Dependency Scheduling: Simple Techniques Towards Achieving Fast and Efficient Container Startup

Silvery Fu
UC Berkeley

Radhika Mittal
UIUC

Lei Zhang
Alibaba Group

Sylvia Ratnasamy
UC Berkeley

Abstract

Tasks demanding fast responsiveness are increasingly common in today's datacenters with containers becoming the canonical way of deploying such tasks. Unfortunately, container *startup* latencies remain high, limiting responsiveness. 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 management systems 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. Designing a dependency-aware scheduler involves a trade-off between complexity (due to the overheads associated with tracking and accounting for dependencies) versus effectiveness (measured as the improvement in startup latency).

We present two scheduler designs that represent two different points on this trade-off space: our *image-match* scheduler tracks dependencies at the granularity of entire container images while our *layer-match* scheduler tracks the finer-grained layers that constitute an image. We implement both designs within Kubernetes and evaluate them through extensive experiments and measurement-driven simulations. We show that dependency scheduling improves task startup latencies by 1.4-21x relative to current dependency-agnostic scheduling for typical scenarios; we further show that layer-match scheduling outperforms image-match by up to 1.65x, but does involve greater implementation complexity. Our implementation of image-match scheduling has been adopted into the mainline Kubernetes codebase. It has been enabled as a default scheduling policy in production clusters including Alibaba, while the adoption of layer-match is currently under discussion.

1 Introduction

Cloud applications are demanding ever faster responsiveness. We observe this trend for user-facing service tasks, where responsiveness can be measured by the time between when a task is submitted and when it is ready to start serving application-level requests. Spurred by the cloud serverless model from services such as AWS Lambda [2], Google Cloud Functions [7], and Azure Functions [12], developers are building applications comprised of short-lived service tasks. Such tasks are provisioned just-in-time upon request arrival to enable improved performance, cost efficiency, scaling, and convenience. A similar trend can be observed in interactive data analytics frameworks [36] and emerging IoT applications [3, 6, 8, 27].

At the same time, application containers are becoming the canonical way of deploying services in datacenters at large [13, 32, 38]. Unfortunately, container startup latency can significantly limit the efficiency of short tasks. This latency comes primarily from fetching container *dependencies* to 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. Google, for example, reports a median task startup latency of 25 seconds in their container clusters, with 80% of that time spent on package installations [38].

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 [28], to improve data locality [40], 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*.

Note that we are not advocating that *all* tasks be scheduled using dependency scheduling, nor that tasks be scheduled based on their dependencies *alone*. Instead, we envision that scheduling based on dependencies is one additional option available to operators and that operators can configure their scheduler to weight the relative importance of different scheduling policies based on the workload at hand.[1]

The fundamental question that arises when designing a dependency-aware scheduler is: at what granularity should we account for and track dependencies? Clearly, tracking dependencies at a finer granularity will enable greater savings in startup time but will also increase scheduling overheads and require more extensive modifications to existing container frameworks. We propose two scheduling solutions that allow us to capture different points in this tradeoff, yet are practical for implementation since they leverage system abstractions already available today.

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 an image match is found. However, because it does not consider the internal composition of a task's image, it cannot optimize launch times 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 explain our motivation for dependency scheduling (on both designs) in more details in §2.

We implement dependency scheduling 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 1.4-21x reduction in startup latency relative to dependency-agnostic schedulers while introducing as little as 0.3ms in scheduling overhead. Interestingly, as we show in §6, 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. In addition, we show that layer-match achieves startup latency up to 1.65x lower than image-match at the cost of modestly higher runtime and implementation overheads (§5).

[1]As we elaborate on in §3, this is exactly the model already supported in container frameworks such as Kubernetes which support multiple scheduling policies each associated with a configurable weighting factor.

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, while layer-match is currently under review [11], corroborating both the relevance and practical nature of dependency scheduling.

The remainder of this paper is organized as follows. We first motivate dependency scheduling (§2) and summarize the relevant background (§3). We then present the design (§4) and implementation (§5) of dependency scheduling. We evaluate our proposal using trace-based simulations (§6) and empirical tests (§7), and then discuss related work (§8) and conclude (§9).

2 The case for dependency scheduling

In this section, we examine the case for dependency scheduling and argue that it is well suited to emerging workloads/trends such as cloud-based voice services [1], cloud robotics [8, 27, 31], connected vehicles [3], smart buildings [6], serverless computing [30], and cloud IoT [17]. 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 runtime 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 §6). Our first observation is:

(1) Dependency scheduling is useful when S is a non-trivial portion of T (i.e., $S \propto R$). As prior work has noted, this regime is relevant as there has been a long-running trend towards shorter tasks. For example, the authors of [36] present a scheduler that targets task times of 100ms; their work is motivated by the observation that the Spark data analytics framework saw a 6000x reduction in task times over the preceding 6 years [36][35].

More recent work on prediction serving using a serverless platform [29] reports an average task time of 10 seconds of which approximately 8 seconds is spent launching the task. Likewise, Pywren, a new serverless platform reports launch times of 20-30 seconds which the authors report is "a main drawback" of their solution, accounting for over 10% of the total execution time [30].

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 back-end cloud-based service and client applications periodically query the same.

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

At the steady state[2], if:

(1) 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.
(2) 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 ($N \cdot C$). In this scenario, at the steady state, the dependency scheduling offers speed-up over agnostic scheduling. Intuitively, although popular layers cannot be cached on every node, they can be cached on some nodes in the cluster. Dependency scheduling is able to identify these nodes to speed-up the container 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 [9]). 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 [18, 37]); 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.
(3) 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.

