Golgi: Performance-Aware, Resource-Efficient Function Scheduling for Serverless Computing

Suyi Li
HKUST
slida@cse.ust.hk

Wei Wang
HKUST
weiwa@cse.ust.hk

Jun Yang
WeBank
jonyang@webank.com

Guangzhen Chen
WeBank
cgzchen@webank.com

Daohe Lu
WeBank
leslielu@webank.com

ABSTRACT

This paper introduces Golgi, a novel scheduling system designed for serverless functions, with the goal of minimizing resource provisioning costs while meeting the function latency requirements. To achieve this, Golgi judiciously overcommits functions based on their past resource usage. To ensure overcommitment does not cause significant performance degradation, Golgi identifies nine low-level metrics to capture the runtime performance of functions, encompassing factors like request load, resource allocation, and contention on shared resources. These metrics enable accurate prediction of function performance using the Mondrian Forest, a classification model that is continuously updated in real-time for optimal accuracy without extensive offline training. Golgi employs a conservative exploration-exploitation strategy for request routing. By default, it routes requests to non-overcommitted instances to ensure satisfactory performance. However, it actively explores opportunities for using more resource-efficient overcommitted instances, while maintaining the specified latency SLOs. Golgi also performs vertical scaling to dynamically adjust the concurrency of overcommitted instances, maximizing request throughput and enhancing system robustness to prediction errors. We have prototyped Golgi and evaluated it in both EC2 cluster and a small production cluster. The results show that Golgi can meet the SLOs while reducing the resource provisioning cost by 42% (30%) in EC2 cluster (our production cluster).

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624645

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Serverless Computing; Resource Management; Scheduling

ACM Reference Format:
Suyi Li, Wei Wang, Jun Yang, Guangzhen Chen, and Daohe Lu. 2023. Golgi: Performance-Aware, Resource-Efficient Function Scheduling for Serverless Computing. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624645

1 INTRODUCTION

Serverless computing, also known as Function-as-a-Service (FaaS), offers a compelling cloud model that greatly simplifies the development and deployment of cloud applications [16, 33]. In a serverless platform, users compose applications as a set of loosely-coupled cloud functions and let the platform take care of their provisioning, scaling, logging, and fault handling. Serverless computing not only relieves users of the server management burden, but is also attractive with its *pay-per-use* pricing model, under which users are only billed when their functions are running [19].

Critical to the management of serverless platforms is function scheduling. In a serverless platform, functions are running in sandboxes, e.g., containers [7] and microVMs [1]. When invocation requests arrive, the scheduler routes them to available function instances for execution. If no instance is currently available, new instances are launched on some selected servers to (horizontally) scale out the functions [7, 21, 37]. As the number of function instances changes, servers (e.g., virtual machines (VMs)) are dynamically added to or removed from the platform. To minimize the provisioning cost, the scheduler should pack function instances to as few servers as possible [7, 37]. In the meantime, this should not be achieved at the expense of degraded function performance

(i.e., extended latency [7, 24, 34, 35, 46].), which is critically determined by the scheduling quality [24, 34, 35, 37].

However, achieving these two objectives is challenging. Recent studies reveal considerable allocation wastes in commercial FaaS platforms: on average, functions only use around 25% of the requested resources [26, 30, 35, 37]. To improve utilization, many platforms choose to overcommit functions based on their past resource usage instead of the specified requests [7, 37]. Yet, blindly performing overcommitment often results in severe resource contention between colocated functions, extending their tail latency by up to 3× in our experiments. To tackle this problem, existing scheduling systems employ a *performance-aware approach* that predicts the function performance based on resource configurations [24] or collocation profiles [37] and uses the predicted results to guide scheduling. These systems, however, either overlook the performance impact of function collocations (e.g., Orion [24]) or do not scale in complex collocation scenarios (e.g., Owl [37] only considers collocation of two functions), limiting their applications in large platforms.

In this paper, we present a new FaaS scheduling system, named Golgi[1], that judiciously overcommits functions for reduced provisioning cost while still meeting their latency requirements. Similar to the existing work and production practices [24, 26, 35, 37, 46], Golgi concerns the tail performance and allows users to specify their SLO (service-level objective) requirements as the target latency at a certain percentile, e.g., the P95 latency should not exceed 200 ms. For each function, Golgi maintains two kinds of instances: (1) the non-overcommitted (non-OC) instances with resource configurations following the user specifications and (2) the overcommitted (OC) instances with resource configurations downsized based on the usage of past function executions [37]. In general, non-OC instances yield the best latency performance but may result in significant resource underutilization, whereas OC instances are resource-efficient yet susceptible to performance slowdown, especially under poor scheduling decisions. For each invocation request, Golgi dynamically decides which OC or non-OC instance should it be routed to (if necessary, creating a new instance on a certain server) with the goal of minimizing the resource provisioning cost while meeting the function's latency SLO. Golgi uses three techniques in achieving this goal.

Identifying low-level runtime metrics to characterize the function performance. Unlike existing approaches that establish a static map between the function performance and resource configuration (or collocation), Golgi identifies nine low-level metrics and uses them to dynamically capture the runtime performance of functions. These metrics cover a variety of sources that impact the function execution, including request load within a function instance (e.g., inflight requests), resource allocation (e.g., CPU and memory), and contention on shared resources (e.g., network and CPU cache). They are easy to collect in production clusters with little overhead, and collectively indicate if the request execution can complete within the specified latency, using machine learning (ML) techniques as explained below.

Performance-aware scheduling. Golgi dynamically collects the nine metrics for each OC function instance, and uses a classification model to predict if its runtime performance can meet the SLO requirement. To maintain high prediction accuracy, Golgi continuously updates the model as requests arrive, without collecting a large amount of training data via offline profiling, which is usually infeasible in practice. We choose the Mondrian Forest [18], a random forest model, as the classification model, for it gives superior performance in online learning. More specifically, we design an *online stratified sampling algorithm* to maintain balanced data samples between the two classification results in model updates.

With the model in place, Golgi schedules requests following a *conservative exploration-exploitation* approach. By default, Golgi routes requests to non-OC instances for satisfactory performance (exploitation). In the meantime, it actively explores opportunities of using more resource-efficient OC instances for reduced provisioning cost, provided that the resulted performance still meets the SLO. Golgi predicts the request response latency given by the candidate OC instances and makes scheduling decisions accordingly. To avoid the long scheduling delay caused by model inference, Golgi lets each function periodically predict its performance for OC instances and cache the prediction results on each worker node. These results are returned to the scheduler upon query, thereby taking the inference off the critical path. The prediction occurs frequently to avoid staleness.

Vertical scaling. Golgi also performs *vertical scaling* to dynamically adjust the *concurrency* of each OC instance with atomic operations, which specifies the maximum number of requests that can be served concurrently. Vertical scaling provides flexibility and scalability in accommodating varying serverless workloads [7]. It not only maximizes the request throughput to improve resource efficiency, but also improves the system robustness as it can quickly react to the scheduling mistakes caused by occasional mispredictions given by the ML models. Our vertical scaling is specifically designed for serverless functions as it supports frequent scaling without service downtime [14]. It hence complements scheduling by reducing the performance degradation risks, encouraging more aggressive exploration that results in 15% VM cost reduction in our experiments.

[1]Golgi apparatus is an organelle in cells. It packages proteins and sends them to destinations. We use it as a metaphor for our routing system.

We have prototyped Golgi as a pluggable scheduler in OpenFaaS [11] with Kubernetes. We evaluate Golgi in an EC2 cluster with a suite of serverless applications that cover a range of real-world business scenarios [37]. Compared to Orion [24], the state-of-the-art serverless scheduling system, Golgi reduces the resource provisioning cost by 42% while still meeting the function SLOs (§8.2). We also deployed Golgi in a small production cluster and measured 30% cost savings (§8.7). Golgi demonstrates strong scalability, capable of handling large load spikes (over 5100 requests per second as reported in Azure Function traces [34]) and supporting diverse functions with little overhead (§8.6).

2 BACKGROUND AND MOTIVATION

2.1 Serverless Computing Background

Serverless computing (*a.k.a.*, FaaS) has emerged as a popular cloud computing paradigm for cloud platforms and tenants. It allows developers to write short-running, stateless functions to deploy their applications that can be invoked by various triggers [33, 34]. Serverless platforms take over the responsibility of resource provisioning, function orchestration, auto-scaling, and quality assurance for serverless applications [37]. They maintain an inventory of servers (VMs) to host users' function instances as sandboxes, e.g., containers. The instances are placed on the servers for execution. As a result, serverless users can concentrate on the development of their applications and products, without the need to worry about server operation and management. Besides, serverless computing appeals to users with a pay-as-you-go billing model, whereby users are only charged for the resources used to execute their functions at a millisecond granularity [4, 9, 19]. Platforms bear the cost of idle resources.

