Fast and Efficient Container Startup at the Edge via Dependency Scheduling

Silvery Fu *UC Berkeley*
Radhika Mittal *UIUC*
Lei Zhang *Alibaba Group*
Sylvia Ratnasamy *UC Berkeley*

Abstract

Containers are becoming the canonical way of deploying compute tasks at the edge. Unfortunately, container *startup* latency and overhead remain high, limiting responsiveness and resource efficiency of edge deployments. This latency comes mostly from fetching container dependencies including system libraries, tools, configuration files, and data files.

To address this, we propose that schedulers in container orchestrators take into account a task's dependencies. Hence, in *dependency scheduling*, the scheduler tries to place a task at a node that has the maximum number of the task's dependencies stored locally. We implement dependency scheduling within Kubernetes and evaluate it through extensive experiments and measurement-driven simulations. We show that, for typical scenarios, dependency scheduling improves task startup latency by 1.4-2.3x relative to current dependency-agnostic schedulers. Our implementation of dependency scheduling has been adopted into the mainline Kubernetes codebase.

1 Introduction

Application containers and Kubernetes are emerging as the canonical way of deploying services at the edge [8, 10, 12, 16, 17]. Unfortunately, container startup latency can significantly limit the efficiency of short tasks. At the same time, a growing number of edge computing workloads are characterized by short task times (*e.g.,* processing periodic updates from IoT sensors) and/or the need for rapid response times (*e.g.,* robot motion, self-driving cars, factory automation [3, 7, 11]). For these and other applications, reducing container startup latency is increasingly important to ensure low end-to-end latency.

In current deployments, high startup latency comes primarily from the time it takes to fetch and install container *dependencies* on the host machine at which the task will run. These dependencies include system libraries, tools, configuration files, and data files that must be present on the host machine before the container is launched.

In this paper, we ask whether *scheduling* can be leveraged to reduce this startup latency. We have an extensive literature on (and practice of) schedulers that are designed to improve task performance but these have typically focused on improving the task's processing time – e.g., scheduling to avoid contention over shared resources [27], to improve data locality [37], and so forth. Given the above trends, we propose extending the traditional view of scheduling to also improve task *launch* time. To achieve this, we propose that task dependencies be treated as another dimension to resource consumption and that schedulers take into account a task's dependencies when placing tasks. Specifically, we propose that a scheduler should aim to place a task T at the node that maximizes the amount of T's dependencies that are already present at the node, thereby reducing the task startup time. We refer to this approach as *dependency scheduling*.

We propose two designs for dependency scheduling that explore different trade-offs. Our first design treats the container image in its entirety as a dependency and hence the scheduler attempts to place a task T at the node that has the maximum overlap (in bytes) between the images it has cached locally and those requested in T. We call this the *image-match* policy. Image-match is very simple to implement and dramatically reduces startup time when a match is found. However, because it does not consider the internal composition of an image, it cannot improve startup latency in situations when two images only *partially* overlap in their internal dependencies.

Our second design avoids these missed opportunities by tracking and matching dependencies at the finer granularity of the *layers* that constitute the image. Hence, our *layer-match* policy places a task T at the node that has the maximum overlap with T's layers (in bytes). Layer match is driven by the intuition that container technology, as it simplifies package reuse, has encouraged non-trivial overlap in the dependencies of different tasks; layer-match exploits this overlap.

We implement dependency scheduling with both policies in Kubernetes, modifying the Kubernetes scheduler, internal APIs, and node agent to support image and layer awareness. We evaluate our scheduling schemes using extensive measurement-driven simulation and experiments. We show that dependency scheduling substantially improves task startup latency: *e.g.,* dependency awareness leads to a 1.4-2.3x reduction in startup latency relative to dependency-agnostic schedulers while introducing as little as 0.3ms in scheduling overhead. Further, we show that the benefits of dependency scheduling arise not only because it reduces the latency and overhead associated with pulling images but also because of its ability to pack more images into its local image storage.