We now compare image-match and layer-match policies falling under dependency scheduling. At the steady state, layer-match enables images that share layers to be grouped together. This reduces the amount of duplicate layers across nodes and hence allows more layers to be cached.

For instance, a plausible scenario for the CVP service example is that each car manufacturer provides the CVP service with an application stack developed for its own car models, and hence the container image for the same car model may share a portion of layers in their images (say 50MB of layers, estimated conservatively; total image size is then 300MB, taking into account the ML model). A 2018 study estimates that there are over 250 car models in the USA alone[15]. Back to the previous setting where there are 4,000 images, assuming they come uniformly from the car models then there are $4000/250 = 16$ images per car model. Layer-match is able to group and keep these 16 images in the same node and hence these layers takes only 50MB of additional space per car model. Image-match is not able to exploit the layer overlapping and can result in every node caching between 50MB to 250×50MB = 12.5GB additional spaces, accounting for 40% of node cache in the worst case.

As a summary, dependency scheduling treats the node caches as a collective cache and actively exploits dependency locality across all nodes. We will capture the subtlties of the comparison in the simulation.

3 Background

We use the following terms throughout the paper. A *node* refers to a machine hosting an OS that supports running containers. A *task* refers to a compute workload that runs on a node. We assume each task will be run in a container. We use the term *dependency* to describe the software packages and data files that a container requires to run.

3.1 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

[2]We loosely define the steady state as the moment where containers requests have been submitted to the cluster for long enough time and the node cache are populated by image layers.

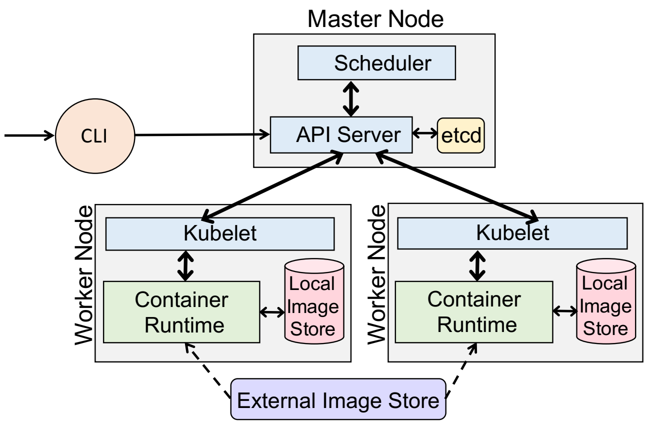

Figure 1. Overview of Kubernetes system architecture

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, *i.e.,* a container encapsulates an image's dependencies in the form of *layers*. When the container image is deployed on a node, its layers are union-mounted to form a complete filesystem for the running container. There is a specific order in which the node is required to install the layers. Two consecutive layers in this ordered list are termed as *parent* and *child* layers.

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. Therefore, the layered filesystem of a container offers an organized summary of the image's dependencies.

3.2 Container Orchestration using Kubernetes

While the concept of dependency scheduling is a general one, we focus on its application to Kubernetes [10] (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 those required image names, to the scheduler.

The k8s scheduler includes a set of *scoring functions* (or "scheduling priority functions" in k8s terminology), each evaluating the goodness of placing a task at a specified node based on different factors such as pod priorities, data locality, or node resource utilization levels. Scheduling a pod request is done in two steps. First, for each pod request, a predefined set of rules (called *predicates*) are applied to filter out nodes that do not meet the resource and affinity requirements for that pod. Each remaining node is then assigned a score by each scoring function and a node's overall score is a weighted average of the scores it receives from each function. Weights can be configured to balance different scheduling goals. The remaining nodes are then ranked according to their overall score and the pod is placed on the node with the highest score. By default, k8s uses equal weights over a set of scoring functions that consider factors such as pod priorities and resource utilization levels. Prior to our work, none of the k8s scoring functions considered pod dependencies.

We now briefly touch on the relevant per-node components in the k8s architecture. 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.

3.3 Task Startup Latency

We define a container's startup latency as the time taken from the submission of a task until its container starts running on a cluster node. Assuming the common-case scenario where the cluster is not overloaded (*i.e.,* there is at least one node that can accommodate the task), the startup phase of a task comprises of the following steps:
(i) Picking the appropriate node: The primary overhead here is simply the *scheduling computation time* needed to run the appropriate scoring functions, which is a small fraction of a task's startup latency, e.g. a few milliseconds for small-to-middle sized clusters (§7) or hundreds of milliseconds for larger ones.
(ii) Provisioning the node: This involves setting up the container's runtime environment and includes both a *resolution* and a *pulling* step. First, the node resolves an image name into its set of constituent layers - this step might involve contacting a remote registry to obtain the necessary mappings. Next, the node fetches and installs missing layers. The corresponding *pull latency* includes the time to download

