Starlight: Fast Container Provisioning on the Edge and over the WAN

Jun Lin Chen, Daniyal Liaqat, Moshe Gabel,
and Eyal de Lara, *University of Toronto*

https://www.usenix.org/conference/nsdi22/presentation/chen-jun-lin

This paper is included in the Proceedings of the
19th USENIX Symposium on Networked Systems
Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the
19th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Starlight: Fast Container Provisioning on the Edge and over the WAN

Jun Lin Chen
University of Toronto

Daniyal Liaqat
University of Toronto

Moshe Gabel
University of Toronto

Eyal de Lara
University of Toronto

Abstract

Containers, originally designed for cloud environments, are increasingly popular for provisioning workers outside the cloud, for example in mobile and edge computing. These settings, however, bring new challenges: high latency links, limited bandwidth, and resource-constrained workers. The result is longer provisioning times when deploying new workers or updating existing ones, much of it due to network traffic.

Our analysis shows that current piecemeal approaches to reducing provisioning time are not always sufficient, and can even make things worse as round-trip times grow. Rather, we find that the very same layer-based structure that makes containers easy to develop and use also makes it more difficult to optimize deployment. Addressing this issue thus requires rethinking the container deployment pipeline as a whole.

Based on our findings, we present Starlight: an accelerator for container provisioning. Starlight decouples provisioning from development by redesigning the container deployment protocol, filesystem, and image storage format. Our evaluation using 21 popular containers shows that, on average, Starlight deploys and starts containers 3.0× faster than the current state-of-the-art implementation while incurring no runtime overhead and little (5%) storage overhead. Finally, it is backwards compatible with existing workers and uses standard container registries.

1 Introduction

Docker and other container engines are a popular approach for software provisioning due to their low overhead, standardization, and ease of use [3, 41, 53, 60]. They provide isolation and standardized packaging for application files, and are supported by a large suite of standard tools [16, 18, 21, 23, 24]. Unlike VMs, containers are lightweight and easy to update: even lightweight VMs [1, 38] require re-building and re-deploying the entire image. Container images, on the other hand, are built as a stack of layers; updating a component can be as simple as rebuilding its layer rather than the entire image [15]. Similarly, we can extend a container by adding layers to the top of its stack. Deploying is also straightforward: fetch compressed layers from a *registry* server such as Docker Hub, decompress them, mount using a layered filesystem [34], and start the container process. The stack-of-layers structure thus makes containers easy to develop and maintain, and fits well into modern development workflows [5].

Though originally designed to be used inside a cloud datacenter [20], containers are becoming increasingly popular in edge computing, mobile, and multi-cloud settings [11, 22, 25, 47, 56, 61].[1] Placing workers outside the cloud and closer to the user brings many advantages such as lower latency, bandwidth and power reduction, and privacy [52, 59]. Containers can be used to provision network functions at mobile base stations [13], Function-as-a-Service (FaaS) runtimes on local datacenters [43], local replicas in distributed stores [42], or components of distributed applications [61].

However, as systems grow larger and more complex, fast container provisioning is increasingly important. For example, Container-as-a-Service (CaaS) and FaaS providers must be able to provision workers quickly [3, 41, 60]. Another common case is rolling software updates, where we must update software across many thousands of workers [6, 50]. Edge computing brings its own set of challenges: high latency upstream links, bandwidth limits, resource-constrained local datacenters and workers, and user mobility. Pulling container images from a registry in the cloud to an edge worker takes a long time over wide-area links [25]. Another issue is user mobility, which causes frequent reconfigurations [57], making worker provisioning a common operation. Finally, limited resources in edge datacenters means that placing a local registry or cache at every edge can be expensive [25].

While there is work on improving container provisioning time, many are designed for the cloud [25, 60, 63], and are ill-suited for edge computing scenarios. For example, FaaSNet [60] uses a tree of workers to deploy containers in parallel, which is infeasible when latency is large and bandwidth

[1] The distinctions between these settings are not relevant for this work, hence we will refer to all of these using the umbrella term "edge computing".

is limited. Another popular approach is on-demand download [28,37,58], where we start containers early and download files on demand. These scale poorly with even moderate latency, even though many containerized applications do not necessarily require all mounted files immediately.

Our Contributions We identify three barriers to fast container provisioning. First, the layer-based structure that makes containers so convenient also prevents effectively applying common optimizations such as eliminating redundancy and downloading files on-demand. Second, the pull-based design of current approaches, where workers request what they need, becomes detrimental as latency grows. Finally, current approaches do not explicitly address the common scenario of software updates. We argue that faster provisioning requires a holistic approach to container deployment.