Function resource configuration. While deploying functions, serverless users should claim a memory size to configure their functions in cloud platforms [33]. Other resources, e.g., CPU power and network bandwidth, are allocated proportionally to the memory size. This billing model is widely adopted by most FaaS providers [4, 9, 19].

FaaS scheduling workflow. While users can effortlessly deploy and run their functions, FaaS providers are responsible for managing the virtual machines and scheduling functions on them. Given a function invocation request, the FaaS scheduler *routes* it to one of the available function container instances. If multiple instances are available, the scheduler selects one of them using a routing algorithm. The default *request routing* algorithm is the most-recently-used (MRU) algorithm [26, 37, 40]. It is a greedy algorithm that prioritizes the most recently used instances to serve requests. MRU's priority setting strategy complements FaaS's *keep-alive* mechanism, which allows a function instance to stay idle for minutes before terminating [34]. As MRU routing leads to a longer idle time for the least invoked function instances, their resources can be reclaimed sooner.

If no instance is currently available for *routing*, the scheduler will create one and place it on suitable servers to scale out the function [7, 21]. Instance placement usually follows the First-Fit bin packing algorithm [37, 40] to minimize the number of servers.

Scheduling objectives. Scheduling is a critical aspect of serverless platforms, as it affects both the operation cost of the platform and the service quality experienced by users. The scheduling objective is twofold: (1) to meet users' function performance requirements, such as tail latency, as suggested by recent research and production practices [24, 26, 35, 37, 46], as functions are online user-facing services, and (2) to pack function instances tightly on as few servers as possible to reduce resource provisioning costs [7, 37].

2.2 Motivations

We describe the challenges in meeting scheduling objectives.

Under-utilized resources. Recent production traces and workloads [30, 35, 37] show that users tend to over-claim resources for their functions. For instance, a survey of AWS Lambda usage reports that 54% of function instances are configured with 512 MB or more but the average and median amounts of used memory are actually 65 MB and 29 MB [26, 30], respectively. In the AliCloud Platform, most function instances only use 20–60% of the allocated memory [37].

There are two main reasons for resource under-utilization. First, users intentionally over-claim a large memory size for the sake of more CPU resources for their functions because of the fixed CPU-to-memory ratio [37]. Second, the keep alive policy allows the resources allocated to function instances to remain unused for long time intervals [26, 37]. Our analysis of Azure Function Trace [34] shows 91.7% of the functions are invoked once per minute or less on average. As the keep alive periods usually last for minutes and 96% of the function executions take at most 60s on average, the memory waste from the keep alive policy can be prohibitively high.

Resource down-sizing can degrade performance. One approach to eliminate resource under-utilization is to overcommit resources by downsizing function resource configurations based on their past resource usage. However, this may degrade function performance. As function performance improves monotonically with more resources allocated [24], the expected performance baseline of a function is the result of allocating all claimed resources and ensuring their commitment. Based on this baseline, users can customize their performance expectations, as described in [37]. Blindly down-sizing functions' resource configurations to host more

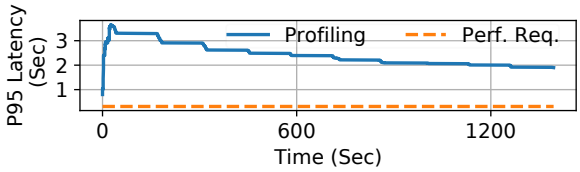


Figure 1: Get-Media-Meta performance in profiling.

function containers on a single server, i.e., resource overcommitment, can intensify resource contention and degrade performance. Our experiments in §8.2 show that this can increase P95 latency by 2.8×.

Worse, resource overcommitment leads to more function containers being collocated within a server, causing severe collocation interferences [12, 22, 37, 41, 45]. Resources, e.g., network I/O and CPU cache, that are not managed by the OS kernel cannot be isolated properly, potentially impairing function performance due to shared resource contention. In §3.1, we show the interferences increase P95 latency by 2.5×.

Practical constraints. In serverless platforms, it is impractical to *offline* profile overcommitted functions to determine their performance. This is because users's functions are typically stateless [44] and rely on external services, e.g., message queues and databases, for storage and coordination [37]. Invoking functions without users' permission may cause data inconsistencies or corruption, as external dependencies are stateful. This practical constraint hinders any serverless platforms from *offline* profiling users' functions for resource optimization [2, 7, 8, 10, 24, 26]. Moreover, the scheduling process should be efficient as it is required that the scheduling latency should be less than 20ms in AWS Lambda [7]. Last, the design should be scalable enough to serve thousands of functions in platforms with little overhead [33].

2.3 Existing Methods

Naive overcommitment. Resource overcommitment is a straightforward approach to reducing resource waste, i.e., placing more function instances in a server. This can be achieved by down-sizing instances' resource requests. Suppose a function instance claims cMB memory but actually takes aMB memory in runtime. We change its memory request from c to $\alpha \times c + (1.0 - \alpha) \times a; \alpha \in [0, 1.0]$. Other resources are adjusted proportionally. The new resource configurations based on actual usage are called usage-based overcommitment in [37], which is especially valuable for workloads with underutilized resources [13]. The hyperparameter α controls the size of slack memory that prevents out-of-memory error.

Though usage-based overcommitment can significantly reduce resource waste, as discussed in §2.2, naive overcommitment degrades function performance due to more intensive resource contention and collocation interferences.

Profiling-driven scheduling. Users' functions rely on external dependencies for storage and coordination, preventing platforms from *offline* profiling them. Thus, general profiling-and-scheduling approaches fall short [2, 7, 8, 10]. We consider approaches that can profile functions online with the user's invocation requests.

1) ***Profiling for right-sizing.*** Orion [24] proposes a right-sizing design that finds the best resource configuration for a function to meet its latency SLO at minimum cost. It strategically profiles function performance with a wide range of memory configurations to decide the right memory size.

However, such profiling methods have limitations. Firstly, profiling functions with a wide range of memory sizes can violate SLO because not all of them can provide satisfactory performance. We re-implement the profiling process and profile a get-media-meta function in Fig. 1, which is one of the benchmark functions used in our evaluation (§8.1). The profiling takes about 25 minutes to find the best memory configuration. The solid line represents the change in functions' tail latency throughout the profiling, much longer than the performance requirements (dashed line). Worse, re-profiling is needed when the workload characteristics change, which can take up to 3.5 hours, depending on the invocation frequency [24]. Second, such profiling methods overlook the collocation interferences among function instances, making the right-sized functions underperform. In Fig. 5, our evaluation shows Orion hardly meets the functions' SLOs (§8.2). Similar results are also reported in Orion's original paper [24], where the right-sizing strategy increases the P95 latency by 2.32× compared to their performance objective.

2) ***Profiling for overcommitment.*** Owl [37] designs a new data structure called collocation profile to record any *two* function types that can be collocated tightly without performance degradation. The profile records the number of instances of each function type such that their instances can be collocated within a single server. The profiles capture the interferences between the collocated functions and guide the scheduler to place instances as tightly as possible for resource overcommitment. Building a collocation profile requires performance profiling of possible collocations.

However, Owl only considers the collocation of two different functions in small-size VMs (2 CPU cores and 4 GB memory) [37]. Trivially extending the approach to consider the collocation of multiple functions in large VMs increases the number of performance profiling significantly. Following their profiling procedures in [38], searching a profile that collocates N function instances of M different function types requires profiling at most $\sum_{n=1}^{N} \binom{(M+n-1)}{(M-1)}$ collocations. Consider a representative profile that collocates 16 instances of two different function types in [37]. If we use a server with 36 CPU cores and 72 GB memory that can host 288 such

instances, the number of collocation performance profiling will increase by 275×. Extending the profile to 3 different functions will increase the number of profiling by 26, 742×.

2.4 Design Requirements

In light of the scheduling objectives (§2.1) and challenges (§2.2) in serverless computing, we have the following requirements in the design of a FaaS scheduling system.

Performance-aware. Schedulers should be aware of function performance to meet performance requirements.

Resource-efficient. The scheduler should pack function instances tightly on servers to improve resource utilization.

Practically applicable. The design should meet the practical constraints, e.g, platform-initiated profiling, in §2.2.

Our solution. In the following sections, we proceed to develop Golgi systematically to fulfill the requirements outlined. In §3, we characterize functions and show that their performance can be predicted by nine low-level metrics. The metrics are function-agnostic and scalable to support diverse functions. We design Golgi's scheduler in §4, leveraging the metrics to make performance-aware scheduling decisions with resource overcommitment to save resource provisioning costs. In §5, we propose a vertical scaling design specifically for FaaS workloads. It adapts the configurations of overcommitted instances to their performance SLOs, reducing performance degradation risks from overcommitment.