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
```

the layers from a remote registry and (importantly) the time to decompress and register them, to set up the filesystem, and so on. Provisioning time is typically dominated by the pull latency. Consistent with prior work [24, 38], our measurements in §6 show that provisioning latency varies from milliseconds to hundreds of seconds.

(iii) Booting the container on the node: Finally, once provisioned, the container is started. This boot latency is typically small as well: we measure boot times of ~1s in §7.

Thus, among the above factors, the provisioning latency (dominated by the time taken to pull the dependencies) is the most variable and the largest contributor to startup latency. This is because container images have very different sizes (ranging from a few megabytes to gigabytes) and hence different download and installation times.

4 Dependency Scheduling

We discuss our overall approach (§4.1) and then describe the detailed design of our image-match (§4.2 and layer-match (§4.3) scheduling.

4.1 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 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 tradeoff between coarse vs. fine-grained dependencies and are practical for implementation.

Given a particular definition of dependencies, the following properties hold for dependency scheduling at at each scheduling iteration:

Property 1: *Dependency scheduling minimizes the provisioning time for an incoming request r under the assumption that the provisioning time is proportional to the size of r.*

Property 2: *Dependency scheduling minimizes the space used to store dependencies across the cluster.*

Property 3: *The overlap in dependencies between an incoming request and a cluster node is at least as large as when a coarser-grained definition of dependencies is used.*

Properties 1 and 2 hold by design of our algorithm, and lead to reduced provisioning time and more efficient storage space utilization within the cluster respectively. Property 3 holds under the assumption that a dependency can be further decomposed into one or more finer-grained dependencies. This assumption is true for the dependencies (images and layers) that we consider in this paper.

4.2 Image-match

For our image-match policy, *dep*() (in Algorithm 1) returns the container images in a request or a node, thus placing an incoming request at a node that has the maximum overlap (in bytes) between the images cached locally at the node and those in the request. For the common case of single image per request, this policy simply prefers a node that has the entire requested image already in cache.

When feasible, image-match entirely eliminates the pulling time for any matched images. Note that such a placement is feasible even when there are containers already running on the selected node (recall that images are read-only and copy-on-write). To perform image-match, the scheduler mainly needs two pieces of information: (i) what image is requested and (ii) what images exist on each worker node. The former is supplied by the request. The latter requires some modification to the container orchestrator. The scheduler cannot simply retain information about which images it has placed at different nodes since this information would be incorrect/stale as images are evicted from a worker node's cache. Hence, we instead rely on worker nodes to periodically propagate this information to the scheduler as follows.

Image State Propagation. Each worker node already tracks which images (their names and sizes) it caches. We refer to this information as a node's *image state* and extend worker nodes to propagate this state to the scheduler. In the specific context of k8s, worker nodes achieve this by storing their image state in the k8s key-value store (etcd) and the scheduler reads the relevant state from the key-value store and caches it locally. Upon a request arrival, the scheduler simply computes our image-match function using the image state information that it has cached locally.

Image Name Normalization. Since an image may be associated with multiple names (see §3.1), the scheduler may miss an image-match if the user-specified name for the requested image differs from what the worker nodes report. Therefore, to maximize the opportunities for image-match, the scheduler *normalizes* the names of requested images, using the same rules as those used at worker nodes (§3.1). Note that name normalization helps but does not guarantee matches when users apply different tags to the same image.

We describe our implementation of image-match in §5.

4.3 Layer-match

The image-match policy can lead to missed opportunities: unless there is an (available) node that has in its cache the exact image, with the same normalized name as the one specified in the request, image-match cannot offer any performance benefit over the dependency-agnostic policies. Indeed, it is common to see images that are only minor variants of each other; e.g., an image with a newly added layer, or with only a slight name change (e.g., a different tag).

In fact, a notable characteristic of container images is that they are subject to frequent and incremental updates that are made convenient by their layered structure. In our study, the 56K most popular container images on DockerHub have 675K layers in total but only 383K layers are unique. Further analysis of the top 1,000 images shows that each image has an average of 39.6 versions (each associated with a unique tag). The overlap in dependencies between these versions is typically high as well. For example, "microsoft/dotnet"(one of the most popular images) has 1,088 versions with a total of 905 layers across versions, of which only 426 are unique.

Our next design, therefore, defines dependencies at the granularity of container *layers* instead of the entire image. The scheduler places a new request at a node which has the maximum overlap (in bytes) between the layers cached locally and those needed by the request. At a high level, layer-match offers two important advantages over image-match: (1) It discovers a greater degree of overlap in dependencies. (2) The layer digest offers a content-addressable name for dependency identification: the layer's name/digest is a unique identifier of its content, allowing the required dependencies to be tracked and matched accurately. These benefits, however, come at the cost of more state that needs to be tracked and propagated and deeper modifications to existing container frameworks. We discuss our design below and present implementation details in §5.

