FAASNET: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute

Ao Wang[1], Shuai Chang[2], Huangshi Tian[3], Hongqi Wang[2], Haoran Yang[2], Huiba Li[2],
Rui Du[2], Yue Cheng[1]
[1]*George Mason University* [2]*Alibaba Group* [3]*Hong Kong University of Science and Technology*

Abstract

Serverless computing, or Function-as-a-Service (FaaS), enables a new way of building and scaling applications by allowing users to deploy fine-grained functions while providing fully-managed resource provisioning and auto-scaling. Custom FaaS container support is gaining traction as it enables better control over OSes, versioning, and tooling for modernizing FaaS applications. However, providing rapid container provisioning introduces non-trivial challenges for FaaS providers, since container provisioning is costly, and real-world FaaS workloads exhibit highly dynamic patterns.

In this paper, we design FAASNET, a highly-scalable middleware system for accelerating FaaS container provisioning. FAASNET is driven by the workload and infrastructure requirements of the FaaS platform at one of the world's largest cloud providers, Alibaba Cloud Function Compute. FAASNET enables scalable container provisioning via a lightweight, adaptive function tree (FT) structure. FAASNET uses an I/O efficient, on-demand fetching mechanism to further reduce provisioning costs at scale. We implement and integrate FAASNET in Alibaba Cloud Function Compute. Evaluation results show that FAASNET: (1) finishes provisioning $2,500$ function containers on $1,000$ virtual machines in 8.3 seconds, (2) scales 13.4× and 16.3× faster than Alibaba Cloud's current FaaS platform and a state-of-the-art P2P container registry (Kraken), respectively, and (3) sustains a bursty workload using 75.2% less time than an optimized baseline.

1 Introduction

In recent years, a new cloud computing model called serverless computing or Function-as-a-Service (FaaS) [40] has emerged. Serverless computing enables a new way of building and scaling applications and services by allowing developers to break traditionally monolithic server-based applications into finer-grained cloud functions. Developers write function logic while the service provider performs the notoriously tedious tasks of provisioning, scaling, and managing the backend servers [36] that the functions run on.

Serverless computing solutions are growing in popularity and finding their way into both commercial clouds (e.g., AWS Lambda [5], Azure Functions [7], Google Cloud Functions [12] and Alibaba Cloud Function Compute[1] [2], etc.) and open source projects (e.g., OpenWhisk [54], Knative [15]). While serverless platforms such as AWS Lambda and Google Cloud Functions support functions packaged as `.zip` archives [8], this deployment method poses constraints for FaaS applications with a lack of flexibility. One constraint is a maximum package size limit (of up to 250 MB uncompressed for AWS Lambda functions).

A recent trend is the support of packaging and deploying cloud functions using custom container images [3, 13, 19, 20]. This approach is desirable as it greatly enhances usability, portability, and tooling support: (1) Allowing cloud functions to be deployed as custom container runtimes enables many interesting application scenarios [4], which heavily rely on large dependencies such as machine learning [23, 47], data analytics [31, 32, 58], and video processing [26, 35]; this would not have been possible with limited function package sizes. (2) Container tooling (e.g., Docker [9]) simplifies the software development and testing procedures; therefore, developers who are familiar with container tools can easily build and deploy FaaS applications using the same approach. (3) This approach will enable new DevOps features such as incremental update (similar to rolling update in Kubernetes [18]) for FaaS application development.

A potential benefit that makes the FaaS model appealing is the fundamental resource elasticity—ideally, a FaaS platform must allow a user application to scale up to tens of thousands of cloud functions on demand, in seconds, with no advance notice. However, providing rapid container provisioning for custom-container-based FaaS infrastructure introduces non-trivial challenges.

First, FaaS workloads exhibit highly dynamic, bursty patterns [50]. To verify this, we analyzed a FaaS workload from a production serverless computing platform managed by one of the world's largest cloud providers, *Alibaba Cloud*. We observe that a single application's function request throughput (in terms of concurrent invocation requests per second or RPS) can spike up to more than a thousand RPS with a peak-to-trough ratio of more than 500× (§2.2.1). A FaaS platform typically launches many virtualized environments—in our case at Function Compute, virtual machines (VMs) that host and isolate containerized functions—on demand to serve request surges [5, 24, 56]. The bursty workload will create a network bandwidth bottleneck when hundreds of VMs that host the cloud functions are pulling the same container images from the backing store (a container registry or an object store). As a result, the high cost of the container startup process[2]

[1] We call Function Compute throughout the paper.

[2] A container startup process typically includes downloading the container

makes it extremely difficult for FaaS providers to deliver the promise of high elasticity.

Second, custom container images are large in sizes. For example, more than 10% of the containers in Docker Hub are larger than 1.3 GB [60]. Pulling large container images from the backing store would incur significant cold startup latency, which can be up to several minutes (§2.2.2) if the backing store is under high contention.

Existing solutions cannot be directly applied to our FaaS platform. Solutions such as Kraken [16], DADI [42], and Dragonfly [10] use peer-to-peer (P2P) approaches to accelerate container provisioning at scale; however, they require one or multiple dedicated, powerful servers serving as root nodes for data seeding, metadata management, and coordination. Directly applying these P2P-based approaches to our existing FaaS infrastructure is not an ideal solution due to the following reasons. (1) It would require extra, dedicated, centralized components, thus increasing the total cost of ownership (TCO) for the provider while introducing a performance bottleneck. (2) Our FaaS infrastructure uses a dynamic pool of resource-constrained VMs to host containerized cloud functions for strong isolation; a host VM may join and leave the pool at any time. This dynamicity requires a highly adaptive solution, which existing solutions fail to support.

To address these challenges, we present FaaSNet, a lightweight and adaptive middleware system for accelerating serverless container provisioning. FaaSNet enables high scalability by decentralizing the container provisioning process across host VMs that are organized in function-based tree structures. A *function tree* (*FT*) is a logical, tree-based network overlay. A FT consists of multiple host VMs and allows provisioning of container runtimes or code packages to be decentralized across all VMs in a scalable manner. FaaSNet enables high adaptivity via a tree balancing algorithm that dynamically adapts the FT topology in order to accommodate VM joining and leaving.

Note that the design of FaaSNet is driven by the specific workload and infrastructure requirements of Alibaba Cloud Function Compute. For example, Function Compute uses containers inside VMs to provide strong tenant-level isolation. A typical FaaS VM pool has thousands of small VM instances. The scale of the FaaS VM pool and the unique characteristics of FaaS workloads determine that: (1) a centralized container storage would not scale; and (2) existing container distribution techniques may not work well in our environment as they have different assumptions on both workload types and underlying cluster resource configurations.

We make the following contributions in this paper.

- We present the design of a FaaS-optimized, custom container provisioning middleware system called FaaSNet. At FaaSNet's core is an adaptive function tree abstraction that avoids central bottlenecks.

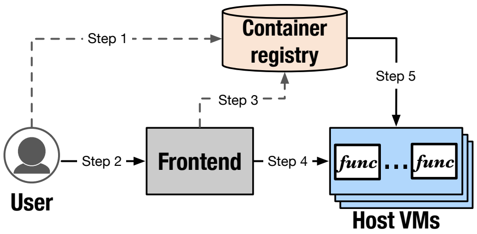


Figure 1: Overview of Alibaba Cloud's FaaS container workflows.

- We implement and integrate FaaSNet in Alibaba Cloud Function Compute. FaaSNet is, to the best of our knowledge, the first FaaS container provisioning system from a cloud provider with published technical details.
- We deploy FaaSNet in Alibaba Cloud Function Compute and evaluate FaaSNet extensively using both production workloads and microbenchmarks. Experimental results show that FaaSNet: finishes provisioning 2,500 function containers within 8.3 seconds (only 1.6× longer than that of provisioning a single container), scales 13.4× and 16.3× faster than Alibaba Cloud's current FaaS platform and a state-of-the-art P2P registry (Kraken), respectively, and sustains a bursty workload using 75.2% less time than an optimized baseline.
- We release FaaSNet's FT and an anonymized dataset containing production FaaS cold start traces at `https://github.com/mason-leap-lab/FaaSNet`.

2 Background and Motivation

In this section, we first provide an overview of the FaaS container workflows in Alibaba Cloud Function Compute. We then present a motivational study on the FaaS workloads to highlight the bursty patterns and their demands of a scalable and elastic FaaS container runtime provisioning system.