We have shared our scheduler implementations with the Kubernetes developer community: our image-match scheduler has been incorporated into the mainline Kubernetes codebase as a default scheduling policy and has been used in production [1], while layer-match is currently under review [15], corroborating the relevance and practical nature of dependency scheduling.

2 Motivation

We give an overview of containers, container dependencies (§2.1), and Kubernetes (§2.2); we then motivate dependency scheduling for container orchestration at the edge.

2.1 Containers, Images, and Layers

Containers are based on lightweight OS-level virtualization technology that isolates and manages an application's resource usage, and optionally provide tools for managing the application's dependencies. Containers offer two major benefits: lightweight resource isolation and container *images*.

The latter allows developers to package and distribute applications using a standard format. An application's image includes all its dependencies, including the code, binaries, system tools, and configurations files. An image is read-only, copy-on-write, and can be shared by multiple containers: when a container wants to apply changes to the image, the target files or directories will be copied to the container's own independent layer, as described later in this section.

To use an image, users specify the image name in the container request. The container runtime *normalizes* the specified image name – *e.g.,* replacing a default generic image name with the specific name of the latest version. It then resolves the image name to get its constituent *layers* (described below) and pulls them from the image repository if they are not already cached. Once the entire image has been retrieved, the container is installed and booted.

Containers are backed by a layered file-system. Each layer encapsulates a set of files and directories that are put together when the image is built and is associated with a collision-resistant hash *digest* taken over its content. Every layer can be uniquely identified using its digest. Tracking layer digests decouples the image contents from the image name. This allows users to rename images without invalidating the entire image cache and is a common practice today.

2.2 Container Orchestration using Kubernetes

While the concept of dependency scheduling is a general one, we focus on its application to Kubernetes [14] (henceforth k8s) as the latter is widely used and open-source. Figure 1 shows a high-level view of the relevant components in k8s. The k8s master node has two main components: (1) an API server, which interfaces with the user and is backed by a distributed key-value store that maintains cluster state, and (2) the scheduler. Incoming requests for executing a job – called *pod* requests – are submitted to the API server. A pod request consists of one or more tasks, each running in a separate container. The API server communicates the request parameters, including required image names, to the scheduler.

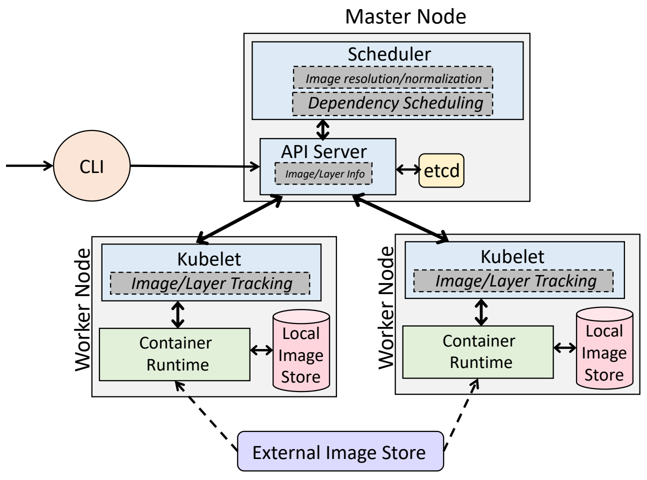

Figure 1: Overview of Kubernetes system architecture with the dependency scheduling extension (in dotted boxes).

Each worker node runs (1) the "kubelet" agent that interacts with the master node, and (2) a container runtime engine (such as Docker) that manages the lifecycle of a container: creation, removal, pausing, and monitoring. The container runtime interacts with the kubelet via the container runtime interface. Each worker node has its own local image store where it can cache images. The required layers that are not already cached in the local image store are fetched from an external image store (such as Amazon's Elastic Container Registry) by the container runtime.

2.3 The case for dependency scheduling at the edge

We now examine the case for dependency scheduling and argue that it is well suited to emerging edge workloads/trends such as connected vehicles [3], smart buildings [6], robotics [7, 26], and IoT [11]. Our discussion in this section provides a highly simplified model that ignores many details of how dependency scheduling works; the remainder of the paper considers the relevant implementation aspects in detail.