Layer State Propagation. Worker nodes already track image state but they must be extended to track the *layer state* as well. Then, as in image-match, this per-node layer state must be communicated to the container scheduler, which in the context of k8s is achieved by having each node store its layer state in the k8s key-value store.

Image Resolution. The content addressable nature of layer names allows the scheduler to skip the name normalization process required in the image-aware policy. However, it now needs to deal with a different issue: since the request does not specify which layers constitute the requested image, the scheduler will not be able to perform layer-match. This issue can be addressed by making the scheduler query the image repository to retrieve and cache image-to-layer mappings.

4.4 Other optimizations and considerations

Eviction Policy The default policy adopted by container runtimes when its local image cache is full is to evict the least-recently-used (LRU) *image*. Ideally, to maximize the benefits of layer-match, this eviction process would also operate at the granularity of layers: i.e., evicting layers instead of images. However, unlike our other changes for scheduling, this would require changing the container runtime. We therefore use the conventional image-based eviction policy in our default design and implementation, although we also evaluate the benefits of layer-based evictions in §6.

Claiming CPU resources for container runtime A container, once scheduled on a particular node, is allocated a weighted fair share of CPU on that node. However, it starts using its CPU fair share only after all of its dependencies have been fetched and it starts running the task. The container runtime, that fetches, extracts, and installs the required dependencies, is allocated a very small amount of CPU by default [14]. This increases the request startup latency when CPU resources are overloaded. We implement an optimization where once the container runtime is ready to fetch a container's dependencies, it is allocated that container's CPU fair share. Once the dependencies have been fetched and installed, the CPU share claimed by the runtime is allocated to the container itself for running the task. This optimization has a minor impact on the results presented in §6 due to low baseline CPU usage. In our microbenchmarks with overloaded CPU, this optimization led to 4.2× speed-up in task startup latency (we omit detailed results for brevity).

Balancing Scheduling Policies. Cluster schedulers often consider balancing resource allocations and/or spreading containers (*e.g.,* microservice instances) across multiple nodes for higher availability and reliability. Both image- and layer-match policies can affect the decision made according to these criteria when, say, jobs with the same dependencies are potentially assigned to a smaller number of nodes, resulting in "hot spots" of overloaded nodes within the cluster. To resolve this, we rely on the scoring function of the schedulers as discussed in §3.2, where the weighting of different functions can be adjusted to navigate such trade-offs.

5 System Implementation

We implement dependency scheduling in k8s; our image-match policy is currently available in k8s (as of v1.12), while we are working on upstreaming our layer-match policy [11].

As discussed in §4, the primary change we make is to add *dependency awareness* to the k8s scheduler. At a high-level, this entails tracking dependencies at a per-node level, propagating this information to the scheduler (via the API server

and key-value store), and finally using this information in making scheduling decisions. This requires changes to the following k8s components (from Figure 1):
(i) Kubelets running at each worker node for tracking cached dependencies and communicating them to the master node.
(ii) API-server for incorporating dependency information in the global cluster state it maintains and in the RPCs it generates to communicate with other system components.
(iii) Scheduler for implementing dependency scheduling.

We describe the changes made to each of the above components in more detail below.

5.1 Extending k8s to support Image-Match

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 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 two key changes:
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, and can be different from the name specified by the user in a pod request even for the same image. This is because of the *image name normalization* done by the container runtime after the pod has been scheduled on it (as described in §3.1). To not miss the opportunity for an image-match in such cases, we normalize the image names before scheduling using the same rules as the container runtime. [3]
Adding Image-match Scheduling Criteria. We extend the node scoring function (see §3.2) to include image-match. When evaluating our image-match policy in the following sections, we set the weight for this criteria to the maximum allowed value so as to clearly isolate the performance gains due to dependency scheduling (we did the same for the layer-match policy discussed later); in practice, these weights can be configured differently depending on how important startup latency is for the workload in question.

5.2 Extending k8s to support Layer-Match

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 layer metadata directly from the local filesystem. The metadata for each layer includes its (a) *digest*, the global collision-resistant hash over the compressed layer blob (introduced in §3.1), (b) *diff-id*, the hash over the uncompressed layer blob computed locally by the node, and (c) *chain-id*, obtained by computing the hash of a layer's diff-id with its parent's chain-id. The topmost layer without a parent has a chain-id equal to its diff-id. The local filesystem indexes each cached layer by its chain-id, which can be used to track its diff-id and size. The diff-id, in turn, is used to track the corresponding layer digest using the *diff-id to digest mapping* maintained by the filesystem. Note that a diff-id can be associated with multiple chain-ids (a cached layer shared by multiple images can have multiple parents), but it is always mapped to its unique layer digest and size. The layer state (its digests and sizes), thus collected, are 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 also involves two key changes.