2.1 FaaS Container Workflows in Alibaba Cloud Function Compute

Function Compute allows users to build and deploy FaaS applications using custom container images and container tools. Figure 1 shows a typical workflow of function deployment and invocation. To deploy (or update) a containerized function, a user sends a `create`/`update` request in order to push the container image to a centralized container registry (Step 1 in Figure 1). To invoke a deployed function, the user sends `invoke` requests to the frontend gateway (Step 2), which checks the user's request and the status of the container image in the registry (Step 3). The frontend then forwards the requests to the backend FaaS VM cluster for servicing the requests (Step 4). Finally, host VMs create function containers and pull their container data from the registry (Step 5). After all the previous steps are successfully completed, host VMs become ready and start serving the invocation requests.

2.2 Workload Analysis

Step 5 in Figure 1, container runtime provisioning, must be fast and scalable in order to enable high elasticity for

image manifest and layer data, extracting layers, and starting the container runtime; in our paper we call the startup process *container provisioning*.

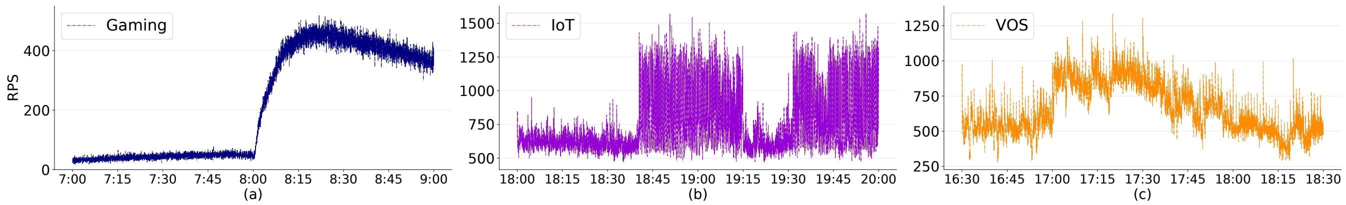
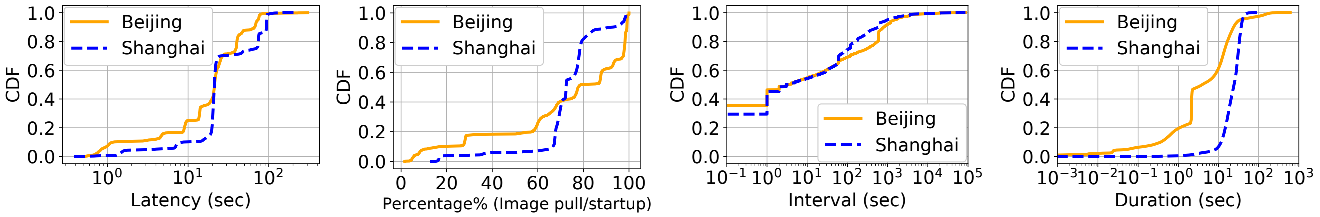


Figure 2: 2-hour throughput timelines of example FaaS applications.



(a) Image pull latency. (b) Proportion of image pull in cold start. (c) Inter-arrival time of cold starts. (d) Function duration.

Figure 3: Performance characteristics of container image pulls (a, b) and function invocations (c, d) in CDF.

FaaS workloads. To obtain a better understanding of the workload requirements, we analyze the workload traces that were collected from Function Compute.

2.2.1 Workload Burstiness

FaaS providers charge users using a fine-grained, pay-per-use pricing model—they bill on a per invocation basis (e.g., $0.02 per 1 million invocations for AWS Lambda) and charge the CPU and memory bundle resource usage at the millisecond level. This property is attractive to a broad class of applications that exhibit highly fluctuating and sometimes unpredictable loads; compared to traditional VM-based deployment approach that charges even when the VM resources are idle, FaaS is more cost-effective as tenants do not pay when the load is zero. Therefore, we analyzed the workload traces and verified that bursty behaviors are indeed common. Figure 2 reports the behaviors of three representative FaaS applications: gaming, IoT, and VOS (video processing).

Figure 2(a) shows that a request spike shoots from 22 to 485 RPS with a peak-to-trough ratio of 22×. As well as being bursty, IoT and VOS show different patterns. As shown in Figure 2(b), IoT exhibit a sustained throughput of around 682 RPS, but the throughput suddenly increased to more than 1460 RPS; the peak throughput lasts for about 40 minutes and the second peak starts 15 minutes after the first peak ends. Whereas for VOS (Figure 2(c)), for the first 30 minutes, it observes an average throughput of 580 RPS with a maximum (minimum) throughput of 982 (380) RPS; the average throughput increases to 920 RPS at 30 minutes, and gradually reduces back to an average of 560 RPS.

Implication 1: *Such dynamic behaviors require scalable and resilient provisioning of large numbers of function containers to rapidly smooth out the latency spikes that a FaaS application may experience during a request burst.*

2.2.2 Cold Start Costs of Containerized Functions

Next, we focus on cold start costs of containerized functions. A cold start, in our context, refers to the first-ever invocation of a custom-container-based function; a cold start latency is typically long, ranging from a few seconds to a few minutes as it requires the FaaS provider to fetch the image data and start the container runtime before executing the function. As noted in prior work [25, 48, 50, 56], the high cold start penalty is a notorious roadblock to FaaS providers as it hurts elasticity. The cold start issue is exacerbated when custom container feature with sizeable dependencies is supported.

We analyzed the container downloading costs in two FaaS regions, *Beijing* and *Shanghai*, managed by Function Compute. We retrieved a 15-day log, which recorded the statistics of the function container registry and reported the performance characteristics of 712,295 cold start operations for containerized functions. As shown in Figure 3(a), for *Beijing*, about 57% of the image pulls see a latency longer than 45 seconds, while for *Shanghai* more than 86% of the image pulls take at least 80 seconds.

We next examined the proportion of time spent on image pull with respect to the total function cold start latency. Figure 3(b) shows that more than 50% and 60% of function invocation requests spend at least 80% and 72% of the overall function startup time on pulling container images, for *Beijing* and *Shanghai* respectively. This indicates that the cost of image pull dominates most functions' cold start costs.

To put the cold start costs into perspective, we further inspected cold starts' inter-arrival time and function duration. Figure 3(c) plots the interval distribution of consecutive cold start requests. In both of the two regions, about 49% of function cold starts have an inter-arrival time less than 1 second, implying a high frequency of cold start requests. As shown in Figure 3(d), about 80% of the function executions in *Beijing* region are longer than 1 second; in *Shanghai* region, about

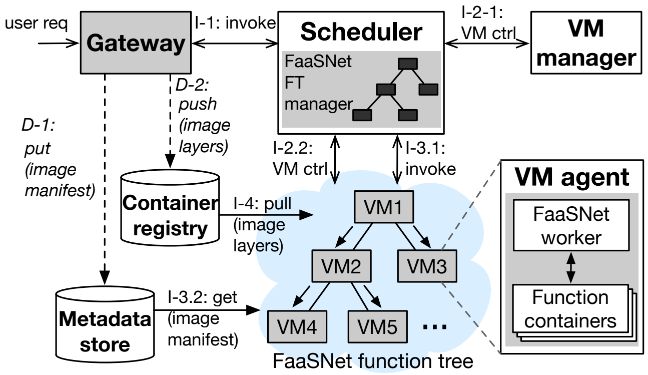


Figure 4: FaaSNet architecture. Our work is in the gray boxes. Function invocation requests (solid arrow: invoke, VM ctrl, get(image manifest), and pull(image layers)) are online operations. FaaSNet minimizes the operation of I-4: pull(image layers) and efficiently decentralizes container provisioning (i.e., image load and container start) across VMs. Function deployment requests (dashed arrow, italic font: *D-1: put(image manifest)* and *D-2: push(image layers)*) are offline operations.

80% of the function duration is less than 32.5 seconds, with a 90^{th} percentile of 36.6 seconds and a 99^{th} percentile of 45.6 seconds. This distribution indicates that cold start costs are of the same magnitude as the function duration, stressing a need for optimizing container startups.

Implication 2: *Optimizing the performance of container provisioning will provide a huge benefit on reducing the cold start costs of container-based cloud functions.*

3 FaaSNet Design

3.1 Design Overview

This section provides a high-level overview of FaaSNet's architecture. Figure 4 illustrates the architecture of the FaaS platform running FaaSNet. FaaSNet decentralizes and parallelizes container provisioning[3] across VMs. FaaSNet introduces an abstraction called *function trees* (*FTs*) to enable efficient container provisioning at scale. FaaSNet integrates a *FT manager* component and a *worker* component into our existing FaaS scheduler and VM agent for coordinated FT management. Next, we describe the main components in our FaaS platform.