Motivated by these insights, we present **Starlight: an accelerator for provisioning container-based applications** that decouples the mechanism of container provisioning from container development. Starlight maintains the convenient stack-of-layers structure of container images, but uses a different representation when deploying them over the network. The development and operational pipelines remain unchanged: users can use existing containers, tools, and registries. In designing Starlight, we revisit every aspect of the container deployment mechanism:

- A redesigned **worker-cloud deployment protocol** sends all file metadata first, allowing containers to start before file contents are available. It uses a push-based approach to avoid costly round-trip requests: workers can specify what they already have in store, so we send only the files they need in the order they would be needed.
- On the worker side, we use a **new filesystem** to mount files as soon as metadata is available, allowing our **custom *snapshotter plugin*** to start containers quickly while downloading file contents in the background. When a container opens a file whose contents are pending, we block until the contents are available.
- Workers connect to a new **proxy** component in the cloud which implements the new protocol. The proxy optimizes the list and order of files on-demand, across multiple layers and containers This reduces duplication and makes updates faster. The proxy works transparently with existing infrastructure: compressed layers are stored in a standard registry, and legacy workers can connect to that registry as normal.
- A seekable **compressed layer format** allows the proxy to send individual compressed files to the worker without having to decompress stored layers first. This format has low overhead (average of 4.2%) and is backwards compatible with existing workers and registries, so there is no need to store container images in two formats.

We use 21 popular container images to evaluate Starlight across a range a range of network latencies, bandwidths, and scenarios. Our results show that Starlight substantially outperforms other approaches across all latencies, with 3.0× faster provisioning than a state-of-the-art baseline [21], and 1.9× faster on average than the next best approach [58]. Starlight also improves provisioning inside the cloud; for example it can deploy updates 35% faster than prior work [58]. In fact, Starlight containers often start faster than the time it would take to merely download an optimized container image. Finally, Starlight has little-to-no runtime overhead: its worker performance matches the standard state-of-the-art approach.

Starlight is currently available as an open source project at https://github.com/mc256/starlight.

2 Background

A *container* is a process that is isolated from the host system using techniques such as cgroups and namespaces [35]. A container is structured as a stack of *layers*, where each layer contains a part of the filesystem tree for the containerized application. Layers are mounted by the container process using a filesystem such as OverlayFS [34] that presents the containerized application with a *merged view*: files in upper layers replace those in lower layers, making it easy to update container contents using copy-on-write from lower layers. Most layers are read-only; writes go to a top read-write layer using copy-on-write as needed. A *container image* is the set of files and associated metadata that represent the container at rest (i.e., when it is not running). Concretely, container images are comprised of container configuration metadata and a sequence of *compressed layers*: compressed files that store the files in the layer and their associated metadata.

Containers are easy to develop, maintain, and deploy, due to their layer-based structure and standardized tooling. For example, developers can build new containerized applications by adding layers on top of an existing container image; packaging application updates is similarly straightforward. This also makes security updates for underlying components fast and automatic: applying an update simply requires updating the base layer. The repository of container images (the *registry* server) thus resembles a tree where individual images are split off from a common point.

Containers also make software provisioning easy using a three-phase process managed by a *container engine* on the worker such as containerd [21] or Docker [18]: (i) **pull** the requested container image from the registry and decompress its layers, (ii) **create** a container instance by preparing an initial snapshot of its filesystem state, and (iii) **start** the container instance, which involves mounting the snapshot filesystem and starting the container process using a standard *runtime*.

2.1 Edge Computing

Edge computing, defined broadly in this work, is the idea of placing computing resources outside the cloud, closer to the data or end users [52, 62]: near the network edge (e.g., local

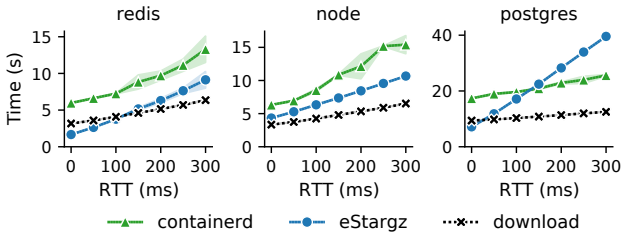


Figure 1: Mean container provisioning time across a range of latencies; shaded area show standard deviation across 5 runs.

datacenter, base station), user devices (e.g., mobile phone), or even in low Earth orbit [14]. We include in this definition settings such as mobile computing, content delivery networks (CDNs), Internet of Things (IoT), wide-area networks, and multi-cloud deployments.

Edge computing provides many benefits. For example, the short distance to users and data means faster and more consistent response times. And, since we no longer need to send all data to the cloud for processing, it improves privacy and reduces bandwidth usage. Other benefits include robustness to network failures and emission reduction [51, 52, 54].

Computing on edge workers has its drawbacks, however. First, they have much higher round-trip times (RTT) to the cloud. Recent work has found RTT ranging from 10ms to 400ms [10, 59] on terrestrial internet, and medians of 45–724ms on satellite-based internet [40]. Cross-datacenter latencies are also high, with one cloud provider reporting RTT between 2 to 400ms [45]. Bandwidth is also limited, with inter-datacenter bandwidths of 30–250Mbps [48]. Second, unlike cloud datacenters that offer virtually endless compute and storage, edge data centers are typically resource-constrained [54]. This encourages aggressive repurposing of workers, which makes fast provisioning even more important. For example, maintaining a pool of "hot" workers for elasticity is common in the cloud FaaS infrastructures, but is more expensive on the edge [43]. Lastly, edge and mobile applications are more affected by user mobility than cloud applications: as users move the nearest edge datacenter changes, which entails more frequent reconfiguration [57], i.e., provisioning.