Image Resolution. We use this term to describe the process of mapping an image name to its corresponding layer digests. With the original k8s design, this resolution is executed by the container runtime at each worker node before pulling the required layers for a new image. With dependency scheduling, we need a pod'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, therefore, 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 respository 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 – when a new image request comes in, the scheduler can continue processing other cached image requests while the external resolution completes.
Adding Layer-match Scheduling Criteria. Similar to image-match, we extend the scoring function to include layer-match; i.e. the total size of layers cached in the node with digest values matching with the layers in the pod request.

[3] Image naming conventions are still in the process of standardization. We followed the conventions of Docker in our implementation.

6 Simulations driven by Real-world Traces

We evaluate dependency scheduling using a combination of measurement-driven simulations (this section) and experiments on a k8s cluster (§7).

We use simulations driven by an image trace obtained from DockerHub to test the limits of dependency scheduling at scale and its robustness over a wide range of parameter settings. We compare the following scheduling policies: (i) *image-match*, (ii) *layer-match*, and (iii) a dependency *agnostic* scheduling policy that places a task at a randomly selected available node. This agnostic policy reflects the absence of dependency-awareness in the default k8s scheduler prior to our changes and we use it as a baseline to compare our two dependency-aware scheduling policies.

We show that dependency scheduling, with either policy, consistently results in smaller startup latency and more efficient cluster usage than the dependency-agnostic baseline, with the finer-grained layer-match policy exhibiting greater benefits compared to the coarser-grained image-match.

We first describe our simulation methodology and default experimental setup (§6.1), then present our results across a variety of test scenarios (§6.2 - §6.5).

6.1 Simulation Process

Overview: Our simulation process comprises of four steps: (i) mirroring a subset of images from DockerHub to our private image registry on EC2, (ii) profiling the image pull latencies (iii) generating a realistic trace by extrapolating the collected latency profiles to a larger image set, and (iv) using the generated trace for cluster-level simulations. Simulations driven by the actual pull latencies as measured for real-world images on an EC2 cluster allow us to closely model reality while giving us the flexibility to experiment with more settings than would be possible using our real system implementation. We elaborate on the above steps below.

(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*).[4] The pull latency includes both the time taken to download a layer and to register it on the node, and we measure each. After all layers in an image have been pulled, we remove the image and ensure no cached layer exists locally, eliminating any layer reuse that can affect the latency readings. Figure 2 shows how the pull latency increases with increasing layer size.

[4]We will make the collected dataset publicly available.

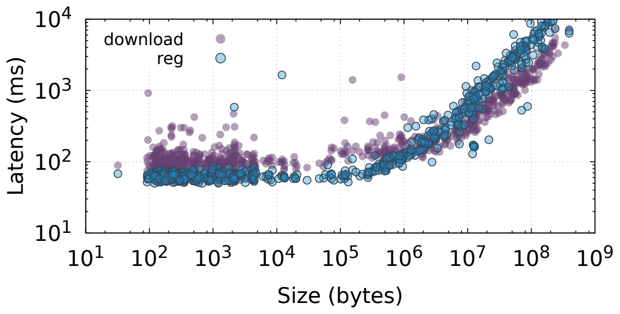

Figure 2. Layer download and registration latency vs layer size, measured with AWS EC2 m4.xlarge dedicated instance and Elastic Container Registry.

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

(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. We also validate our extrapolation strategy on a separate subset of the collected latency profiles and find that the extrapolated pull latencies are, on an average, within 15% of the actual latencies. Table 1 gives a high-level summary of this trace.

(iv) Cluster-level simulation: We wrote a simulator that models a k8s cluster and implements the image- and layer-match policies, as well as the dependency *agnostic* policy described before. To understand the impact of dependency scheduling, most of our experiments do not model the other scoring functions found in k8s' scheduler. This is equivalent to configuring the k8s scoring function with a very high weight for dependency scheduling. We relax this in §6.5. Among the three components of startup latency (§3.3), our simulation only captures the provisioning latency (more specifically, the pull latency) obtained from the latency profile trace. We do not model the *booting latency* and the *computation time*. However, we measure these using our system implementation in §7 and show that, in practice, these are far smaller than the pull latency and hence can be safely ignored.

Default Experiment Setup: Unless explicitly specified, 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, with the default load selected such that the cluster utilization is $\sim$ 80% for the *agnostic* policy. We model the common case of a single task (and,

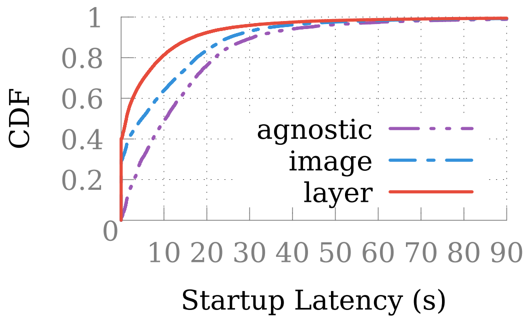

Figure 3. Container startup latency for the default experiment setting.

hence, container image) per pod. [5] Our default 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 pattern observed in a wide range of scenarios [19, 20]. The execution time for each task is uniformly sampled from 1-10 seconds. We vary these default settings to study how they affect our results in §6.4.

6.2 Key Results

We first present the results from our default setup.

Startup Latency: Figure 3 shows the CDF of the startup latency for our default simulation setting. These results confirm the design rationale discussed in §4: 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. The CDF can be divided into three regimes:

(i) The very low percentiles (< 30%ile) correspond to cases where the entire image is cached at a node, thus resulting in similar performance for layer-match and image-match, both having significant benefits over the agnostic policy.

(ii) The very high percentiles (> 90%ile), on the other hand, correspond to cases where no layer is cached, thus resulting in similarly high startup time for all three policies.

(iii) The range between 30-90%ile corresponds to cases where some layers are cached but not the entire image, in which (common) case layer-match has notable benefits over image-match, and both greatly outperform agnostic.

On average, the image-match and layer-match policies result in 1.44x and 2.33x smaller startup latency compared to agnostic policy respectively, with layer-match performing 1.6x better than image-match.

Cluster Usage: In addition to reducing startup latency, dependency scheduling makes more efficient use of cluster resources, both compute and storage.

[5]For cases with multiple containers per pod, we expect some affinity in the container images that are clubbed into a pod (analogous to layers in a particular image being requested together, but with abundant sharing across images). Our dependency scheduling design – both image-match and layer-match – will naturally exploit sharing in such cases as well.

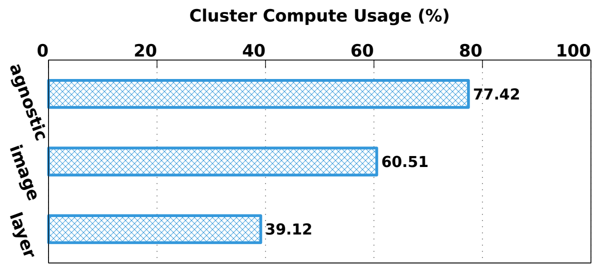

Figure 4. Cluster compute usage under the same load with different policies.

Policy	Avg. no. of cached images per node	Avg. unused space in local store
Agnostic	34.68	5.11GB
Image-match	40.24	5.64GB
Layer-match	60.10	7.98GB

Table 2. Average number of cached images and unused space in the local image store for the three policies.

Compute: Figure 4 shows 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 expected, 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.

Key Takeaway: Our results show that dependency scheduling results in smaller provisioning time and enables more efficient usage of cluster resources than dependency agnostic scheduling. They further reveal that a scheduling policy based on finer-grained dependencies (layers) offers greater benefits than using coarser-grained dependencies (images).

6.3 Impact of Layer Eviction

As discussed in §4, the layer-match scheduling policy can be further enhanced by using layers as the granularity for eviction (instead of the default strategy of evicting entire images based on an LRU policy). We now evaluate the benefits of layer-based eviction by comparing the startup latencies and the cluster resource usage when the layer-match scheduler is used with the default LRU image eviction policy and with

Metric	Eviction Policy		
	LRU Image	LRU Layer	LFU Layer
Startup Latency	6.5s	5.65s	5.42s
Avg. Compute Usage	47.08%	43.48%	42.58%
Avg. no. of cached images	60.10	72.92	71.08
Avg. unused local space	7.98GB	7.8GB	8.18GB

Table 3. Comparing results for layer-match scheduling across different eviction policies.

layer-based LRU and LFU eviction policies. Table 3 presents the results. We find that using layer-based LRU or LFU eviction results in 16% and 13% lower startup latencies respectively. Correspondingly, they result in 7% and 9% smaller compute usage, and 21% and 18% more images cached per node, although the observed average unused storage space remains similar across these different eviction policies.

6.4 Robustness Analysis

We now evaluate the robustness of our results to varying parameters.

Effect of changing cluster configuration: Cluster configuration includes the number of nodes in a cluster and the per-node cache size, which may vary depending on the physical limits of a cluster or its use.

Varying cluster size: Cluster size may range from a couple of nodes to a few thousands [9]. Fig. 5a shows the *speedup* due to using dependency scheduling, measured as the average startup latency for agnostic policy divided by the startup latency for image-match and for layer-match. As the cluster size increases, the number of cached layers – and hence the opportunities for sharing – increase, leading to greater performance benefits for dependency scheduling over the agnostic policy. Comparing the two dependency scheduling policies, the benefits of using layer-match over image-match first increase but then start decreasing, as the cluster size becomes large enough to cache more images.

Varying per-node image cache size: We capture the effect of changing the per-node cache size in Figure 5b. As before, increasing the cache size creates more opportunities for sharing, benefiting both layer and image-match policies. We, therefore, see a similar trend, where the benefits of using dependency scheduling over the agnostic policy increases, while the benefits of using layer-match, when compared to image-match, first increase and then start decreasing as the image cache size is increased beyond 48GB.

Effect of changing workload: We look at how our input workload parameters, i.e. request rate (or load) and image access pattern, affect our results.

Varying offered load: Figure 5c shows how the speedup in startup latency due to dependency scheduling changes as the offered load or the request arrival rate is increased. Dependency scheduling continues to perform better than the agnostic policy as the offered load is varied, with a sudden jump in the benefits seen at a high load of 150 requests/second, since at this point the cluster is almost fully (95%) utilized with the agnostic policy, while image-match and layer-match result in only 82% and 67% cluster compute usage. This shows that dependency scheduling can sustain a higher offered load than the agnostic policy. We again see that finer-grained layer-match gives greater benefits than image-match policy.

Varying image access pattern: In Figure 6, we change the image access pattern for the incoming requests from our default Zipf distribution, to (i) *uniform* distribution where images are assigned to the container uniformly at random and, (ii) the *popularity* distribution where the images are assigned proportionally by their pull count reported by Docker Store.