A *gateway* is responsible for (1) tenant identity access management (IAM) authentication, (2) forwarding the function invocation requests to the FaaS scheduler, and (3) converting regular container images to the I/O efficient data format.

A *scheduler* is responsible for serving function invocation requests. We integrate a FaaSNet *FT manager* into the scheduler to manage *function trees* (§3.2), or *FTs* for short, through FT's `insert` and `delete` APIs. A FT is a binary tree overlay that connects multiple host VMs to form a fast and scalable container provisioning network. Each VM runs a FaaS VM agent, which is responsible for VM-local function management. We integrate a FaaSNet *worker* into the VM agent for container provisioning tasks.

On the function invocation path, the scheduler first communicates with a *VM manager* to scale out the the active VM pool from a free VM pool, if there are not enough VMs or all VMs that hold an instance of the requested function are busy. The scheduler then queries its local FT metadata and sends RPC requests to FaaSNet workers of the FT to start the container provisioning process (§3.3). The container runtime provisioning process is effectively decentralized and parallelized across all VMs in a FT that do not yet have a container runtime locally provisioned. The scheduler sits off the critical path while FaaSNet workers fetch function container layers on demand and creates the container runtime (§3.5) from the assigned peer VMs in parallel.

As described in §2.1, on the function deployment path, the gateway converts a function's regular container image into an *I/O efficient* format (§3.5) by pulling the regular image from a tenant-facing container registry, compresses the image layers block-by-block, creates a metadata file (an image manifest) that contains the format-related information, and writes the converted layers and its associated manifest to an Alibaba Cloud-internal container registry and a metadata store, respectively.

3.2 Function Trees

We make the following design choices when designing FTs. (1) A function has a separate FT; that is, FaaSNet manages FTs at function granularity. (2) FTs have decoupled data plane and control plane; that is, each VM worker in a FT has equivalent, simple role of container provisioning (data plane), and the global tree management (control plane) to the scheduler (§3.3). (3) FaaSNet adopts a balanced binary tree structure that can dynamically adapt to workloads.

These design choices are well aligned with Alibaba Cloud's existing FaaS infrastructure and are attuned to achieve three goals: (1) minimizes the I/O load of container image and layer data downloading on backing container registry, (2) eliminates the tree management bottleneck and data seeding bottleneck of a central root node, and (3) adapts when VMs join and leave dynamically.

Managing Trees at Function Granularity. FaaSNet manages a separate, unique tree for each function that has been invoked at least once and has not been reclaimed. Figure 5 illustrates the topology of a three-level FT that spans five host VMs. Function container images are streamed from the root VM of the tree downwards until reaching the leaf nodes.

Balanced Binary Trees. At FaaSNet's core is a balanced binary tree. In a binary tree, except for the root node and leaf nodes, each tree node (in our case a host VM)[4] has one incoming edge and two outgoing edges. This design can

[3] While this paper mainly focuses on container runtime provisioning, FaaSNet supports provisioning of both containers and code packages.

[4] We use "node"/"VM" interchangeably when describing tree operations.

effectively limit the number of concurrent downloading operations per VM to avoid a network contention. A balanced binary tree with N nodes has a height of $\lfloor log(N) \rfloor$. This is desirable as a balanced binary tree guarantees that the image and layer data of a function container would traverse at most $\lfloor log(N) \rfloor$ hops from the top to the bottom. This is critical as the height of a FT would affect the efficiency of data propagation. Furthermore, the structure of a balanced binary tree can dynamically change in order to accommodate the dynamicity of the workloads. To this end, FaaSNet organizes each FT as a balanced binary tree. The *FT manager* (Figure 4) calls two APIs, `insert` and `delete`, to dynamically grow or shrink a FT.

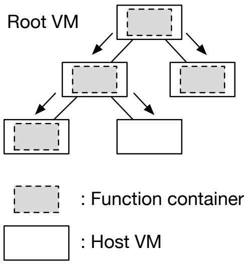


Figure 5: An example FaaSNet FT.

`insert`: The very first node of a FT is inserted as a root node. The FT manager tracks the number of child nodes that each tree node has via BFS (breadth-first search) and stores all nodes that has 0 or 1 child in a queue. To insert a new node, the FT manager picks the first node from the queue as the parent of the new node.

`delete`: The scheduler may reclaim a VM that has been idling for a period of time (15-minute in Alibaba Cloud configuration). Thus, FaaS VMs have a limited lifespan. To accommodate VM leaving caused by reclamation, the FT manager calls `delete` to delete a reclaimed VM. The `delete` operation rebalances the structure of FT if needed. Different from a binary search tree such as an AVL-tree or a red-black tree, nodes in a FT do not have a comparable key (and its associated value). Therefore, our tree-balancing algorithm only needs to hold one invariant—a balancing operation is triggered only if the height difference between any node's left and right subtree is larger than 1. The FT implements four methods to handle all imbalance situations—*left_rotate*, *right_rotate*, *left_right_rotate*, and *right_left_rotate*. Due to the space limit, we omit the details of the tree balancing algorithms. Figure 6 and Figure 7 show the process of *right_rotate* and *right_left_rotate* operations, respectively.

3.3 Function Tree Integration

In this section, we describe how we integrate the FT scheme into Alibaba Cloud's FaaS platform. The integration spans two components of our existing FaaS platform, the scheduler and the VM agent. Specifically, we integrate FaaSNet's FT manager into Alibaba Cloud's FaaS scheduler and FaaSNet's VM worker into Alibaba Cloud's FaaS VM agent, respectively (Figure 4). The scheduler manages VMs of a FT via the FT manager. The scheduler starts a FaaSNet worker on each VM agent. A FaaSNet worker is responsible for (1) serving scheduler's commands to perform tasks of image downloading and container provisioning, and (2) managing the VM's function containers.

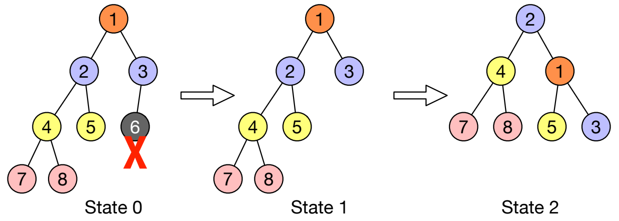


Figure 6: An example *right_rotate* operation. The FT manager detects that Node 6 was reclaimed and calls `delete` to remove it. Removal of Node 6 causes an imbalance, which triggers a *right_rotate* rebalancing operation. The FT manager then performs right rotation by marking Node 2 as the new root and marking Node 5 as Node 1's left subtree.

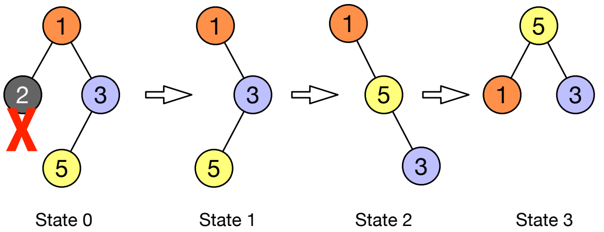


Figure 7: An example *right_left_rotate* operation. FT manager detects that Node 2 gets reclaimed and calls `delete` to remove it. Removal of Node 2 triggers a rebalancing operation. FT manager first right-rotates the right subtree of Node 1 by marking Node 5 as the parent of Node 3. FT manager then performs a *left_rotate* by marking Node 5 as the root.

FT Metadata Management. The scheduler maintains an in-memory mapping table that records the $<function_{ID}, FT>$ key-value pairs, which map a function ID to its associated FT data structure. A FT data structure manages a set of in-memory objects representing functions and VMs to keep track of information such as a VM's *address:port*. The scheduler is sharded and is highly available. Each scheduler shard periodically synchronizes its in-memory metadata state with a distributed metadata server that runs etcd [11].

Function Placement on VMs. For efficiency, FaaSNet allows one VM to hold multiple functions that belong to the same user. Function Compute uses a binpacking heuristic that assigns as many functions as possible in one VM host as long as the VM has enough memory to host the functions. As such, a VM may be involved in the topologies of multiple overlapping FTs. Figure 8 shows an example of a possible FT placement. In order to avoid network bottlenecks, FaaSNet limit the number of functions that can be placed on a VM—in our deployment we set this limit to 20. We discuss a proposal of the FT-aware placement in §5.