3 CHARACTERIZING FUNCTIONS IN THE CLOUD

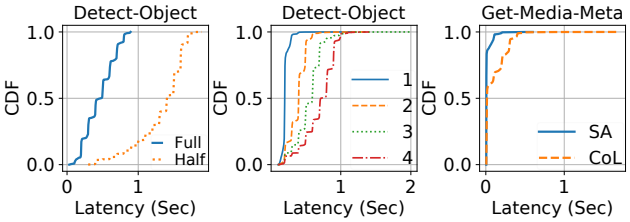
Critical to performance-aware scheduling is answering what affects function's performance in FaaS. In this section, we examine the characteristics of functions and determine that low-level runtime metrics can be utilized to forecast potential performance degradation in function instances. To obtain these informative metrics, the following practical requirements must be met in serverless platforms.

Accurate. The metrics should provide informative insights to predict function performance, accounting for the resource contention and collocation interferences in FaaS workloads.

Scalable. The metrics should be function agnostic to support the diverse function types in serverless platforms[34].

Non-intrusive. To uphold the integrity of users' functions in the platform, the process of collecting metrics must be non-intrusive, lightweight, and accessible *online*.

In §3.1, we carry out extensive controlled experiments to show the factors that contribute to the impacts of resource utilization and collocation interferences on function performance. In §3.2, we quantify these factors by nine low-level runtime metrics that can be collected non-intrusively using existing tools. The metrics are function-agnostic to accommodate diverse function types. In §3.3, we conduct validation experiments show that the metrics are informative and effective enough in predicting performance degradation.



(a) Resource (b) Concurrency (c) Collocation

Figure 2: Function characterizations.

Our characterizations are primarily based on OpenFaaS [11] with Kubernetes and AWS EC2 servers. The functions we characterized are from a suite of eight benchmark serverless applications in [37] that represent popular business domains (Table 1). These applications are implemented in different programming languages and have various external dependencies. Among them, SP and GMI have directed acyclic graph structures. SP has a chaining structure with two functions, query-vacancy (QV) and reserve-spot (RS), while GMI has a fan-out/fan-in pattern [6] with two functions, get-media-meta (GMM) and get-duration (GD).

Table 1: Benchmark applications.

Application	Dependency	Language
GMI: **G**et **M**edia **I**nfo	Object Store	Python, Go
SP: **S**mart **P**arking	Key-Value Store	JavaScript
DA: **D**etect **A**nomaly	Database	Go
ID: **I**ngest **D**ata	Database	Go
CI: **C**lassify **I**mage	TF Serving [27]	C++
DO: **D**etect **O**bject	TF Serving [27]	C++
AL: **A**nonymize **L**og	Message Queue	Rust
FL: **F**ilter **L**og	Message Queue	Rust

3.1 What determines function performance?

Firstly, function performance is a monotonically non-decreasing function of the resources allocated to it [24]. In addition, intra-container resource contention resulting from concurrent inflight requests in a single function instance [5, 9, 42] can exacerbate the performance. Beyond a single function instance, collocation interferences between multiple instances also impact function performance [12, 41, 45, 48]. Therefore, we characterize these factors by setting them as control variables and validating their effectiveness.

Resource configuration. Resource configuration determines the amount of resource allocated. Fewer resources lead to less

computing power and degraded performance. The CDFs of execution latency in Fig.2a confirm this. We deploy two sets of detect-object (DO) functions, which are CPU-intensive to provide model serving service. The first set is configured with 1.5 CPUs and 2.5GB Memory (blue solid line), while the second set is configured with 0.75 CPUs and 1.25 GB Memory (orange dotted line). We test the two sets separately for 1000 seconds with a constant RPS of 8. Clearly, instances with less resources result in a 212.8% P95 latency increase.

Request concurrency. Serverless platforms enable a single function instance to serve concurrent requests, saving costs by reusing resources [5, 9, 42]. However, higher concurrency means intensive resource contention within a container and degraded performance. The CDFs of execution latency in Fig. 2b confirm this. We deploy and test four sets of DO functions and set the maximum concurrency of each set to 1, 2, 3, and 4, respectively. Fig. 2b shows the tail latency monotonically increases with higher maximum request concurrency.

Inter-container interference. Collocation interferences between function instances in the same node impair their performance [12, 22, 41, 45]. Fig.2c confirms this issue. We set up two experiments by placing different sets of containers in a node. The first one tests a set of standalone get-media-meta (GMM) function instances with an RPS of 20 (blue solid line). GMM accesses remote object storage, processes files' metadata, and returns. The second one tests a set of collocated GMM and DO functions in a node with the same workload (orange dashed line). In Fig.2c, we see that the container collocation results for the increased tail latency of GMM, with P95 latency increases by 2.5×.

3.2 Capture Performance Metrics

We quantify the above factors with a handful of runtime metrics that can be accessed with little overhead. They are function-agnostic and scalable to support diverse functions. We classify the metrics into two categories: intra-container metrics, which reflect contention within a function instance, and collocation interference metrics, which reflect interference among collocated instances.

Intra-container metrics. CPU utilization, memory utilization, and the number of inflight requests are intra-container metrics, represented by a 3-D context vector *<CPU, Memo, Inflights>*. CPU and memory utilization can be obtained from the pseudo files managed by the *cgroup*. To record the number of inflight requests in a function instance, we set an atomic integer variable as a counter.

Collocation interference metrics. We consider the contention on the network I/O and CPU cache because most functions are stateless and rely on remote storage service [44].

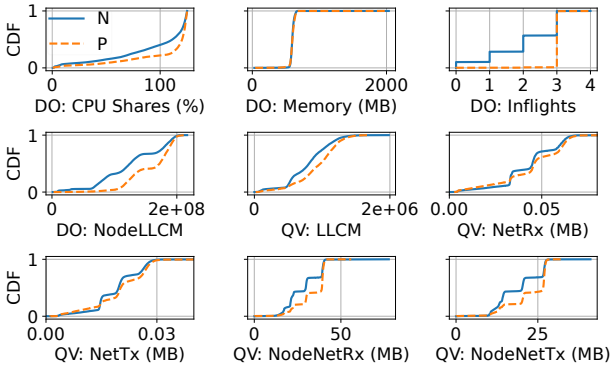


Figure 3: Metric visualization; N: negative; P: positive

1) ***Network.*** Network I/O is shared by collocated containers within a node. We measure the network traffic within a function container and the node it is on by the number of bytes received and transmitted, i.e., a 4-D vector *<NetRx, NetTx, NodeNetRx, NodeNetTx>*. The stats are collected from the Linux pseudo-files under the */proc/net/* directory.

2) ***CPU Cache.*** Collocated containers within the same node will contend the shared resources, e.g., the CPU last level cache (LLC) [12, 22, 45]. We collect the number of LLC-miss (LLCM) within a function container and the node it is on by running Linux *perf stat* tool, i.e., *<LLCM, NodeLLCM>*. As indicated in [43, 48], *perf stat* causes negligible overheads and no visible latency will be incurred.

Function-agnostic context vector. We finally construct a 9-D context vector by stacking the above metric vectors, revealing a container's runtime status. The fixed dimension makes the approach function-agnostic and scalable to support diverse function types.

3.3 Validation Tests

We design validation tests to verify that our metrics contain useful information that can be utilized to forecast potential performance degradation in function instances.

Data validation. For each function request to overcommitted instances, we record the context vector $\mathbf{x}$ at the moment it arrives and its execution latency after service. Define the P95 latency achieved in non-overcommitted instances as h. We label $\mathbf{x}$ positive if the corresponding request's latency $> h$; and negative otherwise. Therefore, a context vector $\mathbf{x}$ labeled as negative indicates no potential performance degradation.

In Fig. 3, we plot the CDFs of runtime metrics of $\mathbf{x}$ from the overcommitted detect-object (DO) and query-vacancy (QV) instances. We can see distinct patterns between negative and positive points. Overcommitted DO instances with lower CPU utilization, fewer inflight requests, and fewer NodeLLCMs are more likely to offer satisfactory performances, because DO is CPU-intensive to provide model serving service.

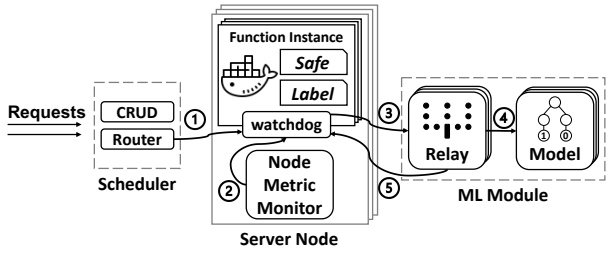


Figure 4: System architecture. The CRUD module creates, reads, updates, and deletes function instances.

As models are loaded when instances launch, little difference exists in the DO:Memory plot. Overcommitted QV instances with lower network traffic and LLCMs tend to offer satisfactory performance because QV, a network-intensive function, interacts with remote databases frequently. Similar results hold for other benchmark applications.