We start by addressing when dependency scheduling is needed or useful. We can view a task's completion time, T, as $T = S + R$ where S is the time required to start/launch the task and R is the application running time required once the task is launched. We further assume that S is dominated by the time it takes to download and install the task's container image (we validate this latter assumption empirically in §4).

Our observation is:

Dependency scheduling is useful when S is a non-trivial portion of T (i.e., $S \propto R$). As prior works have noted [28, 32, 33], this regime is becoming relevant given there has been a trend towards shorter tasks. More generally, we expect that the trend towards short tasks will only increase with the deployment of IoT applications in which a large number of sensors periodically report to a backend edge-based service. Further, we conjecture the reason why dependency locality did not attract much attention in practice and in literature is because R was typically large (*e.g.,* long-running services and batch jobs) and thereby startup time was negligible.

Assuming the above condition holds, we next examine when dependency scheduling is *effective*. As mentioned earlier, dependency scheduling aims to reduce S by caching previously used images at the nodes and scheduling tasks at nodes that cache either the task's entire image (our image-match policy) or a subset of the layers in the image (our layer-match policy). The main parameters that impact the effectiveness of this policy are: N, the number of nodes in the cluster; C, the size of the cache at each node; L, the total size of popular layers involved for the workload in question. Then, if:

(a) the total size of the popular layers L is less than the layer cache size of a single node C. Then, at the steady-state, every node is able to store all popular layers in its cache. Hence the dependency scheduling policies (image-match and layer-match alike) should perform similarly as the agnostic policy, because even if the policy randomly picks a node, the node would have the layers.

(b) the total size of the popular layers L exceeds the layer cache size of a single node C; but is less than or close to the total layer cache size of all nodes combined ($N \cdot C$). Intuitively, dependency scheduling wins in this case because, although popular layers cannot be cached on *every* node, they can be cached on *some* nodes in the cluster. Dependency scheduling identifies these nodes to improve startup time S.

For example, consider the Connected Vehicle Platform (CVP) [3] in which each user/vehicle has a different image with some unique layers, *e.g.,* a unique machine learning model customized to each vehicle's travel and/or application stacks developed for each car model and make. To get a sense of N, C, and L: we consider reports published by Ericsson that cite 4 million connected cars as using their platform [5].

Assuming just 0.1% of these cars are active at any time and issuing one update every 1 seconds, we'd see a total load of 4,000 requests per second on a CVP cluster ($L = 4000$). Handling this load would require a cluster of $N = 40$ nodes if we assume each node can run 100 containers (the latter from a target provided by the Kubernetes community [13]). Finally, if we assume each node has 32G of storage (a typical disk space reserved for rootfs) and each image in the CVP contains a customized ML model sized 250MB (YOLO v3 [9, 34]); assuming the image size equals to this size). This gives us $C = 128$ images and hence $N \times C = 5120$. In this scenario, no single node can store all 4,000+ layers, but the CVP cluster in its entirety could do so comfortably, and hence dependency scheduling can greatly reduce the startup time S associated with user requests.

Algorithm 1 Dependency Scheduling

```
1: at the cluster scheduler:
2: for each job $j$ queued do
3:     /*on nodes meeting resource constraints*/
4:     for $n$ in nodes do
5:         score[$n$] = size($|dep(n) \cap dep(j)|$)
6:     end for
7:     /*tie-break with other scheduling criteria*/
8:     $n^*$ = $\arg\max_n$ score[$n$]; bind($n^*$, $j$)
9: end for
```

(c) the total size of the popular layers L is much larger than the total layer cache size of all nodes combined ($N \cdot C$). In this scenario, dependency scheduling performs the same as agnostic policy due to low layer cache-hit ratio.

3 Design and Implementation

We discuss our overall approach (§3.1) and then describe the detailed design of our image-match (§3.2) and layer-match (§3.3) scheduling.