Container Provisioning Protocol. We design a protocol to coordinate the RPC communications between the scheduler and FaaSNet VM workers and facilitate container provisioning (Figure 9). On an invocation request, if the scheduler detects that there are not enough active VMs to serve the request

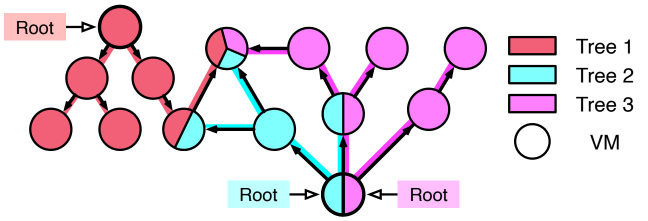


Figure 8: Example function placement on VMs. The color codings of trees and tree edges are red for tree 1 (left), blue for tree 2 (center), and purple for tree 3 (right). Arrows denote provisioning flows.

or all of current VMs are busy serving requests, the scheduler reserves one or multiple new VMs from the free VM pool and then enter the container provisioning process. Without loss of generality, we assume only one VM (VM_1) is reserved in this case. In Step 1, the scheduler creates a new metadata object for VM_1 and inserts it to the FT associated with the requested $function_{ID}$. The scheduler then queries the FT in order to get the *address:port* of the upstream peer VM (VM_2). In Step 2, the scheduler sends the function metadata and *address:port* of VM_2 to VM_1. Once receiving the information, VM_1 performs two tasks: (1) downloads the `.tar` manifest file of the function container image from the metadata store (§3.1), and (2) loads and inspects the manifest, fetches the URLs of the image layers, and persists the URL information on VM_1's local storage. In Step 3, VM_1 replies back to the scheduler that it is ready to start creating the container runtime for the requested function. The scheduler receives the reply from VM_1 and then sends a *create container* RPC request to VM_1 in Step 4. In Step 5 and 6, VM_1 fetches the layers from upstream VM_2 based on the manifest configuration processed in Step 2. In Step 7, VM_1 sends the scheduler an RPC that the container has been created successfully.

FT Fault Tolerance. The scheduler pings VMs periodically and can quickly detect VM failures. If a VM is down, the scheduler notifies the FT manager to perform tree balancing operations in order to fix the FT topology.

3.4 FT Design Discussion

FaaSNet offloads the metadata-heavy management tasks to the existing FaaS scheduler, so that each individual node in a FT serves the same role of fetching data from its parent peer (and seeding data for its child nodes if any). FT's root node does not have a parent peer but instead fetches data from the registry. FaaSNet's FT design can completely eliminate the I/O traffic to the registry, as long as a FT has at least one active VM that stores the requested container. Earlier, our workload analysis reveals that a typical FaaS application would always have a throughput above 0 RPS (§2.2). This implies that, in practice, it is more likely for a request burst to scale out a FT from 1 to N rather than from 0 to N.

An alternative design is to manage the topology at finer-grained layer (i.e., blobs) granularity. In this approach, each individual layer forms a logical layer tree; layers that belong to a function container image may end up residing on different VMs. Note that FaaSNet's FT is a special case of a layer tree model. Figure 10 shows an example. In this example, one VM stores layer files that belong to different function container images. Thus, a network bottleneck may occur when many downstream VM peers are concurrently fetching layers from this VM. This is because many overlapping layer trees form a fully-connected, all-to-all network topology. An all-to-all topology might scale well if VMs are connected with high-bandwidth network. However, the all-to-all topology can easily create network bottlenecks if each VM is resource-constrained, which is our case in Alibaba Cloud. We use small VMs with 2-core CPU, 4 GB memory, and 1 Gbps network in our FaaS infrastructure.

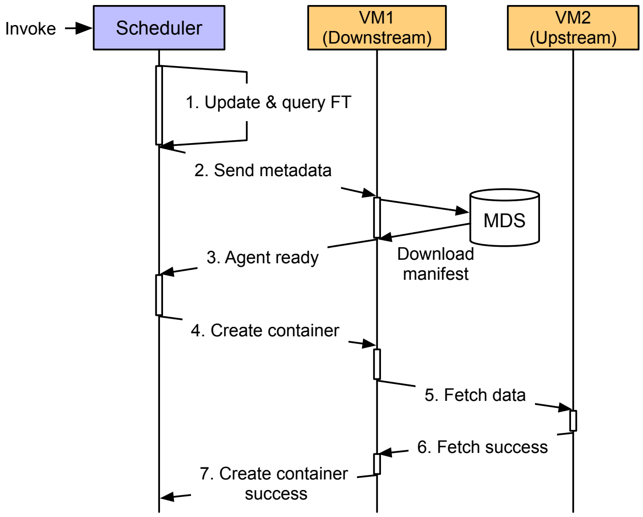


Figure 9: Container provisioning protocol. MDS: metadata store.

Existing container distribution techniques [16, 42] rely on powerful root node to serve a series of tasks including data seeding, metadata management, and P2P topology management. Porting these frameworks to our FaaS platform would require extra, dedicated, possibly sharded, root nodes, which would add unnecessary cost to the provider. FaaSNet's FT design, on the other hand, keeps each VM worker's logic simple while offloading all logistics functions to our existing scheduler. This design naturally eliminates both the network I/O bottleneck and the root node bottleneck. In §4.3 and §4.4 we evaluate and compare FaaSNet's FT design against a Kraken-like approach [16, 21], which adopts a layer-based topology with powerful root nodes.

3.5 Optimizations

We present the low-level optimizations that FaaSNet uses to improve the efficiency of function container provisioning.

I/O Efficient Data Format. Regular `docker pull` and `docker start` are inefficient and time-consuming as the whole container image and the data of all the layers must be downloaded from a remote container registry [37] before

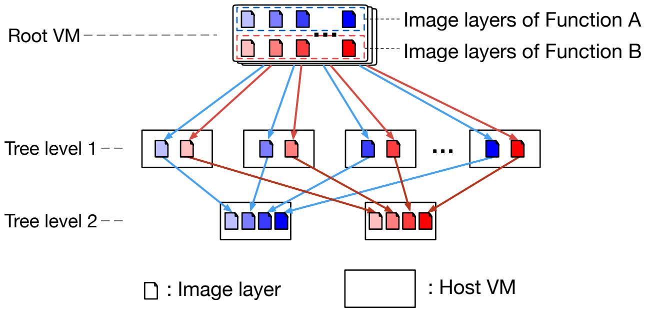


Figure 10: An example tree that manages the topology at layer granularity and relies on root node for data seeding and tree management.

the container can be started. To solve the issue, we design a new block-based image fetching mechanism within Alibaba Cloud. This mechanism uses an I/O efficient compression data file format. Original data is split into fixed-sized blocks and compressed separately. An offset table is used to record the offset of each compressed block in the compressed file.

FAASNET uses the same data format for managing and provisioning code packages. A code package is compressed into a binary file, which will be extracted by VM agent and eventually mounted inside of a function container. FAASNET distributes code packages the same way as it does for container images. §4.5 evaluates the performance benefit of I/O efficient data format on code package provisioning.

On-Demand I/O. For applications that do not need to read all the layers at once on startup, our block-based image fetching mechanism provides them with an option to fetch layer data at fine-grained block level, in a lazy manner (i.e., on-demand), from a remote storage (in our case, a container registry or a peer VM). First, the application, in our case, a FAASNET VM worker, downloads the image manifest file from a metadata store and does an image load locally to load the `.tar` image manifest. Second, it calculates the indices of the first and last (compressed) block and then consults with the offset table to find the offset information. Finally, it reads the compressed blocks and decompresses them until the amount of data that has been read matches the requested length. Since a read to the underlying (remote) block storage device must be aligned to the block boundary, the application may read and decompress more data than requested, causing read amplification. However, in practice, decompression algorithm achieves much higher data throughput than that of a block storage or network. Thus, trading extra CPU overhead for reduced I/O cost is beneficial in our usage scenario. We evaluate the effectiveness of on-demand I/O in §4.6.

RPC and Data Streaming. We build a user-level, zero-copy RPC library. This approach leverages non-blocking TCP `sendmsg` and `recvmsg` for transferring an `struct iovec` incontinuous buffer. The RPC library adds an RPC header directly to the buffer to achieve efficient, zero-copy serialization in the user space. The RPC library tags requests in order to achieve request pipelining and out-of-order receiving, similar to HTTP/2's multiplexing [14]. When a FAASNET worker receives a data block in its entirety, the worker immediately transfers the block to the downstream peer.