We see an interesting trend with uniform distribution where image-match policy shows very small benefits over the agnostic policy. This is because with uniform distribution, there is a larger variety of images among the containers running in the cluster, decreasing the probability of an image-match between an incoming request and one of the available nodes. However, since these images may have multiple shared layers, layer-match results in significantly smaller startup latencies than the agnostic policy.

By contrast, the popularity distribution is highly skewed, with a small number of very popular images which are often cached, resulting in significant benefits for dependency scheduling over agnostic policy: 19x and 21x lower startup latencies for image-match and layer-match respectively.

Key Takeaway: Our trends from §6.2 hold across a wide range of scenarios. Dependency scheduling performs better than the agnostic policy, with layer-match performance being generally better than image-match.

6.5 Supporting Multiple Objectives

In practice, cluster operators may want to achieve a combination of different objectives. We implement the scoring function mentioned in §4.3 in our simulations to see how dependency scheduling interacts with another policy that aims to balance the load within the cluster. This latter policy assigns higher score to nodes that have fewer running containers. For each scheduling round, we measure the ratio of the average number of containers running across all nodes over the maximum number of containers running in a single node. We average this ratio across all rounds to get a *balance ratio* that we use as a metric for the effectiveness of the load balancing scheme. We tune the relative weight of the the load-balancing policy with respect to the dependency scheduling (image- or layer-match) from 0:1 (no load balancing) to 1:0 (no dependency scheduling).

Figure 7 shows the result. We find that as the relative weight of load-balancing policy is increased, the speed-up in startup latency due to dependency scheduling decreases,

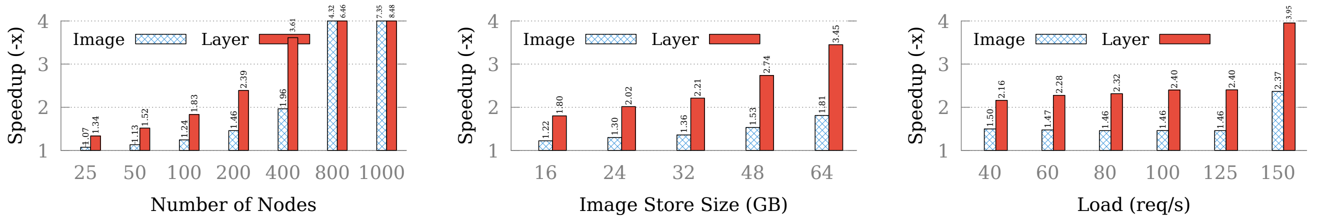

Figure 5. Speedup due to dependency scheduling as the number of nodes, per node cache size, or offered load is varied.

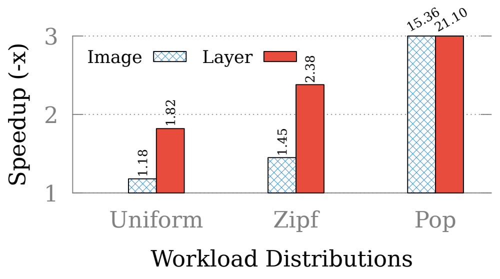

Figure 6. Speedup due to dependency scheduling as the image access pattern is varied.

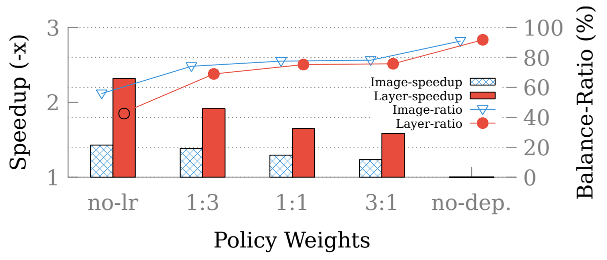

Figure 7. Using adaptive scoring to achieve good load balancing along with small startup latency with increasing relative weightage.

while the balance ratio increases. This shows that our dependency scheduling policies can co-exist well with other policies to produce the desired combination of objectives.

7 System Evaluation

We now evaluate our dependency scheduling implementation on k8s. In §7.1 we show how the startup latency trends observed in our simulations in §6 also hold for the real system. Then, in §7.2, we provide a breakdown of the startup latency to show the dominance of provisioning time.

7.1 Container Startup Latency

Experiment Setup: We set up a 60-node cluster on AWS EC2. Each node in our cluster is an c4.xlarge instance with 16 cores each. We configure the EBS (Elastic Block Service) volume attached to each node to have 32GB disk space reserved for the rootfs, which is used as the local image store. We use an input workload comprising of 5K images (saturating our ECR limit), with Poisson inter-arrival times and Zipf image access pattern as described before in §6.

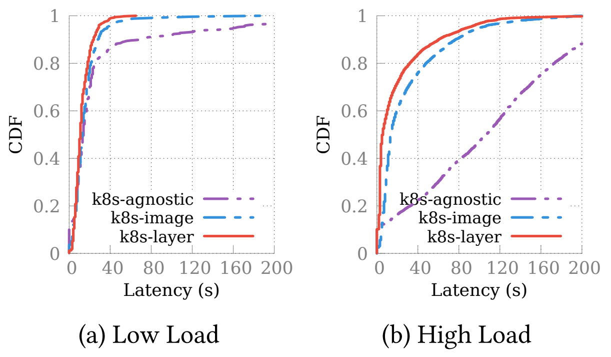

Figure 8. CDF of startup latency with 3 different policies at (a) 1 pod requests/second, and (b) 10 pod requests/second.

Results: In Figure 8(a) and (b) present the CDF of the observed startup latencies under low and high load respectively, for the three policies: dependency scheduling with image-match and layer-match, and k8s' default scoring function that is dependency-agnostic. We continue to see the same trends that were observed in our simulations before: the startup latency is smallest with layer-match and highest with the default (agnostic) policy.

Under low load, image- and layer-match have 1.83x and 2.34x smaller average startup latency than agnostic respectively. Under high load, the absolute values of the startup latency for all three policies are significantly higher than what we observed in the simulations before and for the low load case. This is because the system sees significant queuing in the node's container runtime engine – the runtime engine only provisions one container at a time resulting in a queue build up (an effect not captured in our simulations). With lower provisioning time, dependency scheduling sees much smaller per-node queuing, with layer-match performing better than image-match: image-match and layer-match result in 3.61x and 5.57x lower average startup latency than dependency agnostic respectively.

7.2 Other Startup Latency Components

We now show that provisioning latency is the most dominant of the three startup latency components described in §3.3, by

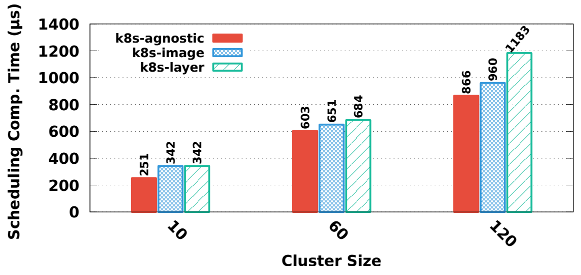

Figure 9. Scheduling computation latency comparisons.

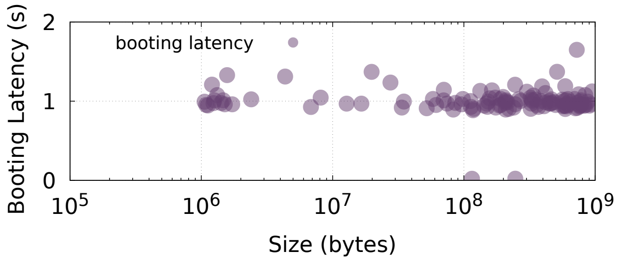

Figure 10. Booting latency vs image size.

measuring the *scheduling computation time* and the *booting latency* using our system implementation.

Scheduling Computation Time: We pre-warm each node with about a hundred images (corresponding to about 1000 layers) and measure the scheduling computation times for the three policies we consider for 100 consecutive pod requests. Fig. 9 shows this time for different cluster sizes. As expected, the computation time for dependency agnostic is the lowest, followed by image-match (8 - 36% higher than agnostic), and then layer-match (13 - 37% higher than agnostic and up to 23% higher than image-match).

Also, as expected, the computation times increase with increasing cluster sizes, as there is more per-node state information that needs to processed. [6] In all cases, the computation time incurred by dependency scheduling stays within 1.2ms, and is therefore negligible compared to the reductions in provisioning time that it achieves.

Booting Latency: We select a subset of 134 images of varying sizes and measure how long each of them take to boot up on a node (after being provisioned). Figure 10 shows our results. We find that irrespective of the image size, the booting latency largely remains fixed at around one second.