3 Motivation

To explore the effect of latency on containers, we use `containerd` [21] to provision three popular containers over a 100Mbps connection with variable latency (see §5.1 for technical details). Figure 1 shows provisioning time, defined as the time it takes for the containerized application to download, decompress, start, and be ready. For comparison, we also show the download time for a file of equivalent size (dashed lines). We observe that `containerd` time increases substantially as RTT grows, and can even triple when RTT is 300ms. Moreover, in all cases provisioning time increases at a faster rate than would be expected simply due to extra network latency, which can be seen by comparing provisioning time to download time.

We also compare to eStargz [58], a recent approach that accelerates provisioning by starting the container before its layers have finished downloading and retrieving individual files on-demand. Prior work has found that many files are not used during container startup [28], indeed our three example containers access less than 1% of their files during startup (comprising 1–39% of data). Rather than wait until all files are available, eStargz starts the container quickly and download files on-demand [27, 28, 65]. It goes further by optimizing the order of files in each compressed layer such that the "hot" files needed early in container startup are placed first, thus avoiding redundant requests. Workers first fetch the hot part of each layer, start the container, and continue fetching the remaining files in the background or lazily on-demand.

As Figure 1 shows, when latency is small eStargz can accelerate provisioning. However, as RTT grows eStargz scales worse than the baseline and can even become slower than baseline `containerd`, as demonstrated for `postgres` with RTT of 150ms or above.

In the rest of this section, we analyze what makes optimizing provisioning difficult. We find that the root cause for slow provisioning time is the overall design of the provisioning pipeline: it is pull-based, designed around the stack-of-layers abstraction container images, and does not explicitly consider container updates. We show below that this design hinders optimization effort – both on the edge and in the cloud.

3.1 Pull-based Protocol

The protocol used to deploy containers to workers is pull-based: workers simply download the compressed layers they need from the registry using HTTP requests. This straightforward design avoids redundant pulls of layers that the worker already has, and works well inside datacenters. However, outside the cloud this can cause queueing delays, since registry implementations limits the number of concurrent connections per client to 2 or 3 [17]. Most containers have more layers [28], so the resulting cumulative delay adds up as RTT grows. Increasing the maximum number of concurrent connections could overwhelm the registry and may be impractical for resource-constrained workers.

On-demand downloading further exacerbates queuing by making even more HTTP requests to the registry. eStargz [58] uses a filesystem file access trace to determine the file order in compressed layers. In practice, however, the file access order of container workloads is not entirely deterministic due to multi-threading and runtime configuration. Container startup is thus slowed as multiple HTTP request due to out-of-order file accesses queue in the registry and delay one another.[2]

[2]Interestingly, excessive round-trips and queuing delays were also ob-

3.2 Layered-based Structure

Container images are structured as a stack of independent layers: each layer is stored separately, and contains its own metadata (e.g., list of files). While convenient for development, we argue that this makes optimizing provisioning more difficult: first, the information on container contents is distributed across multiple layers; second, because layers are the wrong granularity for provisioning protocols; and third, layer reuse does not capture updates well.

Distributed Metadata The first issue is that file metadata, including the list of files in the container contents, is not sent separately as part of the container image. Rather, each compressed layer includes its own list of files, and their metadata is intermingled with file contents. Yet, we cannot start a container early since list of files in a container is unknown until all layers are retrieved.

Consider again eStargz: since standard container images lack file metadata, eStargz stores a table of contents (ToC) at the end of every layer. Unfortunately, neither the size of compressed layers nor the exact beginning of the ToC is encoded in the image metadata. This in turn means at least two and perhaps three HTTP requests per layer: one to determine the size of its compressed image file, another to retrieve the layer's ToC from an estimated position before the end, and potentially a third if the ToC is larger than expected.

Fixing this is not trivial, since container images are standardized; careless changes would make development harder. For example, adding a table of contents to container image metadata requires changing the standard and updating a huge number of existing tools used by developers [18, 21, 23, 46].

Layer vs. File Granularity Second, and perhaps counter-intuitively, the layer-based structure makes deployment slower due to cross-layer (and cross-container) redundancy. Containers evolve one layer at a time by extending other images with new layers. To update a file, we first copy it from the original read-only layer to the top read-write layer. Changing file metadata (e.g., ownership) also requires copying since layers cannot refer to each other. In both cases the original file remains in the previous layer, with no indication that this has happened. This cross-layer data duplication cannot be captured explicitly since file metadata is stored in the layers, and cannot be exploited by compression since layers are compressed independently.