3.1 Design Rationale

Our approach aims to avoid pulling dependencies altogether by modifying the scheduler to place tasks at nodes where some or all of a task's dependencies are already present. The benefit of this scheduler-based approach is that it requires no change to the infrastructure hardware (servers or networks), nor to containerized applications, restricting all changes to the container orchestrator (*e.g.,* k8s).

This dependency scheduling policy is presented in Algorithm 1, where `dep()` extracts the dependency information (i.e., as image or layer) from a node or a job. We rank nodes based on how much their locally-stored dependencies overlap with those of the request. The degree of overlap depends on the granularity of the dependencies we consider.

We propose images and layers as two candidate definitions of dependencies since these are common concepts already present in applications and container frameworks (though not used by the framework's schedulers) and hence they simultaneously capture the trade-off between coarse vs. fine-grained dependencies and are practical for implementation.

3.2 Image-match

Container image is the package that includes all application dependencies (see §2.1). The changes for supporting image-match are as follows.

Kubelets. The container runtime in the worker nodes already tracks the image state (names and sizes of the cached images). We simply extend the kubelet to retrieve this state from the container runtime and peoriodically communicate it to the API server as described next.

API-server. Currently, the interface between the API-server and the kubelet lacks image-awareness. We extend the RPCs between the kubelet and the API-server to communicate the image state. Likewise, we also extend the global cluster state that backs the API-server to store this per-node image state.

Scheduler. We extend the scheduler to implement image-match using the per-node image information stored in the global cluster state. This involves an additional key change:

Image Name Normalization. To implement the image match policy, we must compare the name of each image in a new pod request with the names of the images cached at each node. Recall that the latter is obtained from the container runtime at each node, and can be different from the name specified by the user in a pod request even for the same image. We therefore normalize the image names following canonical image naming rules before image matching.

3.3 Layer-match

Layer is a (sub-)group of dependencies insides a container image (see §2.1). We now describe the additional changes required for supporting layer-match.

Kubelets. Since the container-runtime only tracks cached images, we extend the kubelet to track cached layers as well. We add a *layer tracking* subroutine to the kubelet to collect and parse layer metadata from the node-local layer filesystem. The collected layer state (its digests and sizes) are periodically communicated to the API-server.

API-server. Similar to adding image-awareness, we extend the RPCs between the kubelet and API-server to communicate per-node layer state, along with adding this information in the global cluster state.

Scheduler. Extending the scheduler to support layer-match involves the following key change.

Image Resolution. We use this term to describe the process of mapping an image name to its corresponding layer digests. Dependency scheduling needs a container's layer information in order to assign it to a node and hence resolution must happen earlier so that the scheduler knows the mapping between the image of an incoming pod request and its layer digests.

We thereby implement the image resolver in the scheduler. Whenever a pod request with a new image comes in, the resolver obtains the image to layers mapping by querying the external image repository and caches them. This takes about 200*ms*. However, this is just a one-time penalty paid for new image requests, with the local image resolution from the cached mapping being the common-case occurrence. We implement this external image resolution outside of the scheduler's critical path to avoid any head-of-the-line blocking.

4 Evaluation

We evaluate dependency scheduling using a combination of measurement-driven simulations and experiments on a k8s cluster. The reason for simulation is two-fold: first, we found the real k8s cluster has overhead due to head-of-line blocking issues at the container runtime when pulling images and this overhead can mask the benefits of dependency scheduling. Second, using simulation allows us to do a comprehensive sweep of the parameter space.

Total no. of images	56,218
Total no. of layers	383,326
Sum of all image sizes	33.15TB
Sum of unique layer sizes	20.95TB
Average image pull latency	19.2s
Average layer pull latency	1.75s
Average no. of layers per image	11.95

Table 1: Summary of the trace collected for simulations

We compare the following scheduling policies: (i) *image-match*, (ii) *layer-match*, and (iii) an *agnostic* scheduling policy that places a task on one of the available nodes at random. We first describe our simulation methodology and experimental setup (§4.1) and then present the results (§4.2).