4 Evaluation

In this section, we evaluate FAASNET using production traces from Alibaba Cloud's FaaS platform. We also validate FAASNET's scalability and efficiency via microbenchmarks.

4.1 Experimental Methodology

We deploy FAASNET in Alibaba Cloud's Function Compute platform using a medium-scale, 500-VM pool and a large-scale, 1,000-VM pool. We follow the same deployment configurations used by our production FaaS platform: all VMs use an instance type with 2 CPUs, 4 GB memory, 1 Gbps network; we maintain a free VM pool where FAASNET can reserve VM instances to launch cloud functions. This way, the container provisioning latency does not include the time to cold start a VM instance. FAASNET uses a block size of 512 KB for on-demand fetching and streaming. Unless otherwise specified: we use a function that runs a Python 3.8 PyStan application for about 2 seconds; the size of the function container image is 758 MB; the function is configured with 3008 MB memory; each VM runs one containerized function.

System Comparison. In our evaluation, we compare FAASNET against the following three configurations:

1. ***Kraken*:** Uber's P2P-based registry system [16, 21]. We deploy a *Kraken* `devcluster` [17] with one origin node on our resource-constrained VM infrastructure.
2. ***baseline*:** Alibaba Cloud Function Compute's current production setup. *baseline* downloads container images using vanilla `docker pull` from a centralized container registry.
3. ***on-demand*:** An optimized system based on *baseline* but fetches container layer data on demand (§3.5) from the container registry.
4. ***DADI+P2P*:** Alibaba's DADI [1, 42] with P2P enabled. This approach uses one resource-constrained VM as the root node to manage the P2P topology.

Goals. We aim to answer the following questions:

1. Can FAASNET rapidly provision function containers under bursty FaaS workloads with minimum impact on workload performance (§4.2)?
2. Does FAASNET scale with increasing invocation concurrency levels (§4.3)?
3. How does function placement impact FAASNET's efficiency (§4.4)?
4. How does FAASNET's I/O efficient data format perform (§4.5)?
5. How effective is FAASNET's on-demand fetching (§4.6)?

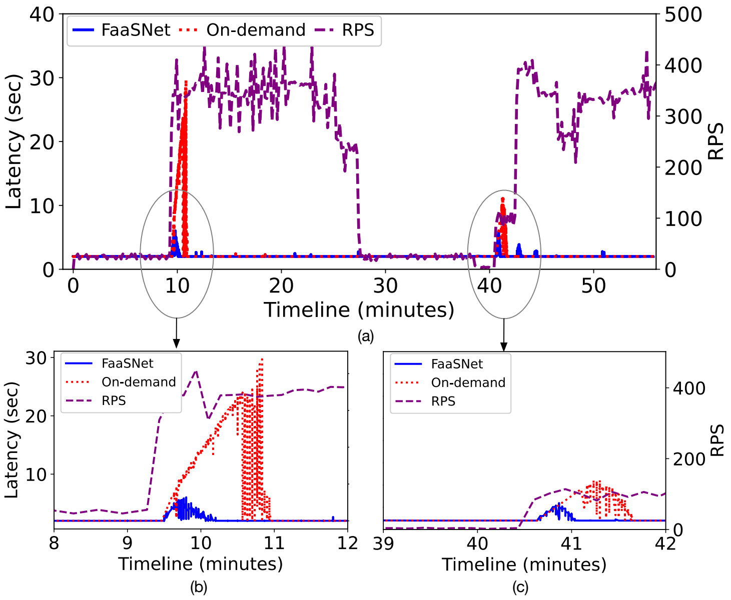


Figure 11: IoT trace timeline.

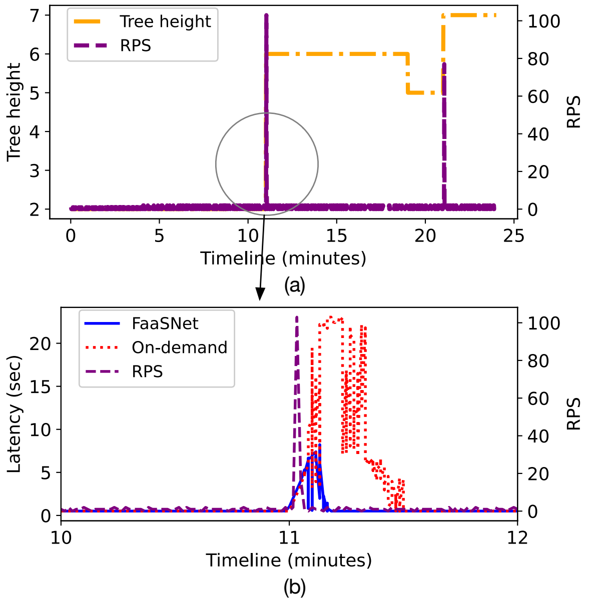
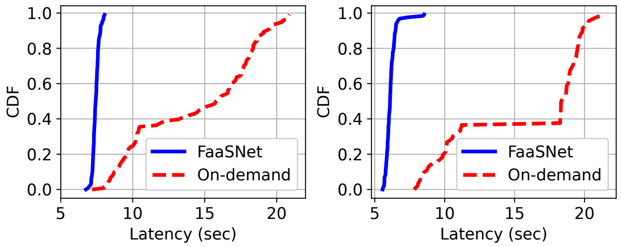


Figure 12: Synthetic trace timeline.



(a) IoT trace.

(b) Synthetic trace.

Figure 13: Distribution of container provisioning latency.

4.2 FaaS Application Workloads

In this section, we evaluate FAASNET using (scaled-down) application traces collected from our production workload (detailed in §2.2).

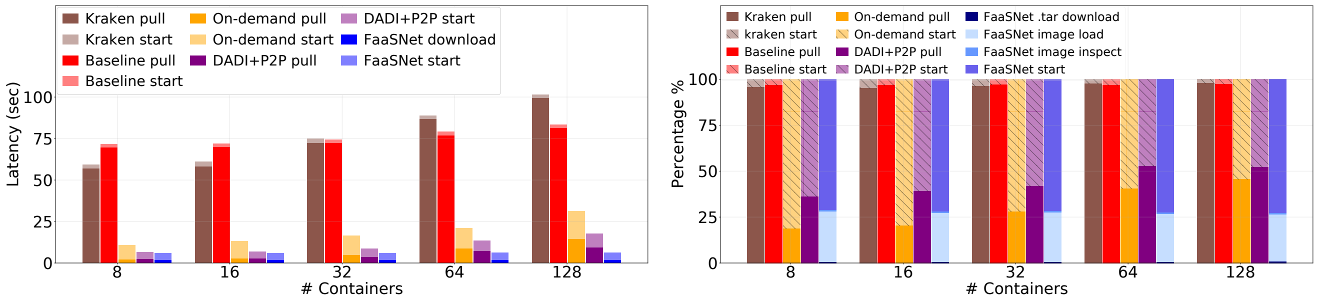
Trace Processing and Setup. We evaluate FAASNET using two FaaS applications: an IoT app and a gaming app. Since the original gaming workload exhibits a gradual ramp-up in throughput (Figure 2(a)), we instead create a synthetic bursty workload based on the gaming workload to simulate a sharp burst pattern for stress testing purpose. Our testing cluster has up to 1,000 VMs, so we scale down the peak throughput of both workload traces proportional (about 1/3 of the original throughput) to our cluster size and shorten the duration from 2 hours to less than 1 hour.

IoT Trace. The IoT trace exhibits two invocation request bursts. The first burst happens at 9 minute and the throughput increases from 10 RPS to 300-400 RPS; the peak throughput lasts for about 18 minutes and returns back to 10 RPS at 28 minute. The second burst happens at 40 minute and the throughput increases to 100 RPS, and then in about 2 minutes, jumps to around 400 RPS. Figure 11(a) plots the 55-minute timeline of the workload's throughput and latency changes.

At 10 minute, the instantaneous throughput increase causes a backlog of function invocation requests at the FaaS scheduler side. Thus, the scheduler scales out the active VM pool by reserving a large number of free VMs from the free VM pool and starts the function container provisioning process. In *baseline* case, all newly reserved VMs start pulling container images from the registry, which creates a performance bottleneck at the registry side. As a result, the application-perceived response time—the end-to-end runtime that includes the container startup latency and the function execution time of around 2 seconds—increases from 2 seconds to about 28 seconds. Worse, the registry bottleneck inevitably prolongs the time that *baseline* requires to bring the response time back to normal. As shown in Figure 11(b), *baseline* finishes the whole container provisioning process and brings the response time back to normal in almost 113 seconds.