Table 1 illustrates the cost of such redundancy for our sample containers by comparing the required download size using the baseline layer-based approach, to the size of an optimized "delta" update that only includes changed files and removes duplicates across layers.[3] The inflation in update sizes ranges from 1.23× (redis) to a whopping 10.54× (node). Indeed, a recent analysis of Docker Hub [64] found that 90% of layers are only referenced by a single image, but over 99.4% of files had duplicates. Exploiting this cross-layer duplication during provisioning is difficult since file metadata is distributed across multiple layer.

Container	From → To	Baseline	Delta
redis	6.2.1 → 6.2.2	9.6	7.8
node (alpine)	16-3.11 → 16-3.12	39.0	3.7
postgres	13.1 → 13.2	109.5	24.9

Table 1: Package size (MB) of standard and optimized update.

While there has been prior work that proposes deduplicating the registry [55, 63], this does not reduce provisioning time since (by design) the downloaded container images and provisioning protocol remain the same. Rather, such work focus on saving registry space.

Limited Layer-reuse Ideally, an updated container image would share common layers with its previous version, so deploying updates requires only fetching and decompressing the new layers. Unfortunately, even a minor change to a single layer low in the stack causes cascading effect where all layers above it must be updated, even though their contents are mostly identical [15]. On such example is updating a worker from `postgres:13.1` to `postgres:13.2`. These two container images share no layers since an update to the `debian:buster-20210208-slim` image forced an update to all downstream layers. Provisioning this update requires downloading and decompressing the entire image, even though the total size of changed files is much smaller. Our analysis of 21 popular containers (Table 2) suggests that layer reuse only captures 3% of duplication, on average.

3.3 No Explicit Update Support

Provisioning a worker is not a rare operation. Rather, over the lifetime of a worker, we will deploy containers many times and for different reasons: initial provisioning, software updates, security patches, and so on. This even more common on edge workers due to user mobility and limited resources at edge datacenters (§2.1). This not only results in frequent provisioning, but also means that worker contents is highly diverse: as workers get updated and repurposed, the version of the container image available in local storage varies from worker to worker. As discussed above, such updates are an opportunity for optimization since many of the files have not, in fact changed (§3.2).

However, the current design of the provisioning pipeline does not allow users to express update operations explicitly.

served in mobile web browsers that use HTTP/2 [36]. The underlying causes, however, are quite different (handshaking and packet losses, respectively). Determining whether the mitigation approaches in QUIC are applicable for container provisioning (or vice versa) is beyond the scope of this work.

[3] Flattening container images down to a single merged layer this way would mitigate many of the issues we discuss. However, would also eliminate the advantages of containers in the first place (§2), and would require an optimized image for every potential update path [47, 55].

Under the current approach, updates are treated as any other deployment: the worker simply pulls the needed layers from the registry. Depending on the update, this may or may not result in faster provisioning. While in theory we could prepare optimized provisioning packages in advance, the diversity in worker contents makes this approach impractical. A better approach is to compile the provisioning package dynamically, on-demand, by taking into account what is already available at the worker when selecting which files to include. Doing so, however, requires a capacity to express worker updates, which the current provisioning protocol does not support.

4 Starlight

Starlight is designed to accelerate container provisioning by considering the deployment pipeline holistically. In designing Starlight, we set out to achieve several goals. First, accelerate deployment on both low and high-latency links, and scale gracefully as latency grows. Second, preserve the advantages of containers for application developers. For the same reason, Starlight should be easy to adopt incrementally, without causing interference or requiring abrupt changes to working systems – Starlight should be backwards-compatible with non-Starlight workers, work with existing infrastructure, and have low overhead. Finally, Starlight should better support the common scenario of container updates.

4.1 Design Considerations

Starlight's design is driven by four key principles, informed by our analysis of container provisioning (§3): (1) start containers early, (2) send workers only what they need, (3) use a push-based design to avoid costly round-trips, and (4) prioritize worker performance over cloud effort. These lead to the following design decisions:

- The provisioning protocol should not resemble the stack-of-layer structure of container images. Instead, it should be pushed-based, and operate at file rather than layer granularity. The list and order of files should be jointly optimized across multiple layers and containers.
- The provisioning protocol should cleanly separate file metadata from contents, and send the metadata first. This allows Starlight to start containers early by mounting a "mock" filesystem while downloading contents in the background.
- The provisioning protocol should let workers explicitly request updates and specify what is available to them locally.
- Avoid changing registry by placing a *proxy* located near it, which lets us to change provisioning protocol without affecting existing workers.
- The proxy should create provisioning packages on-demand based on what the worker already has available. This makes updating workers more efficient, avoids inflating the registry with packages for every conceivable update, and places the computational burden on the cloud where it is cheaper (§2.1). Supporting this requires storing a table of contents and file metadata for every container.
- Storing compressed layers using a seekable backwards-compatible format allows both Starlight and legacy workers to use the same compressed layer files and standard registries, and avoids inflating the registry size.
- Use standard container images to support the large ecosystem of existing tools for building, storing, and serving containers [16, 18, 21, 23, 24, 30, 46].