Key Takeaway: The provisioning latency (including the queuing in the container runtime) is indeed the dominant and the most variable component of startup latency.

[6]Notice that the computation time for dependency-agnostic scheduling also increases with increasing number of nodes. This is because we use the default scoring function implemented in k8s and do not simply pick an available node at random (as in the simulations).

8 Discussion and Related Work

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, 26, 32, 35, 36, 38]; these schemes do not optimize for startup times which is our focus. The latter is increasingly important in modern contexts where the infrastructure runs diverse application workloads composed of ever-shorter tasks [36].

Serverless Computing: Serverless [2, 7, 12, 25] is an emerging model for application deployments in the cloud that typically uses containers to run short tasks. Our work on reducing container startup latency is well-suited to this paradigm (§7).

Container Reuse: A cloud provider can cache a popular pool of *running* containers such that they can immediately accommodate new function requests [16, 30, 39] without incurring the time to provision and boot containers. Such container reuse differs from dependency scheduling along multiple dimensions: (i) container reuse only reduces start time for repeat requests from the same user (and hence cannot exploit the overlap in dependencies across users), (ii) the operator must proactively determine which containers are to be kept on hot standby, ready for reuse (vs. opportunistically exploiting dependencies that happen to be in cache), (iii) container reuse consumes additional runtime resources (not incurred by dependency scheduling), and (iv) container reuse cannot exploit the overlap in finer-grained dependencies (as layer-match does).[7] That said, those user requests that benefit from container reuse do enjoy low sub-second start times. Hence, we view dependency scheduling and container reuse as playing different and complementary roles.

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 [33, 34]. 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.

[7]Moreover, current usage of such container reuse is restricted to serverless architectures with specific language runtimes.

9 Conclusion

Programmers are most productive when they reuse existing libraries to build applications. Such reuse can lead to substantial dependency sharing between containers. Dependency scheduling explores and exploits such sharing. Our work shows that dependency scheduling is highly effective in cutting container startup latency. Our image-match policy has been adopted by mainline Kubernetes, while our layer-match policy is currently under review.

References

[1] 2019. Amazon Alexa. https://developer.amazon.com/en-US/alexa.

[2] 2019. AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[3] 2019. Designing a Connected Vehicle Platform on Cloud IoT Core. https://cloud.google.com/solutions/designing-connected-vehicle-platform/.

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

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

[6] 2019. The eXtensible Building Operating System. https://github.com/SoftwareDefinedBuildings/XBOS.

[7] 2019. Google Cloud Functions. https://cloud.google.com/functions/.

[8] 2019. iRobot Ready to Unlock the Next Generation of Smart Homes Using the AWS Cloud. https://aws.amazon.com/solutions/case-studies/irobot/.

[9] 2019. Kubernetes: Building Large Clusters. https://kubernetes.io/docs/admin/cluster-large/.

[10] 2019. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[11] 2019. Links redacted for anonymity and made available on the submission site.

[12] 2019. Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[13] 2019. Open Container Initiative. https://www.opencontainers.org/.

[14] 2019. Reserve Compute Resources for k8s System Daemons. https://kubernetes.io/docs/tasks/administer-cluster/reserve-compute-resources/.

[15] 2019. Total number of car models offered in the U.S. market 2000-2019. https://www.statista.com/statistics/200092/total-number-of-car-models-on-the-us-market-since-1990/.

[16] 2019. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/lambda/.

[17] 2019. Using AWS IoT Device Management in a Retail Scenario to Process Order Requests. https://aws.amazon.com/blogs/iot/iot-device-management-group-policy/.

[18] 2019. YOLO v3 (COCO). https://supervise.ly/explore/models/yolo-v-3-coco-1849/overview/.

[19] Lada A Adamic and Bernardo A Huberman. 2002. Zipf's law and the Internet. *Glottometrics* (2002).

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

[21] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* (2016).

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

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

[24] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers.. In *Proc. USENIX FAST*.

[25] Scott Hendrickson et al. 2016. Serverless Computation with open-Lambda. In *Proc. USENIX HotCloud*.

[26] Benjamin Hindman et al. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *Proc. USENIX NSDI*.

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

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

[29] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *Proc. IEEE IC2E*.

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

[31] Ben Kehoe, Sachin Patil, Pieter Abbeel, and Ken Goldberg. 2015. A survey of research on cloud robotics and automation. *IEEE Transactions on automation science and engineering* (2015).

[32] Andrew Leung, Andrew Spyker, and Tim Bozarth. 2018. Titus: introducing containers to the Netflix cloud. *Commun. ACM* (2018).

[33] Anil Madhavapeddy et al. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *Proc. USENIX NSDI*.

[34] Filipe Manco et al. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proc. ACM SOSP*.

[35] Kay Ousterhout et al. 2013. The Case for Tiny Tasks in Compute Clusters.. In *Proc. USENIX HotOS*.

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

[37] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

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

[39] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proc. USENIX ATC*.

[40] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proc. ACM EuroSys*.