In contrast, FAASNET avoids the registry bottleneck—instead of downloading the container image from the registry, each newly reserved VM fetches image data block-by-block from its upstream peer in the FT, forming a data streaming pipeline. As long as a VM fetches enough data blocks, it starts the container. FAASNET reduces the maximum response time from 28 seconds to 6 seconds. Out of the 6 seconds, around 4 seconds are spent on fetching image layers from the upstream peer VM. (We present the container provisioning latency later in Figure 13.) More importantly, FAASNET requires only 28 seconds to bring the service back to normal, an improvement of 4× compared to the *on-demand* case.

Synthetic Trace. In the synthetic trace test, we simulate two function invocation request bursts and evaluate FT's adaptivity. Figure 12(a) shows the timeline of a FAASNET FT's height changes. At 11 minute, the throughput suddenly grows from 1 RPS to 100 RPS. FAASNET detects the burst and



(a) Average function container provisioning latency.

(b) Fraction of time spent at different stages.

Figure 14: Container provisioning scalability test.

rapidly scales the FT from a height of 2 (one root VM and one peer VM) to 7 (82 VMs in total). The FT starts parallel container provisioning instantly at 11 minute and sustains the latency spikes in about 10 seconds (Figure 12(b)). After the first burst, the throughput drops back to 1 RPS. Some VMs become cold and get reclaimed by the VM manager in about 15 minutes since the first burst. The number of VMs gradually reduces to 30 before the second burst arrives. Correspondingly, the height of the FT reduces from 6 to 5 (Figure 12(a)). When the second burst comes at 21 minute, the FT manager decides to grow the FT by adding another 62 VMs. With a total of 102 VMs, the height of the FT reaches up to 7 for serving the concurrent requests of the second burst.

Container Provisioning Cost We next analyze the container provisioning latency seen in the two workloads. As shown in Figure 13, since the registry in *on-demand* incurs a performance bottleneck, *on-demand* sees highly variant container provisioning latency, ranging from around 7 seconds to as high as 21 seconds. About 80% of the containers take at least 10 seconds to start. The container startup latency is highly predictable in FAASNET, with significantly less variation. For the synthetic workload, around 96% of the functions require only 5.8 seconds to start. For the IoT workload, almost all the functions start execution within a short time range between 6.8-7.9 seconds. This demonstrates that FAASNET can achieve predictable container startup latency.

4.3 Scalability and Efficiency

Next, we evaluate FAASNET's scalability and efficiency via microbenchmarking.

Scaling Function Container Provisioning. In this test, we measure the time FAASNET takes to scale from 0 to N concurrent invocation requests, where N ranges from 8 to 128. Each invocation request creates a single container in a VM. Figure 14 reports the detailed results. As shown in Figure 14(a), *Kraken* performs slightly better than *baseline* under 8 and 16 concurrent requests but scales poorly under 32-128 concurrent requests. This is because *Kraken* distributes containers at layer granularity using a complex, all-to-all, P2P topology, which creates bottlenecks in the VMs. *Kraken* takes 100.4 seconds to launch 128 containers.

baseline achieves slightly better scalability than *Kraken*. The average container provisioning latency reaches up to 83.3 seconds when *baseline* concurrently starts 128 functions. Two factors contribute to the delay: (1) the registry becomes the bottleneck, and (2) *baseline*'s `docker pull` must pull the whole container image and layers (758 MB worth of data) from the registry and extract them locally.

Adding *on-demand* container provisioning to *baseline* improves the latency significantly. This is because *on-demand* eliminates most of the network I/Os for image layers that will not be instantly needed container startup. Despite pulling much less amounts of data from the registry, *on-demand* still suffers from the registry bottleneck; provisioning 128 function containers requires 2.9× longer time than provisioning 8 containers in *on-demand* system.

DADI+P2P enables VMs to directly fetch image layers from peers, further avoiding downloading a large amount of layer blocks from the registry. However, *DADI+P2P* still has two bottlenecks: one at the registry side—image pulls are throttled at the registry, and layer-wise extract operation may also be delayed in a cascading manner by local VMs; the other bottleneck at the P2P root VM side—in addition to seeding data, the root VM in *DADI+P2P* is responsible for a series of extra tasks such as layer-tree topology establishment and coordination, thus forming a performance bottleneck. This can be evidenced from Figure 14(b) that the fractions of *DADI+P2P*'s image pull and container start maintain at the same level when scaling from 64 to 128 function starts.

Figure 14(a) shows that FAASNET scales perfectly well under high concurrency and achieves a speedup of 13.4× than *baseline* and 16.3× than *Kraken*. FAASNET is 5× and 2.8× faster than *on-demand* and *DADI+P2P*, respectively. As shown in Figure 14(b), FAASNET's average container provisioning latency is dominated by two operations: image load and container start. FAASNET eliminates the bottleneck on both the two operations—on image load, FAASNET enables decentralized image loading (functionality-wise equivalent to image pull) at each VM by allowing each FAASNET worker to fetch the image manifest from the metadata store (with negligible overhead) and then starting the image loading process locally in parallel; on container start, each FAASNET VM

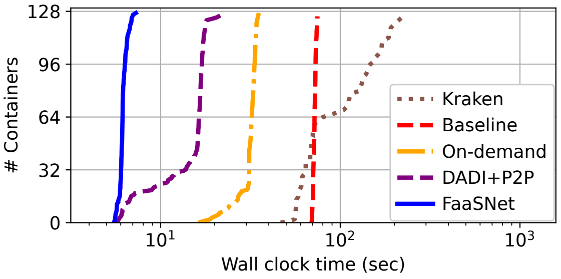


Figure 15: Container provisioning scalability test: wall clock time (X-axis) for starting N functions (Y-axis).

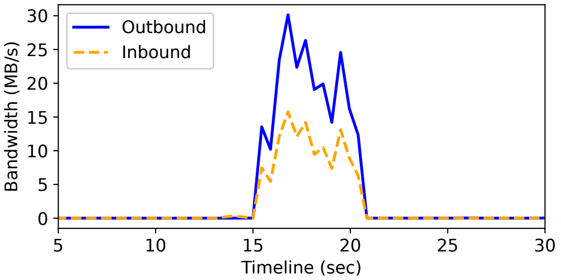


Figure 16: A timeline of the VM network bandwidth usage.

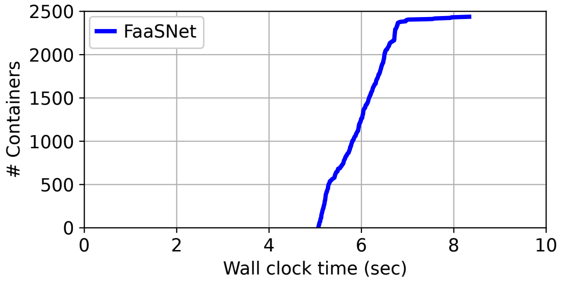


Figure 17: Large-scale function container provisioning: the wall clock time (X-axis) for starting N functions (Y-axis).

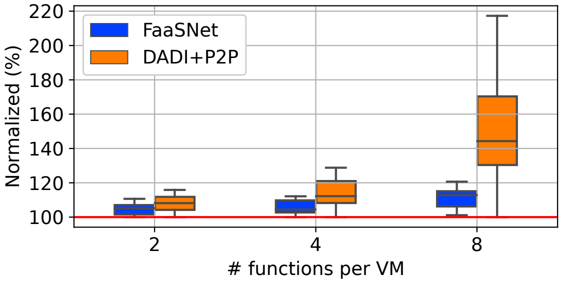


Figure 18: Container provisioning latency as a function of various function placement situations. For FaaSNet and *DADI+P2P*, the latency is normalized to that of provisioning a single container in one VM in their own case.

worker directly fetches layer blocks from peer VM and starts the function container once enough blocks are fetched. With all these optimizations, FaaSNet maintains almost identical latency when scaling from 8 to 128 function startups.

Function Container Provisioning Pipeline. We next examine how long the whole container provisioning process spans. Figure 15 plots the timeline process that each system goes through to start N function containers. We only report the 128-function concurrency case. We observe that FaaSNet starts the first function at 5.5 second and the 128^{th} function at 7 second respectively. The whole container provisioning process spans a total of 1.5 seconds. Whereas *on-demand* and *DADI+P2P* span a total duration of 16.4 and 19 seconds, respectively. Specifically, it takes *DADI+P2P* a total of 22.3 seconds to start all the 128 containers, which is 14.7× slower than that of FaaSNet. This demonstrates that FaaSNet's FT-based container provisioning pipeline incurs minimum overhead and can efficiently bring up a large amount of function containers almost at the same time.