4.1 Trace-driven Simulation

Our simulation process comprises of four steps: *(i) Image Mirroring:* We select the latest versions of the 5K most frequently used images from DockerHub [4], forking them to the Amazon Elastic Container Registry (ECR). This saturates our repository limit on Amazon ECR (which was increased from the default of 1K images on request).

(ii) Latency Profiling: We deploy a Docker engine on an m4.xlarge dedicated Amazon EC2 instance, and pull the images from ECR. We instrument the Docker engine to log the size of each layer in the image, and the time taken to pull the layer (its *pull latency*)[1].

(iii) Extrapolating Latency Profile: We extrapolate the results from the latency profile of the above 5K images to create a latency profile for approximately 56K of the most frequently used DockerHub images. We use K-nearest neighbours for this extrapolation. Table 1 gives a high-level summary of this trace and the profiling results.

(iv) Cluster-level simulation: We wrote a simulator[2] that models a k8s cluster and implements the image- and layer-match policies, as well as the dependency *agnostic* policy.

Experiment Setup: We use the following parameters throughout the simulations: 200 nodes in the cluster, with at most 16 running containers and 32*GB* image cache size per node. Pod requests arrive with Poisson inter-arrival times; load is selected such that the cluster utilization is $\sim$ 80% for the *agnostic* policy. This setup falls into the regime (b) in §2.

The execution time for each task is uniformly sampled from 1-10 seconds. Our workload uses a realistic Zipf distribution (with an exponent value of 0.75) when picking the container image for each pod request, since it is the common access

[1] https://depsched.s3.amazonaws.com/layer_stats.csv

[2] https://github.com/depsched/sim

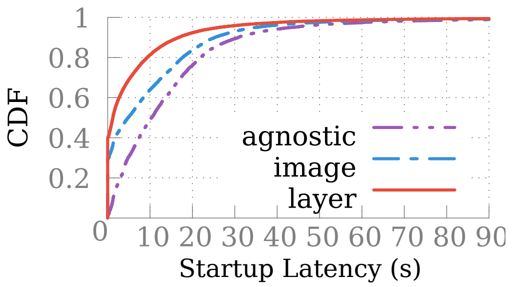

Figure 2: Latency cumulative distribution function (CDF) under different policies.

pattern observed in a wide range of scenarios [19, 20].

4.2 Key Results

We first present the results from our default setup.

Startup Latency: Figure 2 shows the CDF of the startup latency for our default simulation setting. These results confirm the design rationale discussed in §3: dependency scheduling (both image-match and layer-match) result in smaller startup latencies when compared to the *agnostic* policy, with layer-match generally performing better than image-match. On average, the image-match and layer-match policies result in 1.44x and 2.33x smaller startup latency than the agnostic policy respectively, with layer-match performing 1.6x better than image-match.

Resource Usage: In addition to reducing startup latency, dependency scheduling makes more efficient use of cluster resources, both compute and storage. *Compute:* The second column of Table 2 describes compute usage with the three policies for the same input load, measured as the sum of the total time each core is occupied divided by the product of the total number of cores in the cluster and the simulation duration. As shown, reduced provisioning time directly translates to smaller usage of compute resources in the cluster. Layer-match is most efficient, followed by image-match, with the agnostic policy being the least efficient.

Storage: In Table 2, we report the number of cached images per node and the amount of unused space in the per-node image store, computed as an average across all nodes and all scheduling rounds from the second half of the simulation (the latter to avoid startup effects). Dependency scheduling allows better packing of images in each node by co-locating images with larger numbers of shared dependencies. This results in more images stored per node as well as slightly higher unused (or free) space in the local image store.

We also used the simulator to conduct sensitivity analysis over key setup parameters including the number of nodes in the cluster, the per-node image cache size, and the image popularity distributions. We validated the trends of improvements under change of setups. Due to space limit, we include the results in the technical report [18].

4.3 System Evaluation

We evaluated dependency scheduling on a 60-node k8s cluster with the same setup as the simulation (except we are running containers on real nodes). We measured the container startup latency (1.83x and 2.34x speedup from image- and layer-match respectively at mean latency), the scheduling latency (8% and 13% higher than the agnostic policy, which has ≈0.6ms mean latency), and the container boot latency (≈1s mean latency). Note the scheduling latency is an order of magnitude lower than the boot and the overall startup latency; thereby it is not a major latency contributor.