Model validation. We design a model validation test to verify that our metrics contain useful information that can be learned to predict function performance. We formulate a binary classification problem for fast and accurate modeling [15] as we care more about whether a request's latency exceeds specified requirements. We deploy the benchmark functions in Table 1 and make 1.5 million invocations. For each function request, we collect its **x** and label to build train and test datasets. Then, for each function in Table 1, we can train an off-the-shelf random forest classifier from scikit-learn [28], which has an ensemble of 100 decision trees. The F1 scores of the functions range from 0.71 to 0.84. Especially, the recalls range from 0.74 to 0.97, which means our metrics can be learned to avoid 74%-97% of the invocations that violate performance SLO. In §4.4, we describe how to use the metrics *online* to assist scheduling in Golgi.

4 GOLGI'S SCHEDULER DESIGN

With the metrics , we proceed to design Golgi, a performance-aware and resource efficient scheduler for FaaS computing.

4.1 Overview

Fig. 4 illustrates Golgi's architecture. Upon the arrival of requests, the router probes the status of function instances, such as their availability, to make routing decisions (①). Function instances run as containers to serve requests. Golgi hosts two kinds of instance for each function. The first is the non-overcommitted (non-OC) instance, which adheres to the user's resource configuration but may suffer resource underutilization. The second is the overcommitted (OC) instance, which down-sizes users' resource requests to pack more instances tightly on a server, using the usage-based overcommitment strategy [37] (§2.3) . To achieve the scheduling goal, Golgi dynamically makes online scheduling decisions between the two kinds of instances: it can exploit non-OC function instances that satisfy SLOs or explore the opportunities of using more resource-efficient OC instances for reduced provisioning cost, provided that the resulted performance still meets the SLO.

Golgi incorporates ML modules that leverage the runtime metrics discovered in §3 to identify overcommitted instances that may violate SLO. Each ML module corresponds to a function type and works for its instances. An ML module comprises a relay and a model. The relay collects runtime metrics from multiple overcommitted instances (③) at a time and sends a *batch* inference request to the model (④), which is a binary classifier(§4.4). The inference results are cached in each instance as the *Label* tag (⑤) for the router to query. Instances with positive *Label* tags, i.e., 1, are inferred to violate SLO. The *Label* tags are updated frequently, taking model inference off the scheduling critical path (§4.3).

Golgi trains ML models *online* with users' requests without collecting a large amount of training data via *offline* profiling. When a request is routed to a function instance, the instance records the runtime metrics and sends them, along with its execution latency, to the ML module asynchronously. This approach enables the model to collect online data for model updates. We adopt the Mondrian Forest (MF) [18] to develop our ML module, which excels in the *online* training setting and features short bootstrapping process. More specifically, we design an *online stratified sampling algorithm* to maintain balanced data samples in model updates (§4.4).

By collecting execution latencies, the ML module monitors its function's overall performance, i.e., P95 latency. When the function's SLO is violated, the ML module sets the *Safe* tags in its instances to 0, indicating that it is unsafe to *explore* OC instances and that the router should *exploit* non-OC instances. When the *exploration* will not lead to SLO violation, the *Safe* tag will be set to 1. The router is informed the overall function performance by *Safe* tags when it queries instances(①) to schedule requests.

Runtime metrics collection. A watchdog [2] runs lightweight daemon threads to collect intra-container metrics. Collocation interference metrics are reported to watchdogs by a node metric monitor, a daemon process in each server(②).

4.2 Conservative Routing

To save resource provisioning costs while meeting SLOs, we formulate the request routing as an online decision making problem. Basically, the router has two routing options, i.e., routing it to non-OC instances (*exploit default options*) or OC instances (*explore aggressive options*). Though the *Label* tags can differentiate OC instances to enable performance-aware *exploration*, occasional incorrect inferences pose a risk.

[2]Watchdog is a built-in module in OpenFaaS [11]. It launches and monitors functions. Any binary can become a function with a watchdog.

We control the *exploration* to achieve *conservative* routing. To route a request, the router probes instances' status, including their tags, to make a scheduling decision. It only *explores* the opportunities of resource overcommitment when the *Safe* tags are 1, meaning the exploration will not compromise the performance SLO. The *Safe* is set to 1 initially, meaning the first request of a function is routed to an OC-instance. For subsequent requests, the ML module monitors requests' latencies to set the *Safe* accordingly.

Performance-aware exploration. The ML module leverages runtime metrics from OC instances to identify whether they can meet the performance SLOs (§4.4). Instances with a *Label* tag of 0 (negative) are classified as eligible, while those with a *Label* tag of 1 (positive) are not. Our microbenchmark experiment in §8.4 show the effectiveness of the ML model in classifying OC instances to meet performance requirements. In Golgi, we first adopt the *power of two choices* approach [25] to pick up a handful of instances. Among these instances that are eligible for performance, we further select one that greedily reduces the provisioning costs, i.e., the MRU algorithm (§2.1). The greedy approach is reasonable because the resource provisioning cost is determined by several dynamic factors, including invocation patterns, function execution latency, functions' configurations. It is difficult to figure out the optimal routing decisions that minimize the cost in the long run. When none of the instances are available, we create new ones to scale out.

4.3 Enable Model Inference in Scheduling

Our router relies on model inference to make performance-aware decisions. However, a single model inference can take 100ms in our measurements, and the inference latency surges if we run inference for every incoming request. It is impractical to use model inference in the routing critical path.

To mitigate inference overhead, we decouple the model inference (③④⑤) from the routing critical path (①). We achieve this by using a relay that continuously collects runtime metrics from a *group* of function instances(③). Upon collection, the relay then sends a *batch* inference request(④) and moves on to the next collection round without blocking. The inference requests are cached in each function instance as the *Label* tag(⑤). Our measurements show that the *Label* tag is updated every 82.2ms, when the *group* size is 100, comparable to the average RPS of popular functions[3] in production, i.e., 11. Frequent updates prevent staleness.

By taking model inference off critical path, the router only queries cached tags (①) to make routing decisions. This design allows Golgi to scale across a wide range of workloads, while maintaining an acceptable routing latency. In §8.6, we show Golgi can handle the maximum RPS recorded in the Azure Function trace [34], i.e., 5139, and incur little overhead.

[3]Functions that are invoked at least once per second. Their invocations account for 93.3% of the total invocations in the Azure Function [34].

4.4 Leverage Runtime Metrics

Next, we dive into the model design of Golgi's ML module.

Stratified Sampling to Enforce Data Balance. As described in §4.1, the model in Golgi is a binary classifier, which updates online with users' invocations to meet the practical profiling constraints(§2.2). However, as metric vectors are labeled according to tail latency SLOs, the label distribution tends to be skewed. The negatives can be 10× more than the positives in our collected data. A classifier trained on imbalanced datasets commonly generalizes poorly on the less represented classes. To tackle the label imbalance, we design a memory efficient online stratified sampling mechanism based on the reservoir sampling algorithm [39]. Specifically, given a batch size N, we randomly sample $\frac{N}{2}$ negative data and $\frac{N}{2}$ positive data from the online data stream in a single pass. We concatenate them to form a batch of N data points and update the model. The online stratified sampling process is described in Algorithm 1. Fig.9(left) shows that a balanced label distribution helps the classifier trained online achieve an F1 score of 0.78, while the imbalanced one makes classifier underperform with an F1 score of 0.26 (§8.4).

Algorithm 1: Online Stratified Sampling Algorithm

Input: A batch size: N; Two placeholders and two counters for sampling positive and negative training data $pos = [\,]$, $neg = [\,]$, $posCntr = 0$, $negCntr = 0$;

```
while len(pos) + len(neg) ≠ N do
    A data point (x_i, y_i) arrives ;
    if y_i == 1 then
        pos, posCntr = Sample(pos, posCntr, x_i, y_i)
    else
        neg, negCntr = Sample(neg, negCntr, x_i, y_i)
    end
end
Function Sample(placeholder, cntr, x, y):
    cntr += 1
    if len(placeholder) < N/2 then
        placeholder.append((x, y))
    else
        Generate a random number j from [0, cntr]
        if j < len(placeholder) then
            placeholder[j] = (x, y)
        end
    end
    return placeholder, cntr
```

Model selection. We build the classifier based on the random forest model because the runtime metrics are tabular data. Based on the runtime metrics in function instances, the classifier helps the router track instances' changing runtime

status by setting their *Label* tags. We also try other models, e.g., neural network classifier. However, it results in F1 scores from 0.0 to 0.73, worse than those of their random forest counterparts, i.e., 0.71 to 0.84.