Figure 16 shows the bandwidth usage timeline for a VM that we randomly select from the 128-function concurrency test. Recall that a FaaSNet worker along a FT path (i.e., not the root VM nor the leaf VM) performs two tasks: (1) fetches layer data from the upstream VM peer, and (2) seeds layer data to the two children VM peers in its downstream paths. We observe that the bandwidth usage of the inbound connection (fetching layers from upstream) is roughly half of that of the two outbound connections (sending layers to downstreams) during container provisioning. The aggregate peak network bandwidth is 45 MB/s, which is 35.2% of the maximum network bandwidth of the VM. We also observe that, the outbound network transfer is almost perfectly aligned with the inbound network transfer, again demonstrating the efficacy of FaaSNet's block-level data streaming scheme.

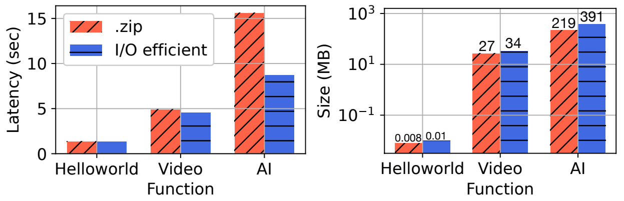
Large-Scale Function Startup. In this test, we create 1,000 VMs and concurrently invoke 2,500 functions on them. Each function uses a container of 428 MB and is configured to run with 1024 MB memory. Each VM runs two or three functions in this test. Figure 17 shows that all function containers finish provisioning and start running between 5.1 second and 8.3 second, again demonstrating FaaSNet's superb scalability. None of *on-demand* and *DADI+P2P* finishes the test due to timeout errors.

4.4 Impact of Function Placement

We conduct a sensitivity analysis to quantify the impact of function placement on container provisioning. In this test, we concurrently invoke 8 functions on N VMs, where N varies from 4 to 1. Each function has a different container (75.4 MB) and is configured to use 128 MB function memory (since a VM has 4 GB memory, it is allowed to host as much as 20 functions with 128 MB memory). We compare the container provisioning latency between FaaSNet and *DADI+P2P*. As shown in Figure 18, *DADI+P2P* sees much higher latency variation when 4 functions and 8 functions are placed on the same VM, because *DADI+P2P*'s root VM is overloaded by the establishment processes of many small layer trees.

4.5 I/O Efficient Data Format

We next evaluate how the I/O efficient format helps with code package provisioning. We choose three functions: a simple, Python-written HelloWorld function that sleeps for 1 second



(a) End-to-end latency.

(b) Code package size.

Figure 19: End-to-end invocation latency and code package size comparison between I/O efficient format and `.zip`.

(Helloworld), an FFmpeg video encoding function (Video), and a TensorFLow Serving function (AI), and compare FaaSNet's I/O efficient format with the baseline `.zip` format.

Figure 19(a) plots the end-to-end function invocation performance including the latency of code package downloading and function duration. Our I/O efficient format performs the same as `.zip` for Helloworld, since Helloworld's code package has only 11 KB in size (Figure 19(b)). The I/O efficient format achieves better performance compared to `.zip` for Video and AI since the I/O efficient format fetches data on demand rather than extracting all data as `.zip` does. Figure 19(b) shows the code package sizes. Functions have a larger code package size when using I/O efficient format, because I/O efficient format's compression incurs extra storage overhead.

4.6 On-Demand I/O: Sensitivity Analysis

Finally, we evaluate on-demand I/O and compare the impact of block sizes on read amplification. With on-demand fetching, a FaaSNet worker only needs to fetch enough layer data blocks in order to start the function container. We choose three different function container images: (a) a 195 MB helloworld image with a Python 3.9 runtime pulled from Docker Hub; (b) a 428 MB PyStan image based on an AWS Lambda Python 3.8 base image; and (c) a 728 MB PyStan image based on an Alibaba Cloud Python 3.8 base image.

As shown in Figure 20, on-demand fetching can reduce the amount of data transferred via network. The reduction is especially profound for image b and c, because base images are dependency-heavy and are commonly used in the image building process. For example, with a block size of 512 KB (the block size configuration that we use in our evaluation), on-demand fetching sees a 83.9% reduction in network I/Os, compared to that of regular `docker pull`.

We also observe different levels of read amplification under different block sizes. This is because the starting and ending offset position is likely to be misaligned with the boundary of the (compressed) blocks in the underlying block device, the larger the block size is, the more useless data the FaaSNet worker may read from the starting and ending blocks. The actual amount of data read (for starting a container) after decompression is even smaller, indicating that most of the dependencies included in the original container image is not used at the container startup phase. Exploring optimization to reduce the read amplification is part of our future work.

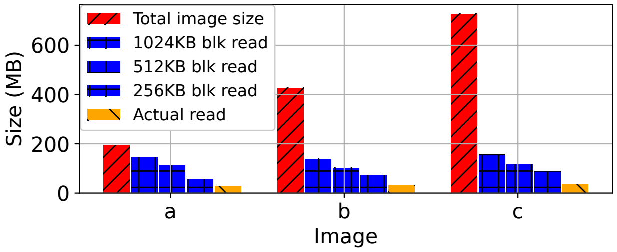


Figure 20: Amounts of data fetched (on-demand) as a function of block sizes. Left-most bar in each bar cluster represents the size of the original container image; right-most bar in each bar cluster represents the actual amount of data read from the data blocks fetched via network.

5 Discussion

In this section, we discuss the limitations and possible future directions of FaaSNet.

FT-aware Placement. When the number of functions grows, the contention of network bandwidth ensues. Though §4.4 proves it less a concern in FaaSNet than prior work, for the sake of safety in production, we program the system to avoid co-locating multiple functions if the cluster resources permit. Anticipating a future demand increase of custom containers, we plan to address the problem by extending the container placement logic. The general goal is to balance the inbound and outbound communication of each VM when multiple functions are being provisioned. Intuitively, by adjusting container placement, we can control the number of FTs that a VM is involved in and the role (e.g., leaf vs. interior node) a VM serves, and thus the bandwidth consumption. A further optimization is to co-locate functions that share common layers, so they could reduce the amount of data transfer.

Multi-Tenancy. As mentioned, Alibaba Cloud achieves strong, tenant-level function isolation using containers and VMs. As such, our FaaS platform cannot share VMs among tenants. This means that FaaSNet's FTs are naturally isolated between different tenants. Porting FaaSNet to other secure and lightweight virtualization techniques [24, 45, 52, 57] is our ongoing work.

FaaSNet for Data Sharing. Technically, our work can enable the sharing of container images among VMs through P2P communication, There is potentiality for it to generalize to a broader scope: data sharing for general container orchestration systems such as Kubernetes [27]. Such a need is arising in FaaS platforms with the emergence of data-intensive applications, such as matrix computation [31, 51], data analytics [39, 49], video processing [26, 35], and machine learning [30, 38], etc. Most of them rely on a centralized storage for data exchange, which is a similar bottleneck as the container registry in our work. Hence we believe the design of FaaSNet can also accelerate data sharing, only with two additional challenges: (1) how to design a primitive interface for users; (2) how to adapt the tree management algorithms for more frequent topology building and change. We leave the exploration as a future work.

Adversarial Workloads. Extremely short-lived functions with a duration at sub-second level and sparse invocations may be adversarial to FAASNET and custom-container-based FaaS platforms. Function environment caching and pre-provisioning [22, 48, 50] can be used to handle such workloads but with extra infrastructure-level costs.

Portability. FAASNET is transparent to both upper-level FaaS applications and underlying FaaS infrastructure. It reuses Function Compute's existing VM reclaiming policy and could be applied to other FaaS platforms without introducing extra system-level costs. Porting FAASNET to Alibaba Cloud's bare-metal infrastructure is our ongoing work.