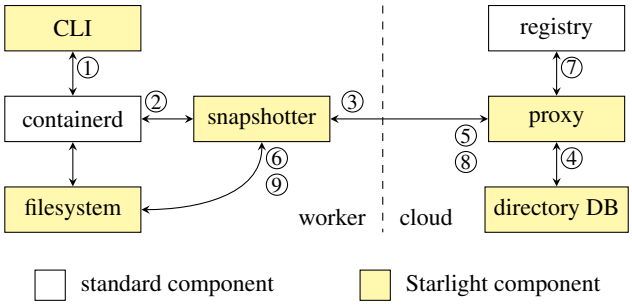


Figure 2: Starlight architecture.

4.2 Overview

Figure 2 shows Starlight's architecture, which is comprised of a *proxy* and a *directory database* (§4.4) in the cloud next a standard registry; and a *snapshotter* plugin (§4.5) on the worker. The proxy and the snapshotter plugin communicate using the Delta Bundle Protocol (§4.3). The snapshotter plugin manages the lifecycle of the *filesystem* (§4.6) for the container instance. We also include a command line tool.

We first describe Starlight's operation at a high level, and how it maps to the three-step PULL-CREATE-START process.

Once the user issues a worker PULL command to deploy a container ①, the command is received by the standard `containerd` daemon. `containerd` then forwards the command to the Starlight *snapshotter* daemon ②, and waits for confirmation that the requested images have been found. The Starlight snapshotter opens an HTTPS connection to the Starlight *proxy* and sends the list of requested containers as well as the list of relevant containers that already exist on the worker ③. The proxy queries the *directory database* ④ for the list of files in the various layers of the requested container image, as well in the image already available on the worker. The proxy will then begin computing the *delta bundle* that includes the set of distinct compressed file contents that the worker does not already have, specifically organized to speed up deployment; In the background, the proxy issues a series of HTTPS requests to the registry ⑦ to retrieve the compressed contents of files needed for delta bundle. Once the contents of the delta bundle has been computed, the proxy creates a *Starlight manifest* (SLM) – the list of file metadata,

container manifests, and other required metadata – and sends it to the snapshotter ⑤, which notifies `containerd` that the PULL phase has finished successfully.

We can then use the SLM execute the CREATE phase: configuring the container and the Starlight filesystem ⑥.

Starlight then proceeds to the START phase: it mounts then launch container instances. Note that even though the container instances have launched, at this point the worker does not have the contents of many files (or perhaps all, for new deployments). Such files are mounted based on metadata only; when opened, the Starlight filesystem will block until file contents have arrived from the proxy. The proxy streams file contents back to the snapshotter ⑧, which in turn updates the filesystem to unblocks the access of the file ⑨. To optimize provisioning times, the proxy sends files in the order they will (likely) be needed; this order is determined when building the delta bundle, and relies on preprocessed information stored in the directory database.

4.3 Delta Bundle Protocol

Starlight uses a novel Delta Bundle Protocol to send container images to the worker. This single-request HTTP-based protocol is designed to reduce unnecessary transfers, avoid round-trips, and prioritise information needed to start the container. Much of Starlight's design is informed by the need to support the Delta Bundle Protocol.

A provisioning request includes the name and version of the requested the container image as part of the request URL, and optionally the old version in the worker's local storage. The response consists of two parts: a header and a body.

The Header The header contains of all the information needed to start container instances of the requested images. It it comprised of (1) a Starlight Manifest (SLM), (2) a table of all layers digests from both the existing and requested container images, and (3) other data required by the implementation such as protocol version and authentication..

An SLM includes a standard Open Container Initiative (OCI) container image manifest file [24, 39], an OCI configuration file for the instance, a list of indices into the table of layer digests, and the filesystem table of content (ToC).

The ToC presents a merged (flattened) view of the requested container's filesystem (§2), providing sufficient information for the worker to mount the container's filesystem using StarlightFS without waiting for the response body. Each entry in the ToC includes the file name and path, type (e.g. regular file, link, or directory), attributes (e.g. ownership and timestamps), and an SHA256 hash of the file content. Additionally, every entry includes an index to the shared layer digests table in the delta bundle header, which enables reusing file contents on the worker's local storage. Finally, each entry also has an offset field which points to the file's *payload* – compressed file content – in the body of the delta bundle.

Using the SLM The name, metadata, offset, and index into the digest list allow workers to reconstruct the requested container's filesystem. For new or updated files – those that the worker does not already have in its local storage – the offset points to the payload. This allows multiple file entries to reuse the same payload in the body of delta bundle, reducing the transfer volume. Alternatively, if a file's metadata has changed (e.g., ownership), the payload already exists on the worker. The ToC entry thus contains the new metadata, an empty payload offset, and an index pointing to the original layer in the list of digests.