We require the classifier to update *online* with users' requests without explicit profiling to collect training data. This is because invoking users' functions without their permission may cause data inconsistencies in their external dependencies, as explained in §2.2. Inspired by [18], we adopt the Mondrian Forest (MF) to develop an online binary classifier. It can update with online data efficiently without complex calculations, e.g., multiple epochs of gradient descent. In contrast, classical random forest models adopt full batch training. Once new data points arrive, they require re-training against all training data, which is inefficient.

The MF is an ensemble of Mondrian Trees (MTs), which grow incrementally in an online setting and rival the performance of its batch training counterpart [18]. It also features a short bootstrapping process, requiring less than 50 model updates, as shown in Fig. 9(left). The prediction of MF is the average of MTs' predictions.

MT overview. MT training takes a batch of N training data $\{\mathbf{x}, y\}$, where $\mathbf{x}$ is a D-dimensional vector, and y is a binary scalar label. Training a decision tree requires searching the optimal feature dimension d_{opt} and a corresponding split location δ_{opt} to split data at each node in a tree. The optimal split (d_{opt}, δ_{opt}) creates child nodes to grow the tree. As the training proceeds, the tree recursively partitions the space into subspaces, denoted as leaf nodes. In model inference, MT predicts labels for a new batch of data $\{\mathbf{x}'\}$.

Build an MT. The MT model differs from the classical decision trees in that it randomly samples d and δ for a split, which is independent of the label or the decrease in impurity. A split at node j in the tree associates with a split time $\tau_j > 0$. Split times increase with depth, i.e., $\tau_j > \tau_{parent(j)}$, and the split time of the root node ϵ's parent is $\tau_{parent(\epsilon)} = 0$. We use a tuple $(T, \mathbf{d}, \delta, \tau)$ to represent a Mondrian tree T.

At a node j in the tree, we associate it with a block $B_j \subseteq \mathbb{R}^D$ as the input space. The smallest block that encloses all the training data at node j is denoted as $B_j^{\mathbf{x}}$. Let $N(j)$ denote the indices of training data points at node j, i.e., $N(j) = \left\{n \in \{1, \ldots, N\} : \mathbf{x}_n \in B_j\right\}$. Denote l_{jd}^x and u_{jd}^x the lower and upper bounds across the d^{th} dimension against training data at node j. Therefore, $B_j^{\mathbf{x}} = (l_{j1}^x, u_{j1}^x] \times \cdots \times (l_{jD}^x, u_{jD}^x] \in B_j$.

We start with a non-negative lifetime parameter λ and a handful of n training data $\mathbb{D}_{1:n} \subseteq \mathbb{R}^D$ to build an MT. The process starts from a root node ϵ and recurses down the tree. At a node j, we sample its split time E from an exponential distribution with the rate being the sum of ranges across all dimensions $\sum_{d=1}^{D}(u_{jd}^x - l_{jd}^x)$. If $E + \tau_{parent(j)} \geq \lambda$, the current split time exceeds the defined lifetime; hence, we assign node j as a leaf node and stop. Otherwise, node j is an internal node, and we sample a split (d_j, δ_j). To be more specific, we first sample a feature dimension d_j with a probability proportional to the range of d of the data at the node j, i.e., $(u_{jd}^x - l_{jd}^x)$. The intuition behind such a split is that a feature with a huge range is more likely to be important. Next, the split location δ_j is drawn uniformly from $[l_{jd_j}^x, u_{jd_j}^x]$. The procedure recurses along the left and right children. Algorithm 2 describes the whole process.

Algorithm 2: Build a Mondrian Tree

Initialize: a MT $T = \emptyset$; $leaves(T) = \emptyset$; training data at the root node $N(\epsilon) = \{1, 2, \ldots, N\}$; a non-negative λ ;
BuildMondrianBlock$(\epsilon, \mathbb{D}_{N(\epsilon)}, \lambda)$;
Function *BuildMondrianBlock*$(j, \mathbb{D}_{N(j)}, \lambda)$**:**
  Add node j to tree T;
  For all dimension $d \in D$, $l_{jd}^x = \min(\mathbf{x}_{\mathbf{N(j),d}})$, $u_{jd}^x = \max(\mathbf{x}_{\mathbf{N(j),d}})$; `// dimension-wise operations`
  Sample E from an exponential distribution with rate $\sum_d (u_{jd}^x - l_{jd}^x)$;
  if $\tau_{parent(j)} + E < \lambda$ **then**
    Set $\tau_j = \tau_{parent(j)} + E$;
    Sample a dimension d_j with probability proportional to $u_{jd}^x - l_{jd}^x$;
    Sample a split location δ_j uniformly from $[l_{jd_j}^x, u_{jd_j}^x]$;
    Set $N(left(j)) = \left\{n \in N(j) : \mathbf{x}_{n,d_j} \leq \delta_j\right\}$;
    Set $N(right(j)) = \left\{n \in N(j) : \mathbf{x}_{n,d_j} > \delta_j\right\}$;
    BuildMondrianBlock$(left(j), \mathbb{D}_{N(left(j))}, \lambda)$;
    BuildMondrianBlock$(right(j), \mathbb{D}_{N(right(j))}, \lambda)$;
  else
    Set $\tau_j = \lambda$ and add node j to $leaves(T)$;
  end

We next compute a label distribution to each node j in the tree. Denote G_j the distribution of labels at node j and G_{jk} the probability that a point is labeled k in the node j. In [18], they adopt a hierarchical Bayesian approach for smooth label distributions. A prior is chosen under which the label distribution of a node is similar to that of its parents. The prior mean of the root nodes is initialized as $[0.5, 0.5]$ in the binary classification problem. For each leaf node j, denote $c_{j,k}$ the number of data with label k at node j, and $t_{j,k} = \min(c_{j,k}, 1)$. For every non-leaf node j, we set its $c_{j,k} = t_{left(j),k} + t_{right(j),k}$ and $t_{j,k} = \min(c_{j,k}, 1)$. Denote $c_{j,\cdot} = \sum_k c_{j,k}$, $t_{j,\cdot} = \sum_k t_{j,k}$ and $d_j = \exp(-\gamma(\tau_j - \tau_{parent(j)}))$. The predictive probability of labels, i.e., posterior mean, at node j is computed recursively:

$$\bar{G}_{jk} = \begin{cases} \frac{c_{j,k} - d_j t_{j,k}}{c_{j,\cdot}} + \frac{d_j t_{j,\cdot}}{c_{j,\cdot}} \bar{G}_{parent(j),k} & c_{j,\cdot} > 0 \\ \bar{G}_{parent(j),k} & c_{j,\cdot} = 0 \end{cases} \tag{1}$$

The discount d_j interpolates between the counts c and the prior. If d_j is closer to 1, $\bar{G}_j$ is more alike with $\bar{G}_{parent(j)}$. If d_j

is closer to 0, $\bar{G}_j$ weights the count c more. All probabilities can be computed in a single pass from root to leaves.

Online Update. Given a new data point $(\tilde{\mathbf{x}}, \tilde{y})$, the procedure starts at the root node ϵ and recurses down the tree. At a node j, we first check if the new data point lies within $B_j^{\mathbf{x}}$ by computing the additional extents across feature dimensions incurred by the new data point, i.e., $\mathbf{e}^l = \max(\mathbf{l}_j^x - \tilde{\mathbf{x}}, 0)$ and $\mathbf{e}^u = \max(\tilde{\mathbf{x}} - \mathbf{u}_j^x, 0)$. When $\mathbf{e}^l = \mathbf{0}$ and $\mathbf{e}^u = \mathbf{0}$, the new data point lies in B_j^x and the procedure traverses its child nodes recursively. Otherwise, we should extend the current node to incorporate the new data point. To extend a node j, we first sample a split time E from an exponential distribution with the rate being the sum of the additional extent, i.e., $\sum_d^D (e_d^l + e_d^u)$ and set the split time to $E + \tau_{parent(j)}$. If $E + \tau_{parent(j)} > \tau_j$, there is no split, and we extend the existing block's bounds to include the new data $\tilde{\mathbf{x}}$. Otherwise, a new split dimension d_j is chosen with probability proportional to $e_d^l + e_d^u$ and a split location is drawn from a uniform distribution $[u_{j,d_j}^x, x_{d_j}]$ if $u_{j,d_j}^x < x_{d_j}$, otherwise $[x_{d_j}, l_{j,d_j}^x]$. The new split is consistent with the existing tree structure.

Online Inference. Consider a new data point $\mathbf{x}'$. If $\mathbf{x}' \in B_j^x$ for any leaf node j, the prediction is $\bar{G}_{jk}$. Otherwise, we extend the existing tree by including $\mathbf{x}'$ following the online update procedures and set the prediction probabilities to the label distribution of the leaf node that contains $\mathbf{x}'$. We average over all possible extensions as including $\mathbf{x}'$ may generate multiple tree extensions. The integration can be done analytically with linear complexity in the tree depth.

