alibabacloud.com/help/en/function-compute/latest/manage-functions.
[15] Alibaba Function Compute. 2022. Metrics (See "CPU utilization - FunctionCPUQuotaPercent"). https://www.alibabacloud.com/help/en/function-compute/latest/metrics.
[16] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. 2020. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Middleware*.
[17] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ASPLOS*.
[18] Christina Delimitrou and Christoforos E. Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. *ASPLOS*.
[19] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *SoCC*.
[20] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the optimal size of serverless functions. In *Middleware*.
[21] TheOperstack Foundation. 2017. Kata Containers: The speed of containers, the security of VMs. https://katacontainers.io/.
[22] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *EuroSys*.
[23] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*.
[24] Google. 2018. gvisor: Container runtime sandbox. https://github.com/google/gvisor.
[25] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *SIGCOMM*.
[26] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *OSDI*.
[27] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An analysis of Facebook photo caching. In *SOSP*.
[28] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *SOSP*.
[29] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. 2020. Improving resource utilization by timely fine-grained scheduling. In *EuroSys*.
[30] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. 2022. Principled and Practical Scheduling for Real-World Serverless Computing. *SoCC* (2022).
[31] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *CLOUD*.
[32] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *ATC*.
[33] 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 *OSDI*.
[34] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, et al. 2021. OFC: an opportunistic caching system for FaaS platforms. In *EuroSys*.
[35] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *SOSP*.
[36] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, et al. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *ATC*.
[37] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A scalable low-latency serverless platform. In *SoCC*.
[38] Amoghvarsha Suresh and Anshul Gandhi. 2019. FnSched: An Efficient Scheduler for Serverless Functions. In *Proceedings of the 5th International Workshop on Serverless Computing*.
[39] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi. 2020. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*.
[40] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling quality-of-service in serverless computing. In *SoCC*.
[41] 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 *ATC*.
[42] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *ATC*.
[43] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. 2020. Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale. In *SC*.
[44] AWS Whitepaper. 2017. Choosing the Optimal Memory Size. https://amzn.to/3k9bVMj.
[45] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *NSDI*.
[46] Minchen Yu, Yinghao Yu, Yunchuan Zheng, Baichen Yang, and Wei Wang. 2020. RepBun: Load-Balanced, Shuffle-Free Cluster Caching for Structured Data. In *IEEE INFOCOM*.
[47] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, et al. 2020. Characterizing Serverless Platforms with Serverlessbench. In *SoCC*.
[48] Yinghao Yu, Renfei Huang, Wei Wang, Jun Zhang, and Khaled B. Letaief. 2018. SP-Cache: Load-balanced, Redundancy-free Cluster Caching with Selective Partition. In *IEEE/ACM SC*.

[49] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *NSDI*.

Algorithm 3 Place sandboxes according to function profiles.

```
1: $m_i, m_j$: Configured memory size of function $i$ and $j$.
2: $M$: Memory limit of a VM.
3: function PROFILE($m_i, m_j, M$)
4:     $s_i, s_j \leftarrow \lceil M/5m_i \rceil, \lceil M/5m_j \rceil$
5:     $P \leftarrow []$
6:     Append $(\lfloor M/2m_i \rfloor, \lfloor M/2m_j \rfloor)$ to $P$.
7:     $n_i, n_j \leftarrow \lfloor M/2m_i \rfloor, \lfloor M/2m_j \rfloor$
8:     while $n_i > \lfloor M/4m_i \rfloor$ do
9:         $n_i, n_j \leftarrow n_i - s_i, n_j + s_i \cdot m_i/m_j$
10:        Append $(n_i, n_j)$ to $P$.
11:    $n_i, n_j \leftarrow \lfloor M/2m_i \rfloor, \lfloor M/2m_j \rfloor$
12:    while $n_j > \lfloor M/4m_j \rfloor$ do
13:        $n_i, n_j \leftarrow n_i + s_j \cdot m_j/m_i, n_j - s_j$
14:        Append $(n_i, n_j)$ to $P$.
15:    $res \leftarrow (0, 0)$
16:    while $P$ is not empty do
17:        $n_i, n_j \leftarrow$ first pair popped from $P$
18:        Place $n_i$ sandboxes of function $i$ and $n_j$ of function $j$.
19:        if no degradation is detected then
20:            Set $res$ as $(n_i, n_j)$ if the allocation ratio is the highest.
21:            Append $(n_i + 1, n_j), (n_i, n_j + 1)$ to $P$.
22:    return $res$
```

A PROFILING ALGORITHM

The collocation profiles are obtained by adjusting sandboxes on a VM and monitoring their performance (§5.1). Algorithm 3 describes how the sandbox quantity is adjusted. The basic intuition is to increase the both types of function instances in the order of iterative deepening search (IDS), e.g., $(a, b), (a + 1, b), (a, b + 1), (a + 2, b), (a + 1, b + 1), (a, b + 2)$.... Line 6-14 calculate several initial combinations. Line 15-22 iteratively increases the quantity in each pair and checking which combination leads to the highest utilization. Line 4 sets the step size of each increase so that each step will increase memory allocation by 20%. It ensures the profiling can finish within reasonable steps

B BENCHMARK SUITE

Table 4 and 5 describe the benchmark functions in detail. The benchmark is designed to span the three most popular domains in our production environment: backend hosting, multi-media processing and IoT data analytics. We select five real-world applications from existing customers and implement part of their functionality.

Table 4: Description of benchmark functions.

Name	Description
Smart Parking[6]	An application that manages smart parking sites and provides a mobile app to users.
Query Vacancy	Users query the vacancy of a specific spot.
Reserve Spot	Users reserve a specific parking spot.
Log Processing	A storage service provider processes its logs according to regulation requirement.
Anonymize Log	Anonymize the sensitive personal information contained in the log.
Filter Log	Filter the logs and store it to object storage.
Computer Vision	A CV company provides image processing service through API.
Detect Object	Detect whether an image contains a certain object.
Classify Image	Classify the category of a given image.
Video Processing	A content media processes its video content uploaded by the users.
Get Media Meta	Extract the meta information contained in the video.
Convert Audio	Convert the audio contained by the video.
Smart Manufacturing	A factory processes its sensor data and display anomaly on monitors.
Ingest Data	Ingest the data generated from the sensor network.
Detect Anomaly	Detect the anomaly patterns contained in the sensor data.

Table 5: Attributes of benchmark functions.

Abbreviation	Function	Memory Size	Actual Usage	Lanaguage	Dependencies
QV	Query Vacancy	256 MiB	~70MiB	JavaScript	Key-Value Store
RS	Reserve Spot	256 MiB	~70MiB	JavaScript	Key-Value Store, Message Queue
AL	Anonymize Log	1024 MiB	~20MiB	Rust	Message Queue
FL	Filter Log	1024 MiB	~20MiB	Rust	Message Queue
DO	Detect Object	3072 MiB	~1700MiB	Python	Model Serving Framework
CI	Classify Image	2560 MiB	~500MiB	Python	Model Serving Framework
GMM	Get Media Meta	128 MiB	~20MiB	Python	Object Store
CA	Convert Audio	256 MiB	~100MiB	Python	Object Store
ID	Ingest Data	768 MiB	~10MiB	C++	SQL Database
DA	Detect Anomaly	768 MiB	~10MiB	C++	SQL Database