The Delta Bundle Body The body is a sequence of payloads (compressed file contents) for new or updated files, sent in the order which they are likely to be accessed. Since the header allows multiple file to reference the same payload – all payloads in the body are unique.

4.4 Proxy and Directory Database

Despite the name, the Starlight proxy is not merely a proxy server or a simple bridge. It is in charge of optimizing and building the delta bundle sent to the workers, as well as collecting and analyzing filesystem traces used in this optimization.

The Directory Database The directory database stores the table of contents and file metadata for each container image in the registry, as well as additional information used by the proxy to compute and optimize the delta bundle.

Whenever a new container image is uploaded to the registry (triggered manually or by hooks), the proxy captures file metadata from all layers, generates the ToC for the merged view of the image, and then save the ToC, container manifest, and image configuration file to the directory database.

The ToC in the directory database is the same as the ToC included in the SLM with additional fields that fascilitate building the delta bundle body. First, it records the source compressed layer file, payload offset, and size for each file to help retrieve it from the registry. Second, it includes two extra columns, rank sum and hit count, used when sorting payloads; we discuss these below.

Trace Collection To sort payloads in the order that the worker is likely to access, Starlight collects filesystem traces from the worker to analyse the file usage. Trace collection is identical to running a container until it reports it is ready. When initiated by the user, the worker starts the container image locally using a special mode of the Starlight filesystem (§4.6) that collects file accesses. The worker then uploads the trace to the proxy, which ranks all files in the trace according to their access order. Finally, for each file in a container image, the proxy increases its hit count by one and adds its rank to the rank sum column. The average rank of a file can be computed from its rank sum and count.

Since file access can be non-deterministic, our design supports multiple collection runs. Collecting one trace usually

takes up to 2 minutes per run, depending on the container. By default, we collect 10 traces for each container.

Note that while prior work [37, 58] stores file order information per layer inside the compressed layers, Starlight associates this information with a container image and stores it outside the registry. This provides several benefits. First, it is possible to update file usage without rebuilding the compressed layers. Second, it allows for the likelihood that a file in a layer used by different containers to be accessed differently during startup. Third, new container image can reuse the traces from a previous version – solving the cold start problem. Finally, it allows for future development such as adjusting payload orders online based on provisioning feedback.

Provisioning Process A provisioning request from a worker contains the names and tags of two images: the image requested for deployment, denoted by R, and the old version of the image in its local storage (assuming there is one) denoted by A. To build the delta bundle, the proxy first retrieves metadata from the directory database for both container images R and A. It then issues a series of asynchronous requests to the registry to retrieve the compressed layers for R. These will be used to construct the body of the delta bundle.[4] It then proceeds to prepare an optimized delta bundle header and send it to the worker. Once all requested layers to arrive from the registry, the proxy send the delta bundle body: for each payload in the delta bundle body as determined by the header, we copy compressed file content directly from the compressed layer and send them to the worker.

Optimizer The optimizer is responsible for selecting which compressed file content (payload) should be included in the body of the delta bundle and in which order, and then building the delta bundle header. Crucially, the optimizer does not require retrieving the compressed layers; the directory database contains all necessary information to build the delta bundle header. The optimization proceeds in several phases:

- **Merge:** load the merged (flattened) ToC for R and A from the directory database, denote them T_R and T_A.
- **Difference:** Compute the set difference $T' = T_R \setminus T_A$: for every file f in T_R, we look for a corresponding entry f' in T_A with the same hash and name. If we find one, we update the source layer index for f in T_R to the corresponding one in the old the entry in T_A update its source layer index to the corresponding layer of f'. This step takes $O(|T_R| + |T_A|)$ time and $O(1)$ space.
- **Consolidate:** Consolidate files in T' with the same payload. Assuming the chance of hash collision is low, this step takes $O(|T_R|)$ time and $O(|T_R|)$ space.
- **Select:** Remove from T' files already available on the worker (whose source layer is in T_A).
- **Sort:** Sort the payloads in order of increasing average rank, $O(|T'| \log |T'|)$. If different files point to the same payload due to previous steps, use the lowest rank.

Compressed Layer Format The current format used to store compressed layers is the tar gzip format: a sequence of concatenated files with interleaved headers for metadata (e.g., timestamps, ownership), compressed as one data stream. This format is non-seekable: extracting a specific file requires decompressing the entire compressed layer until we reach the file, which takes time.

eStargz [58] uses an alternative seekable compressed layer format that compresses files individually (or 4KB chunks of larger a file) and appends an index at the end of the compressed layer into the offsets of compressed files and chunks. To maintain backwards compatibility, each file includes tar headers and footers, so the tarball data stream remains unchanged. The result is an increase in the size of compressed layers due to the index at the end and the additional tar headers and footers. Furthermore, compression is less effective since file are compressed separately.