5 VERTICAL SCALING

Golgi incorporates a vertical scaling design that dynamically adjusts the configurations of OC instances to meet their performance SLO. This encourages the router to *explore* OC instances by reducing performance degradation risk (§4.2).

Motivations. Deciding on optimal resource configurations for serverless applications is hard [32]. In §2.3, we see the existing strategy, i.e., Orion [24], requires profiling to search the optimal configuration, causing a temporary unsatisfactory performance. Vertical scaling, however, provides flexibility in accommodating varying serverless workloads [7], enabling efficient resource utilization by allocating resources in real-time to match the performance SLOs. In Golgi, we dynamically scale up and down function instances based on their runtime performance, maximizing resource utilization while strictly meeting performance SLOs.

Our vertical scaling design also encourages Golgi's scheduler to explore the resource *overcommitment* options, saving further resource provisioning costs. Overcommitment by packing function instances too tightly leads to performance degradation due to intensive resource contention and severe collocation interferences (§2.3). In this setting, function instances are overloaded, and executions are throttled. Golgi's router leverages runtime metrics to avoid overloaded instances (§4). However, the classifier may occasionally make incorrect inferences. Our vertical scaling mechanism, however, can automatically adapt the overcommitted instances to their performance requirements, remedying the occasional incorrect inferences to improve system robustness.

Existing vertical scaling approaches are not well-suited for a serverless setting. They adjust containers' serving capabilities by changing their resource allocations [14, 32], e.g., adding CPU cores. Though these approaches work best with long-running jobs, they fall short in serverless settings as changing a container's resource allocation usually requires re-launching the container, leading to unavoidable service downtime [14]. Worse, function instances require frequent vertical scaling due to their highly dynamic resource utilization [7]. We require frequent scaling operations that can be performed seamlessly without any service disruption.

Monitor Performance. The objective of scaling is to maximize the number of requests that a function instance can serve without any degradation in performance. To achieve this, we create two counters within a function instance to efficiently monitor its runtime performance in terms of tail latency (such as P95 latency). These counters keep track of the number of requests that exceed the tail latency requirement and the total number of requests served, respectively, within a monitoring window of size W. The *ratio* between these two counters indicates whether the function instance is meeting the tail latency requirements. If the *ratio* exceeds 0.05, it means that the performance of the instance is not meeting the P95 latency requirement and scaling operations will be triggered. Additionally, speculative scaling will also be initiated once the first counter exceeds $0.05W$.

Continuous and In-place Scaling. We scale an instance's maximum request concurrency, which specifies the maximum number of requests that can be served concurrently in a single function instance, according to its runtime performance. As shown in Fig. 2b and discussed in §3.1, function performance is impacted by the request concurrency. A low concurrency can avoid excessive requests and mitigate resource contention. Suppose the performance requirement is defined by P95 latency. Then, if the violation *ratio* is greater than 0.05, we scale the concurrency down by one. Conversely, we also consider scaling up concurrency by one if runtime performance is too well; say, the *ratio* falls below 0.03. The buffering difference of 0.02 is left for stability. After each scaling operation, we reset the two counters. To be thread-safe, the scaling up/down operates atomically.

Effectiveness. Fig.8 presents scaling operations in a classify-image instance in evaluation (§8.2). Its maximum request concurrency (green solid line) adapts frequently to latency requirement (red dotted line) without service downtime. Fig.7 (right) highlights that vertical scaling encourages exploration by reducing 21% resource usage of non-OC instances.

6 FURTHER DESIGN CONSIDERATIONS

Scalability. Golgi is scalable to host thousands of functions [34]. For each type of user function, we deploy an ML module to help the scheduler make routing decisions. In case a user function spawns thousands of instances, we adopt a *shared-nothing* strategy that divides the instances into a set of disjoint shards, each of size S. Then, for each shard, we use an ML module to manage. Our measurements show that the tag update period is around $82.2ms$ when $S = 100$, comparable to the median function execution latency of 152ms and the average RPS in production, i.e., 11 [34]. In §8.6, we present more evaluation details of scalability.

Fault Tolerance. Modules in Golgi, including routing, function instances, and ML modules, run in stateless Kubernetes pods. We can deploy multiple replicas to support fault tolerance. Explicit program errors in function instances are returned with status codes, and developers are responsible for handling side-effects if their functions are not idempotent, similar to AWS Lambda and other FaaS platforms [36].

7 IMPLEMENTATION

We implement Golgi based on OpenFaaS [11] with Kubernetes. Fig.4 shows its architecture with three main parts.

1) ***Scheduler.*** We plug our routing policy in OpenFaaS's faas-netes module. Following production practices, we implement a first-fit container placement strategy [37] by adding a customized plugin in the Kubernetes scheduler [3].

2) ***ML Module.*** The MF model is implemented in Python, and the ML relays are implemented in Go. They interact with other modules via the high-performance gRPC framework.

3) ***Function Instances.*** Functions are implemented based on OpenFaaS's templates. We implement the watchdog by extending OpenFaaS's of-watchdog module. The vertical scaling design (§5) is implemented in the watchdog module.

8 EVALUATION

We prototype Golgi and evaluate its effectiveness using scaled production traces [34]. We conducted extensive experiments. Evaluation highlights include:

(1) Golgi reduces 42% memory usage and 35% server cost while meeting performance SLOs (§8.2), outperforming state-of-the-art resource efficient mechanisms.
(2) Golgi's router design and vertical scaling mechanism are effective in ablation studies. (§8.3 and §8.4)
(3) Golgi can explore the trade-off between function performance and resource provisioning cost. (§8.5)
(4) Golgi has acceptable overhead and scalability (§8.6).
(5) Golgi is effective in a production deployment, reducing 30% memory usage under performance SLOs (§8.7).

8.1 Methodology

Cluster Setup. We set up ten AWS EC2 instances to prototype and evaluate Golgi. One c5.9xlarge instance serves as the master node and hosts the routing modules. Seven c5.9xlarge instances serve as worker nodes for hosting function instances. Each has 36vCPUs and 72GB memory. One c5.2xlarge instance hosts function dependencies, e.g., database. Another c5.2xlarge instance sends function requests. Each c5.2xlarge instance has 8vCPUs and 16GB memory.

Trace. We use the publicly released Azure Function Trace [34]. We randomly select traces of day 10 (weekday) and day 13 (weekend) because the production traces show diurnal and weekly patterns [34]. We scale the original trace down, while keeping its fluctuations, illustrated in Fig. 6. For a day-long trace, we further scale it to an hour long proportionally, which has about 600, 000 invocations.

Benchmark Applications. We use the benchmark applications [37] in Table 1 (§3), which reflect real-world business scenarios. Functions' resource configurations are set according to [37], with an initial request concurrency limit of 4.

Baseline Methods. We compare Golgi with four baselines.

1) ***Non-Overcommitment (BASE).*** The common practices in existing serverless platforms are non-overcommitted configurations with the MRU routing strategy (§2.1).

2) ***Naive Overcommitment (OC).*** Naive overcommitment strategy [37] down-sizes functions' memory requests to $0.3\times c + 0.7 \times a$ (§2.3). The routing strategy is MRU.

3) ***Profiling and Performance Modelling (Orion).*** Orion uses profiling and performance modeling [24] to right size function instances so that they can meet their SLOs at minimum cost (§2.3). The routing strategy, in this case, is MRU.

4) ***E&E Router (E&E).*** We disable the vertical scaling mechanism to show the effectiveness of router design (§4). Memory configurations of the overcommitted instances are set as $0.3 \times c + 0.7 \times a$.

5) ***Golgi.*** E&E router with vertical scaling enabled (§5).

8.2 End-to-End Comparison

Procedures. We start with the non-overcommitment baseline (BASE). We send requests by the scaled traces and record the P95 latency of each application as the performance SLO.

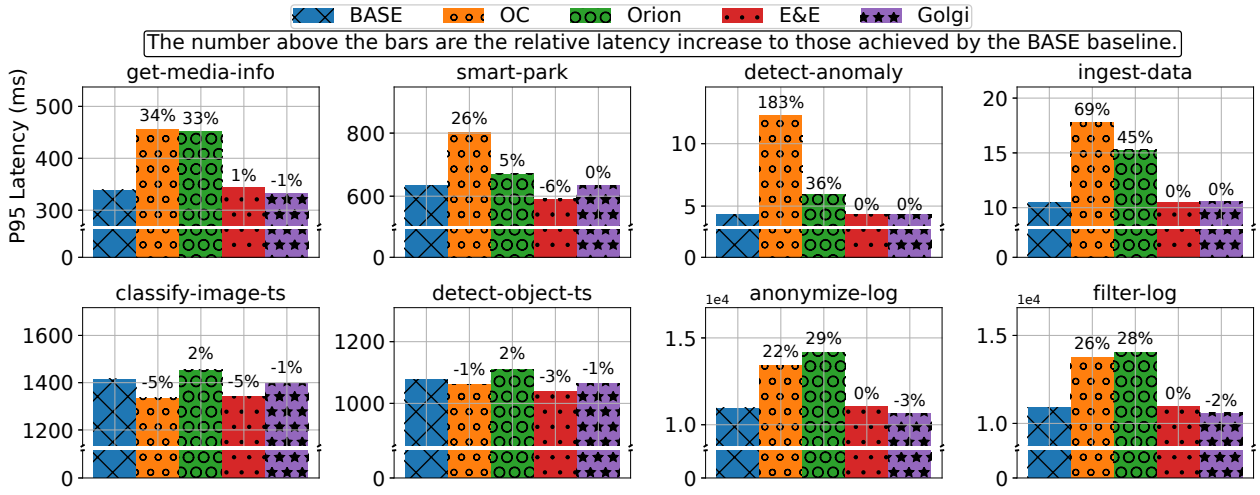


Figure 5: Application performance in end-to-end experiments.