Policy	**Cluster compute usage**	**Avg. no. of cached images per node**	**Avg. unused space in local store**
Agnostic	77.42%	34.68	5.11GB
Image-match	60.51%	40.24	5.64GB
Layer-match	39.12%	60.10	7.98GB

Table 2: Cluster compute usage and the per-node image cache utilization for the three policies.

Further, the difference in startup latency between the agnostic- and dependency-aware policies are much higher at higher-load (*i.e.,* having larger number of pod submitted to the k8s). This is caused by the queuing and consequential head-of-line blocking at the container runtime: there weren't enough resources (*e.g.,* network, CPU) to pull the image. We plan on investigating this fully in future works. More details on system evaluation can be found in our technical report [18].

5 Discussion and Related Work

Storage Optimization: Slacker [24] uses a proprietary NFS implementation to lazily pull the contents of the container image, and thus improve startup latency. Such strategies increase the complexity of the storage backend (e.g., to maintain many active client connections) and non-trivial infrastructure changes (e.g. modifications to the linux kernel and the use of a proprietary NFS server). Dependency scheduling is an orthogonal technique that is simpler to implement, and that can complement such storage optimization techniques to get even smaller startup latencies.

Dependency Trimming: Trimming dependencies is an orthogonal approach to reduce startup time that has been studied in the context of unikernels [30, 31]. These use an offline process for trimming dependencies and report lower startup latency than untrimmed unikernels and containers. Dependency scheduling is, again, orthogonal and complementary to dependency trimming, and offers benefits without requiring that users change their submitted container images.

Container Reuse: An edge provider can cache a popular pool of *running* containers such that they can immediately accommodate new function requests without incurring the time to provision and boot containers, similar to the way FaaS is implemented in the cloud [2, 11, 28, 36]. In edge environment, however, such hot-caching can be prohibitive due to the excessive use of limited resources such as memory.

Cluster Scheduling: There is a large body of work on cluster scheduling that focuses on reducing contention over shared resources, achieving better data locality and so on [21–23, 25, 29, 32, 33, 35]; these schemes do not optimize for startup times which is our focus.

6 Discussion Topics

Feedback and discussion points: Data locality has been a fundamental problem that repeatedly manifests itself in new contexts. A goal of this paper is to bring the discussion about data locality, especially dependency locality, to the fore in the context of edge computing. To this end, we are looking for feedback on (i) whether and how the real-world production systems and workloads on edge would fit into the application regimes we outline in §2.3; (ii) besides dependency locality, what are the other forms of data locality related issues that are pronounced for edge workloads; (iii) in the edge context, how valuable are the improvements that scheduling based optimization enables in terms of both performance and resource efficiency?

We are exploring other scheduling and system techniques to further improve the container startup latency and overhead. During the workshop, we would like feedback on these ongoing efforts and/or would welcome a discussion of future extensions and alternate approaches.

Open issues: There are open issues the paper does not address: (i) how dependency scheduling interacts with other scheduling policies such as load balancing and bin-packing. For example, dependency scheduling may lead to over-utilization of nodes having large set of popular dependencies. (ii) how much overhead does dependency scheduling introduce to k8s such as the API Server and KV store? (iii) container runtime bottleneck (mentioned in the end of §4). (iv) How much smarter caching would help improve the startup latency, *e.g.,* caching top-k popular layers per node (§2 includes some initial discussions)? We hope to learn additional issues from the workshop.

Controversial points and when does our work fail: We assumed short tasks are common for edge workloads (*i.e.,* much like serverless/Function-as-a-Service types of workloads). In such contexts, startup latency emerges as a potential bottleneck to low-latency processing. This characterization of edge workloads is up for debate! In fact, it would be great to discuss whether/how one might corroborate this assumption.