Our proposed format follows similar ideas, with three differences. First, we do not include an index at the end of the compressed layer, and instead use the directory database to store the table of contents. This not only reduces the overhead of our compressed layer format, but allows the proxy to build the delta bundle while fetching compressed layers in the background. Second, we do not need to split files into 4kb chunks since we retrieve files wholly, which simplifies our provisioning protocol and reduces the size of the ToC. Finally, since we do not need the metadata in tar headers and footers during provisioning, we do not include them as part of the compressed stream of file contents, which further reduces payload size. The overhead of Starlight's format is only 4.4% for containers in Table 2 comparable to eStargz (4.7%).

4.5 Snapshotter Plugin

The `containerd` snapshotter daemon manages the life cycle of a container filesystem: from downloading images to keeping track of changes in the container's mounted file system. We take advantage of the snapshotter plugin-based design [7] to write a snapshotter plugin to support Starlight provisioning. Figure 3 shows an overview of the Starlight snapshotter plugin, which includes two components: the *downloader* and *metadata manager*. The snapshotter also maintains the instances of the user space component of StarlightFS – one for every mounted container instance.

Delta Bundle Downloader The downloader is responsible for downloading the delta bundle from the proxy and decompressing the payloads to designated locations (if an image has been completely downloaded, it is not started). Once the downloader receives the delta bundle header, it saves the

[4] Our current implementation retrieves entire compressed layers. This does not substantially affects provisioning time since the registry and the proxy are located in the same cloud datacenter. Nevertheless, we stress that Starlight's directory database and the seekable image format support retrieving only the contents of compressed files, by issuing HTTP range requests to the registry when building the delta bundle body. We are planning to implement this optimization in the immediate future.

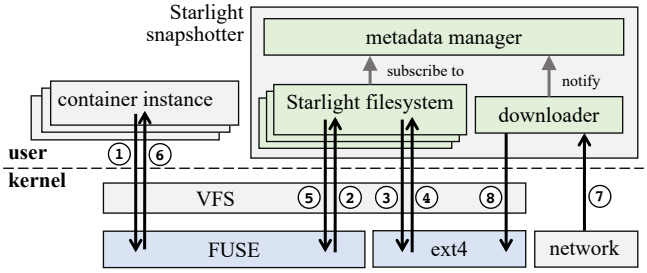


Figure 3: The flow of filesystem requests in Starlight.

SLM to the local storage and creates a metadata manager using the SLM. At this point, we have enough information to mount and start the container, so the snapshotter notifies the `containerd` daemon that the `PULL` phase has finished. In the background the downloader keeps receiving the payloads: it decompresses the payloads to its designated location according to the source layer information in the received SLM and the share layer digests in the delta bundle. Once the decompression of the payload has finished, it notifies the metadata manager. When a payload belongs to multiple files, the downloader creates hard links to avoid writing to the underlying filesystem multiple times.

Metadata Manager Multiple container instances can start from the same container image. The metadata manager therefore acts as a centralize place for managing file's availability and its metadata. It maintains file metadata of all the files in a container image and manages files' actual location in the host file system. Most importantly, it manages the availability of file contents, and notifies StarlightFS once a file payload has decompressed. Once the worker has downloaded the entire image, we store its SLM locally so that future container instances launch from the local storage. When removing an old container image, the metadata manager removes any hard link references (if any) and copies the file to a new location if it is used by a newer version of this image.

4.6 The Starlight Filesystem (StarlightFS)

The customized filesystem serves two goals. First, we need to start containers early using only their SLM. Second, we want to reuse file contents across layers and images. As OverlayFS and other filesystems do not support both of these features, we use FUSE [33] to implement StarlightFS.

Structure StarlightFS relies on the underlying host filesystem (e.g., ext4), similar to a typical OverlayFS and FUSE-OverlayFS. Like OverlayFS, StarlightFS provides a container with a merged view that combines multiple directories in the underlying filesystem that represent multiple read-only layers and a single read-write layer.

Starlight maintains a filesystem tree in memory, created from the merged view ToC in SLM. Each file (or directory) node keeps track of the actual location of the file contents – whether it is in the read-only layer, in the read-write layer, or pending payload. As with the ToC, some nodes might reference read-only layers from the previous version of the container image (§4.3). Nodes of pending files will be notified by the metadata manager when the payload is available (in our implementation, by subscribing to a Go signal channel in the corresponding file entry of the metadata manager). The user space portion of StarlightFS is located in the snapshotter process to allow such low-overhead communication.

Note that StarlightFS does not maintain any file system state on its own, nor does it have any on-disk structures. Metadata for files in read-only layers is stored in the ToC. State for files in the read-write layer (i.e., mutable state) is stored in the underlying host filesystem, with changes forwarded to it immediately. For example, if a file is deleted by the container, we write a `whiteout` entry to the read-write layer, similarly to OverlayFS [34]. In case of a crash, error or remount, the tree and all other state are rebuilt using the saved SLM and underlying filesystem.

Operation When starting a container instance, the snapshotter creates a filesystem instance which builds a filesystem tree from the metadata manager's ToC for this container image.