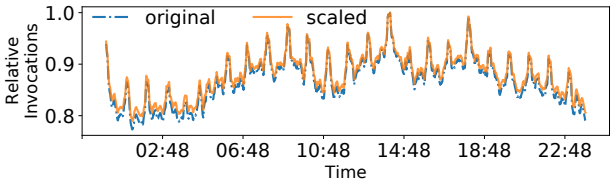


Figure 6: Comparison between the original [34] and the scaled production traces, normalized to the peak.

Next, we run other baselines with the same traces by setting their performance objectives as the performance SLO derived from the BASE. We repeat experiments for five times.

Metrics. We focus on the benchmark applications' performance and the resource provisioning costs (§2.1). The application performance is measured by P95 latency, as suggested by recent research and production practices [24, 26, 35, 37, 46].

We measure the resource provisioning cost by two metrics. The first one is memory footprint, defined as MB×Sec. For instance, the memory footprint of a 128MB function running for 0.1 seconds is 12.8MB × Sec. As described in §2.1, most serverless platforms charge their users according to the memory footprint [4, 9, 19]. It reflects precise resource utilization. The second one is VM time, which is the total running time of VMs occupied by function instances. It reflects the serving cost borne by platforms [37, 47]. Ideally, the two metrics are asymptotically equal in the large-scale production system, if all servers run with full workloads. The difference between them is due to resource fragmentation, which is not rare.

Results. Fig. 5 compares applications' P95 latency achieved by the baseline methods. Golgi and the E&E router baseline substantially outperform the others with their negligible performance loss. E&E router causes an increase in GMI's P95 latency by 1%. Other applications' SLOs are well satisfied by Golgi and the E&E router. In contrast, the OC and Orion baselines cause significant performance degradation. Applications' P95 latency from the naive OC surges up to 183% while that from the Orion increases up to 45%. Orion's original paper [24] also provides similar results, where its right-sizing strategy increases the P95 latency by 35% ∼ 132% compared to the defined performance objectives.

Fig.7 (left) compares the resource provision costs incurred by each baseline method. The values are normalized by those from the BASE, and we report the relative costs. Note that the BASE takes about 742 TB×Sec memory footprint and 18,000 second VM time on average. Compared to BASE, Golgi achieves a 42% reduction in memory footprint and a 35% reduction in VM time. The naive OC baseline achieves a 57% reduction in memory footprint and a 56% reduction in VM time, but it fails to meet the performance requirements shown in Fig. 5. Our E&E router achieves a 34% reduction in memory footprint and a 20% reduction in VM time.

Fig.7 (right) presents a memory footprint breakdown and highlights the effectiveness of the vertical scaling design. We measure the resource provisioning costs in terms of memory footprint to reflect precise resource usage. With the vertical scaling enabled, Golgi saves an addition of 8% memory footprint. The additional savings is due to the reduced resource utilization of non-OC instances, indicating that the vertical scaling mechanism encourages *exploration*.

8.3 Vertical Scaling

The vertical scaling design adapts overcommitted function instances to their performance requirements. We illustrate the scaling operations in a classify-image instance in Fig. 8. We sample 300 seconds out of its 1800 second lifetime. We can see that the instance's maximum request concurrency (green

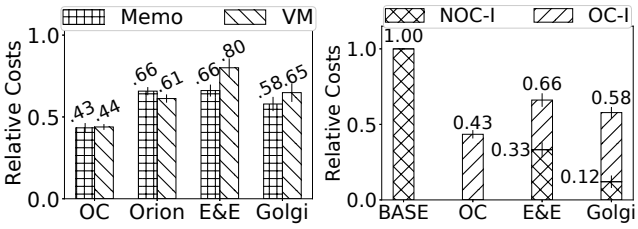


Figure 7: Left: Cost Comparison (Memo: memory footprint; VM: VM time). Right: Memory Footprint Breakdown (NOC-I/OC-I: resource usage of non-OC/OC instances). Numbers indicate bar heights.

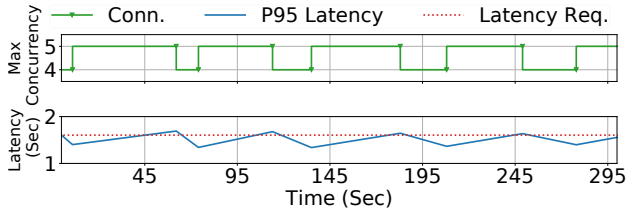


Figure 8: Vertical scaling illustration. Blue solid line represents the monitored P95 latency of requests in a CI instance. The red dotted line is the P95 latency requirement. The max concurrency (green solid line) automatically scales according to runtime performance.

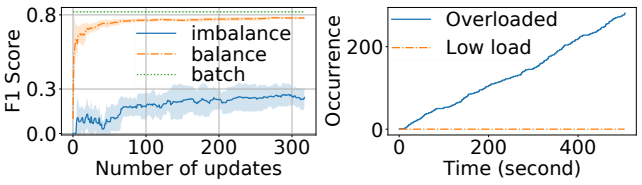


Figure 9: Left: F1 score of the MF classifier under different settings; the *balance* line also illustrates model bootstrapping process in Golgi; Right: Occurrence times of positive instances.

solid line) is adjusted frequently according to its runtime P95 latency (blue solid line). Every time the runtime P95 latency is higher than the requirements (red dotted line), the maximum concurrency decreases by 1. When the requirement is well satisfied, the maximum concurrency increases by 1. Fig.7 (right) highlights that vertical scaling encourages the scheduler to explore overcommitted instances by reducing 21% resource usage of non-overcommitted instances.

8.4 Router Evaluation

Ablation Study. To highlight the effectiveness of our router, we run Golgi by replacing its router with the MRU one and enabling the vertical scaling mechanism. Experimental results show that MRU and vertical scaling can help save 60% of resource provision costs. However, applications' performances degrade, with P95 latency increasing from 3% to 71.5% because MRU is not performance-aware.

Classifier Performance. We evaluate the MF performances for each function in Table 1 and report F1 scores, the harmonic average of precision and recall. Across all functions, the F1 scores range from 0.70 to 0.84, rivaling the performance of its batch counterpart from sklearn[28]: 0.71 to 0.84.

Label Imbalance. We evaluate the performance of the MT classifier w/ and w/o label balance enforced to highlight the importance of our *stratified sampling mechanism*(§4.3). We simulate the online learning setting and set the batch size to 16. The function considered in this case is get-media-meta.

In Fig.9 (left), we can see that a balanced label distribution improves the classifier's performance significantly, achieving an F1 score of 0.78. Label imbalance, however, causes the classifier to underperform with an F1 score of 0.26.

Microbenchmark. We further dive into the routing process and show the effectiveness of a classifier in a microbenchmark experiment. We set a tiny testbed with two worker nodes and consider three functions: get-media-meta (GMM), classify-image (CI), and detect-object (DO). The first node is set as an overloaded environment and instances of the three functions collocate within it. The second node is set to be a contention free environment. For each function, we send requests at a constant speed of 30 RPS for 500 seconds.

We focus on the GMM and place 40 GMM instances with half in the overloaded environment and half in the contention free environment. We train a classifier from scratch and record the occurrence times of positive GMM instances in each node every second, see Fig.9(right). We see that no positive instances occur in the contention free environment (orange dash-dot line), while GMM instances in the overloaded environment are classified as positive from time to time (blue solid line), indicating that the classifier can help our router avoid overloaded server nodes. GMM's P95 latency in the overloaded node is 1.5× longer than that in the contention free node. Throughout the experiment, the overloaded node has an average LLCMs of 192,369,792, while the average LLCMs in the contention free node is 2,332,726.