More generally, we anticipate debate surrounding our assumption that containers and Kubernetes are emerging as the de-facto technologies at the edge, and look forward to discussion on what compute abstractions and frameworks are best suited to edge workloads.

Our work will fail (or rather, will be less relevant to this venue) if the *trend* in edge workloads does not match our workload assumptions, *i.e.,* for long-running workloads, the overhead of pulling dependencies are substantially amortized by the task compute time and hence the benefits that dependency scheduling offers in terms of lower startup latency and improved resource efficiency are less valuable.

Acknowledgement

We thank our shepherd, Istemi Ekin Akkus, and the anonymous reviewers for their helpful suggestions and insights.

References

[1] Enable ImageLocalityPriority as a default scheduling policy. https://github.com/kubernetes/kubernetes/pull/68081.

[2] Understanding container reuse in aws lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2014.

[3] Connected vehicle platform. https://cloud.google.com/solutions/designing-connected-vehicle-platform/, 2019.

[4] Docker official images. https://github.com/docker-library/official-images, 2019.

[5] Ericsson: Connected vehicle cloud. https://www.ericsson.com/en/internet-of-things/automotive/connected-vehicle-cloud, 2019.

[6] The extensible building operating system. https://github.com/SoftwareDefinedBuildings/XBOS, 2019.

[7] irobot ready to unlock the next generation of smart homes using the aws cloud. https://aws.amazon.com/solutions/case-studies/irobot/, 2019.

[8] Open container initiative. https://www.opencontainers.org/, 2019.

[9] Yolo v3 (coco). https://supervise.ly/explore/models/yolo-v-3-coco-1849/overview/, 2019.

[10] Akraino edge stack. https://www.lfedge.org/projects/akraino/, 2020.

[11] Aws iot greengrass. https://aws.amazon.com/greengrass/, 2020.

[12] Kubeedge: A kubernetes native edge computing framework. https://kubeedge.io/, 2020.

[13] Kubernetes: Building large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large/, 2020.

[14] Kubernetes: Production-grade container orchestration. https://kubernetes.io/, 2020.

[15] LayerLocalityPriority proposal. https://shorturl.at/biuJ1., 2020.

[16] Lightweight kubernetes: The certified kubernetes distribution built for iot & edge computing. https://k3s.io/, 2020.

[17] Onf cord. https://www.opennetworking.org/cord/, 2020.

[18] Technical report. https://depsched.s3.amazonaws.com/report.pdf., 2020.

[19] Lada A Adamic and Bernardo A Huberman. Zipf's law and the internet. *Glottometrics*, 2002.

[20] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proc. IEEE INFOCOM*, 1999.

[21] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 2016.

[22] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert Nicholas Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proc. USENIX OSDI*, 2016.

[23] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proc. ACM SIGCOMM*, 2014.

[24] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *Proc. USENIX FAST*, 2016.

[25] Benjamin Hindman et al. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. USENIX NSDI*, 2011.

[26] Guoqiang Hu, Wee Peng Tay, and Yonggang Wen. Cloud robotics: architecture, challenges and applications. *IEEE network*, 2012.

[27] Michael Isard et al. Quincy: fair scheduling for distributed computing clusters. In *Proc. ACM SOSP*, 2009.

[28] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99. In *Proc. ACM SoCC*, 2017.

[29] Andrew Leung, Andrew Spyker, and Tim Bozarth. Titus: introducing containers to the netflix cloud. *Communications of the ACM*, 2018.

[30] Anil Madhavapeddy et al. Jitsu: Just-in-time summoning of unikernels. In *Proc. USENIX NSDI*, 2015.

[31] Filipe Manco et al. My vm is lighter (and safer) than your container. In *Proc. ACM SOSP*, 2017.

[32] Kay Ousterhout et al. The case for tiny tasks in compute clusters. In *Proc. USENIX HotOS*, 2013.

[33] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed, low latency scheduling. In *Proc. ACM SOSP*, 2013.

[34] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[35] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proc. ACM EuroSys*, 2015.

[36] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *Proc. USENIX ATC*, 2018.

[37] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. ACM EuroSys*, 2010.