6 Related Work

Function Environment Caching and Pre-provisioning. FaaS applications face a notoriously persisting problem of high latency—the so-called "cold start" penalty—when function invocation requests must wait for the functions to start. Considerable prior work has examined ways to mitigate the cold start latency in FaaS platforms. FaaS providers such as AWS Lambda and Google Cloud Functions pause and cache invoked functions for a fixed period of time to reduce the number of cold starts [22, 55, 56]. This would, however, increase the TCO for providers. To reduce such cost, researchers propose prediction methods that pre-warm functions just in time so that incoming recurring requests would likely hit on warm containers [50]. SAND shares container runtimes for some or all of the functions of a workflow for improved data locality and reduced function startup cost [25]. SOCK caches Python containers with pre-imported packages and clones cached containers for minimizing function startup latency [48]. PCPM pre-provisions networking resources and dynamically binds them to function containers to reduce the function startup cost [46]. While function requests can be quickly served using pre-provisioned, or cached, virtualized environments, these solutions cannot fundamentally solve the issue of high costs incurred during function environment provisioning.

Sandbox, OS, and Language-level Support. A line of work proposes low-level optimizations to mitigate FaaS cold start penalty. Catalyzer [34] and SEUSS [29] reduce the function initialization overhead by booting function instances from sandbox images created from checkpoints or snapshots. Systems such as Faasm [53] and [28] leverage lightweight language-based isolation to achieve speedy function startups. Unlike FAASNET, these solutions either require modified OSes [29, 34] or have limited compatibility and usability in terms of programming languages [28, 53].

Container Storage. Researchers have looked at optimizing container image storage and retrieval. Slacker speeds up the container startup time by utilizing lazy cloning and lazy propagation [37]. Images are stored and fetched from a shared network file system (NFS) and referenced from a container registry. Wharf [61] and CFS [44] store container image layers in distributed file systems. Bolt provides registry-level caching for performance improvement [43]. These work are orthogonal in that FAASNET can use them as backend container stores. Kraken [16] and DADI [42] use P2P to accelerate container layer distribution. These systems assume a static P2P topology and require dedicated components for image storage, layer seeding, or metadata management, which leave them vulnerable to high dynamicity (demanding high adaptability of the network topology) and unpredictable bursts (requiring highly scalable container distribution).

AWS Lambda Containers. AWS announced the launch of container image support for AWS Lambda [19] on December 01, 2020. Limited information was revealed via a re:Invent 2020 talk [6] about this feature: AWS uses multi-layer caching to aggressively cache image blocks: (1) microVM-local cache, (2) shared, bare-metal server cache, and (3) shared, availability zone cache. The solution, while working for powerful, bare-metal server-based clusters that can co-locate many microVMs [24], is not suitable for our FaaS platform, which is based on thousands of small VMs managed by Alibaba Cloud's public cloud platform.

P2P Content Distribution. VMThunder uses a tree-structured P2P overlay for accelerating VM image distribution [59]. A BitTorrent-like P2P protocol is proposed for achieving similar goals [33]. Bullet uses an overlay mesh for high-bandwidth, cross-Internet file distribution [41]. FAASNET builds on these works but differs with a new design that is attuned to the FaaS workloads.

7 Conclusion

Scalable and fast container provisioning can enable fundamental elasticity for FaaS providers that support custom-container-based cloud functions. FAASNET is the first system that provides an end-to-end, integrated solution for FaaS-optimized container runtime provisioning. FAASNET uses lightweight, decentralized, and adaptive function trees to avoid major platform bottlenecks. FAASNET provides a concrete solution that is attuned to the requirements of a large cloud provider's FaaS platform (Alibaba Cloud Function Compute). We show via experimental evaluation that FAASNET can start thousands of large function containers in seconds. Our hope is that this work will make container-based FaaS platforms truly elastic and open doors to a broader class of dependency-heavy FaaS applications including machine learning and big data analytics.

To facilitate future research and engineering efforts, we release the source code of FAASNET's FT prototype as well as an anonymized dataset containing production FaaS cold start traces collected from Alibaba Cloud Function Compute at: https://github.com/mason-leap-lab/FaaSNet.

Acknowledgments

We are grateful to our shepherd, Jon Crowcroft, and the anonymous reviewers, for their valuable feedback and suggestions. We would like to thank Benjamin Carver for his proofreading, and Jianlie Zou, Qi Sun, and Yifan Yuan for their kind help on this work. This work is sponsored in part by NSF under an NSF CAREER Award CNS-2045680, CCF-1919075, and CCF-1919113; Ao Wang is supported by Alibaba Group through the Alibaba Research Intern Program.

References

[1] Alibaba: Accelerated Container Image. https://github.com/alibaba/accelerated-container-image.

[2] Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.

[3] Alibaba Cloud Function Compute Custom Container Runtime. https://www.alibabacloud.com/help/doc-detail/179368.htm.

[4] Alibaba Cloud Function Compute's custom container doc. https://github.com/awesome-fc/custom-container-docs.

[5] AWS Lambda. https://aws.amazon.com/lambda/.

[6] AWS re:Invent 2020: Deep dive into AWS Lambda security: Function isolation. https://www.youtube.com/watch?v=FTwsMYXWGB0.

[7] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[8] Deploy Lambda functions with .zip file archives. https://docs.aws.amazon.com/lambda/latest/dg/python-package.html.

[9] Docker. https://www.docker.com/.

[10] Dragonfly: An Open-source P2P-based Image and File Distribution System. https://d7y.io/en-us/.

[11] etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.

[12] Google Cloud Functions. https://cloud.google.com/functions/.

[13] Google Cloud Run. https://cloud.google.com/run.

[14] HTTP/2. https://http2.github.io/.

[15] Knative: Kubernetes-based platform to deploy and manage modern serverless workloads. https://knative.dev/.

[16] Kraken: A P2P-powered Docker registry that focuses on scalability and availability. https://github.com/uber/kraken.

[17] Kraken devcluster deployment. https://github.com/uber/kraken/tree/master/examples/devcluster.

[18] Kubernetes: Performing a Rolling Update. https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/.

[19] New for AWS Lambda – Container Image Support. https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/.

[20] OpenFaaS. https://www.openfaas.com/.

[21] Uber Releases Kraken: An Open Source P2P Docker Registry. https://www.infoq.com/news/2019/04/uber-kraken-p2p-docker/.

[22] Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[23] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[24] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[25] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, 2018. USENIX Association.

[26] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. ACM.

[27] The Kubernetes Authors. Production-grade container orchestration - kubernetes. https://kubernetes.io/.

[28] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[29] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 13–24, New York, NY, USA, 2019. ACM.

[31] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Benjamin Carver, Jingyuan Zhang, Ao Wang, and Yue Cheng. In search of a fast and efficient serverless dag engine. In *4th International Parallel Data Systems Workshop (PDSW 2019)*, 2019.

[33] Zhijia Chen, Yang Zhao, Xin Miao, Ying Chen, and Qingbo Wang. Rapid provisioning of cloud infrastructure leveraging peer-to-peer networks. In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, pages 324–329, 2009.

[34] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

[36] Jim Gray. Why do computers stop and what can be done about it?, 1985.

[37] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, 2016. USENIX Association.

[38] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2021)*, June 2021.

[39] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, 2017. ACM.

[40] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[41] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 282–297, New York, NY, USA, 2003. Association for Computing Machinery.

[42] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI: Block-level image service for agile and elastic application deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 727–740. USENIX Association, July 2020.

[43] Michael Littley, Ali Anwar, Hannan Fayyaz, Zeshan Fayyaz, Vasily Tarasov, Lukas Rupprecht, Dimitrios Skourtis, Mohamed Mohamed, Heiko Ludwig, Yue Cheng, and Ali R. Butt. Bolt: Towards a scalable docker registry via hyperconvergence. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 358–366, 2019.

[44] Haifeng Liu, Wei Ding, Yuan Chen, Weilong Guo, Shuoran Liu, Tianpeng Li, Mofei Zhang, Jianxing Zhao, Hongyin Zhu, and Zhengyi Zhu. Cfs: A distributed file system for large scale container platforms. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1729–1742, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[46] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna

Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[47] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[48] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, 2018. USENIX Association.

[49] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association.

[50] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[51] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[53] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[54] Markus Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast! https://goo.gl/zvqtBP.

[55] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[56] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[57] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 199–211, New York, NY, USA, 2018. Association for Computing Machinery.

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for inmemory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[59] Zhaoning Zhang, Ziyang Li, Kui Wu, Dongsheng Li, Huiba Li, Yuxing Peng, and Xicheng Lu. Vmthunder: Fast provisioning of large-scale virtual machine clusters. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3328–3338, 2014.

[60] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2019.

[61] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 174–185, New York, NY, USA, 2018. Association for Computing Machinery.