8.5 Explore performance-cost trade-off

A slack in performance SLO can help Golgi save more resource provisioning costs. Denote a function's P95 latency from the BASE as h; we set two customized objectives as "P95 latency $\leq 1.2h$" and "P95 latency $\leq 1.4h$" [37]. We run Golgi with the customized SLOs and find that the cost savings increase with more slacks in performance requirements. Golgi meets applications' SLOs in all settings. With 1.2h and 1.4h, Golgi saves 50% and 55% memory footprint.

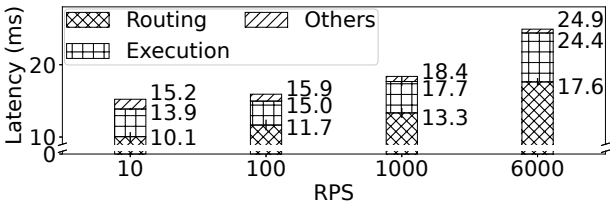


Figure 10: Scalability evaluation with various RPSs.

8.6 Scalability and Overhead

We evaluate Golgi's scalability under a set of RPSs according to the Azure trace [34], which has a maximum RPS of 5139. We consider the get-duration function. We deploy enough function instances to serve requests. Define routing latency as time elapsed to make a routing decision. In Fig. 10, we show the end-to-end latency and the breakdown results. Golgi can scale with different workloads. During a request spike of 6000 RPS, Golgi can make a routing decision within 20ms, which is considered acceptable by AWS Lambda [7].

We next evaluate the tag update latency with a large size of 1,200 function instances in our system. Note that AWS lambda has a default regional quota of 1,000 instances [21]. We find the update latency is 82.2ms when the group size is 100, comparable to the average RPS of popular functions in production, i.e., 11[34], and median execution latency 152ms. Besides, frequent tag updates incur negligible overheads on network bandwidth, transmitting 80K bytes per second.

We next consider the resource overhead of hosting ML modules. Take Azure Function as an example [34]. The most popular functions that are invoked at least once per second account for 1.2% of total functions but their invocations account for 93.3% of the total invocations in the platform. The total number of their instances is 12,912 [20], and the average runtime memory utilization is 136 MB. According to the recent analysis [26, 30, 35, 37], each function instance should be configured with 512MB on average. As each ML module requires 200MB and one module manages 100 instances, the overall overhead is 3.2% in terms of memory footprint. This is acceptable compared to the 42% saving by Golgi.

8.7 Golgi in Action

We finally evaluated Golgi in an internal production cluster, where function instances coexist with other data analytics tasks, e.g. keyword filter and log statistic calculation. Nodes in the cluster have heterogeneous configurations, with the number of available CPU cores ranging from 4 to 16. We deployed Golgi and BASE separately. For a fair comparison, we replayed the production trace to invoke two serverless applications. The first is an executor monitor and the second is for log processing. Fig. 11 depicts the experimental results. The latency distribution (Fig. 11 left and center) shows that Golgi protects applications' performance while saving the cost by 30% (Fig. 11 right). Besides, no performance degradation is reported from the coexisting analytic tasks.

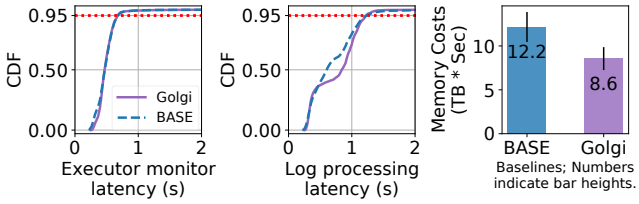


Figure 11: Evaluation in production cluster. Left and center: CDFs of latencies. Right: memory costs.

9 RELATED WORK

Optimizing scheduling in serverless computing is the target of recent studies. In §2.3, we present Owl [37] and Orion [24] that solve the similar problem as Golgi. Besides, OFC [26] collects under-utilized memory to build caches that optimize inter-function communication. Atoll [35] targets meeting the latency requirements for serverless applications. It uses a hierarchical design to reduce scheduling overheads and proactively launch instances to mitigate the cold start. However, they do not consider reducing the serverless platforms' provisioning cost. Fireplace [7] targets packing function instances tightly to reduce provisioning costs. Their scheduling goal is to place function instances tightly so that the resource utilization of each server does not exceed its capacity. They do not strive to meet function SLOs in scheduling.

A series of works optimize the intermediate data transfer between functions to improve serverless application performance [17, 23, 29, 31, 44]. Their works are orthogonal to ours. Golgi targets cost-efficient and performance-aware scheduling and can integrate their designs seamlessly.

10 CONCLUSION

We present Golgi, a performance-aware and resource-efficient FaaS scheduling system. Driven by the characteristics of functions, Golgi achieves performance-aware resource overcommitment and scheduling, reducing operational costs while meeting function SLOs. It also supports an automatic vertical scaling mechanism that adapts overcommitted instances to their performance requirements. Compared with existing production scheduling methods, Golgi yields significant cost reduction by 42% with negligible performance loss.

ACKNOWLEDGEMENT

We would like to thank our shepherd, Sadjad Fouladi, and the anonymous reviewers for their valuable feedback that helps improve the quality of this work. This research was supported in part by the WeBank Faculty Fellowship and RGC GRF grants under the contracts 16202121 and 16210822.

REFERENCES

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*.

[2] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. 2018. On the diversity of cluster workloads and its impact on research results. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[3] The Kubernetes Authors. 2023. Kubernetes Scheduling Framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[4] Microsoft Azure. 2022. Azure Functions Pricing. https://azure.microsoft.com/en-us/pricing/details/functions/.

[5] Microsoft Azure. 2022. Concurrency in Azure Functions. https://docs.microsoft.com/en-us/azure/azure-functions/functions-concurrency.

[6] Microsoft Azure. 2022. What are Durable Functions? https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp.

[7] Bharathan Balaji, Christopher Kakovitch, and Balakrishnan (Murali) Narayanaswamy. 2020. FirePlace: Placing FireCracker virtual machines with hindsight imitation. In *Proceedings of the MLSys 2021, NeurIPS 2020 Workshop on Machine Learning for Systems.*

[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 *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys 21)*.

[9] Alibaba Cloud. 2022. Aliyun Function Compute Pricing. https://www.alibabacloud.com/help/en/doc-detail/54301.html.

[10] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 17)*.

[11] Alex Ellis. 2022. OpenFaaS : Server Functions, Made Simple. https://www.openfaas.com/.

[12] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys 18)*.

[13] Google. 2022. Overcommitting CPUs on sole-tenant VMs. https://cloud.google.com/compute/docs/nodes/overcommitting-cpus-sole-tenant-vms.

[14] Google. 2022. Vertical Pod autoscaling. https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler.

[15] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation.*

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

[17] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[18] Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. 2014. Mondrian Forests: Efficient Online Random Forests. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS 14)*.

[19] AWS Lambda. 2022. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[20] AWS Lambda. 2022. How do I request a concurrency limit increase for my Lambda function? https://aws.amazon.com/premiumsupport/knowledge-center/lambda-concurrency-limit-increase/.

[21] AWS Lambda. 2022. Lambda function scaling. https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html.

[22] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. 2021. George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 21)*.

[23] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC 21)*.

[24] 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 *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*.

[25] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* (2001).

[26] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: An Opportunistic Caching System for FaaS Platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys 21)*.

[27] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. 2017. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139* (2017).

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* (2011).

[29] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[30] Ran Ribenzaft. 2019. What AWS Lambda's Performance Stats Reveal. https://epsagon.com/observability/what-aws-lambda-performance-stats-reveal-key-metrics/.

[31] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 21)*.

[32] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys 20)*.

[33] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J Yadwadkar, Raluca Ada Popa, Joseph E Gonzalez, Ion Stoica, and David A Patterson. 2021. What serverless computing

is and should become: The next phase of cloud computing. *Commun. ACM* (2021).

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

[35] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 21)*.

[36] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *In Proc. VLDB Endow.* (2020).

[37] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. 2022. Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 22)*.

[38] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. 2022. Owl: Performance-Aware Scheduling for Resource-Efficient Function-as-a-Service Cloud. https://www.cse.ust.hk/~weiwa/papers/owl-techreport.pdf.

[39] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* (1985).

[40] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC 18)*.

[41] 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 *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*.

[42] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-Aware Dynamic Resource Configuration for Serverless Function Workflows. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*.

[43] Renyu Yang, Chunming Hu, Xiaoyang Sun, Peter Garraghan, Tianyu Wo, Zhenyu Wen, Hao Peng, Jie Xu, and Chao Li. 2020. Performance-Aware Speculative Resource Oversubscription for Large-Scale Clusters. *IEEE Transactions on Parallel and Distributed Systems* (2020).

[44] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.

[45] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with Serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 20)*.

[46] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC 19)*.

[47] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.

[48] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI^2: CPU performance isolation for shared compute clusters. In *Proceedings of the SIGOPS European Conference on Computer Systems (EuroSys 13)*.