Figure 3 shows the flow of operations in StarlightFS. When a container instance performs a file operation, it is forwarded to StarlightFS via FUSE ①②. In the best scenario, the content of the file is already in the local filesystem (e.g., ext4in Figure 3). Starlight uses the file path provided by the file node to opens the underlying file ③④ and then return the file handle back to the container instance ⑤⑥. In case the file contents are still pending, but the operation only involves reading metadata (e.g. `GETATTR`), StarlightFS returns the metadata immediately using the information in the file node.

When an operation on a pending file involves setting metadata (e.g. `SETATTR`) or accessing file content (e.g. `OPEN`, `FSYNC`), StarlightFS blocks the operation until the file is ready by subscribing to a Go signal channel associated with the file's ToC entry in the medtadata manager. Once the downloader has extracted the file payload ⑦⑧, it notifies the corresponding entry in the metadata manager, which closes the channel associated with the file's ToC entry. This releases any filesystem tree nodes that are waiting for the payload, while newly created instances will not be able to subscribe to a closed message channel. StarlightFS can then load the file from the local storage and update the file metadata if necessary ③④, then return the file to the container instance ⑤⑥. If this requires changing the file metadata or content, this file will be copied from the read-only layer to the read-write layer. All subsequent requests will be forwarded to the read-write layer.

5 Evaluation

We use 21 popular container images to evaluate Starlight's performance in both controlled and real-world networks. Our

main metric is *provisioning time*, defined as the time from the initial command to deploy a container on a worker, to the time the containerized application reports it is ready (as with HelloBench [28], this is determined by monitoring the application's `stdout`). To show the benefit of Starlight, we define two types of provisioning: a *fresh* deployment means the container worker does not have any prior images in its local storage, while an *update* means deploying the next available version of a container to a worker that already has the previous version deployed.

5.1 Experimental Setup

We use AWS EC2 to run our experiments. Container workers use `m5a.large` instances (AMD EPYC 7000 at 2.5GHz) with 2 cores (vCPUs) and 8GB RAM. The registry server runs Docker Registry 2.0[5] v2.7.1 on a `c5.xlarge` instance (Intel Xeon at 3.4GHz) with 4 vCPUs and 8GB RAM. The Starlight proxy and the metadata database run on a second `c5.xlarge` instance. All the machines run Ubuntu 20.04.3 LTS. We use Linux's Traffic Control tool [2, 29] to control round-trip time and bandwidth between the worker and the other machines. Bandwidth is limited to 100Mbps unless otherwise specified. For cloud experiments, bandwidth and latency are not limited (RTT is ∼0.15ms). Each experiment is run 5 times.

Benchmark Approaches We compare Starlight to two state-of-the-art approaches: the `containerd` baseline [21] v1.5.0 and eStargz [8, 58] v0.6.3.[6] The **baseline** implementation first downloads and decompresses all new compressed layers before launching the container. **eStargz** presorts the files in each compressed layer according their expected order of or use, and uses on-demand "lazy" download during deployment to handle for unexpected accesses: when a running container opens a file whose contents are not yet available, eStargz pauses the container and requests the file from the registry. We also plot two reference times: **warm startup** time denotes the container startup time once its image has already been downloaded and decompressed to local storage; **wget** time denotes the time to compute and download the Starlight delta bundle over the network, serving as a lower bound on provisioning time when not starting containers early.

Containers We evaluate Starlight on a variety of popular containers from Docker Hub [30]. Since many of the containers in the original HelloBench container suite [28] are outdated and can no longer be deployed using moderns tools, we instead take several of its most popular containers, finally, we add several container images used in edge computing applications. The full list of containers is available in Appendix A.1.

[5] This is the official Docker registry server [30, 49].

[6] We do not compare to Slacker [28] as its source is not public and since eStargz is explicitly designed to supersede it in performance and features. Similarly, our preliminary experiments showed eStargz offers similar or superior performance to DADI [37].

5.2 Provisioning Time

Figure 4 shows the average normalized provisioning time for all the containers in Table 2 across a range of round-trip times (RTT) and network bandwidths. We normalize the provisioning time of each container to the time it takes to deploy a fresh worker using the baseline approach over a 100Mbps network with 0.15ms RTT. We also show the 95% confidence intervals to help establish statistical significance [12].

Our first immediate observation is that Starlight is the fastest provisioning approach for all latencies, bandwidths, and scenarios we study, except when provisioning fresh workers in the cloud, where Starlight provides similar performance to eStargz. It is significantly faster than both the state-of-the-art baseline approach and eStargz. Overall, Starlight provisioning is 3.0× faster on average than the baseline, and 1.9× faster than eStargz. Surprisingly, Starlight also frequently outperforms wget. In other words, Starlight early start design and effective scheduling of file payloads allows it to provision a fresh worker faster than the time it takes to merely download an optimized package. Conversely, eStargz, which also starts containers early, is on average slower than wget except when bandwidth is 54Mbps and RTT is low. Neither early start nor building optimized container images is sufficient in isolation; Starlight effectiveness is the result of its holistic design.

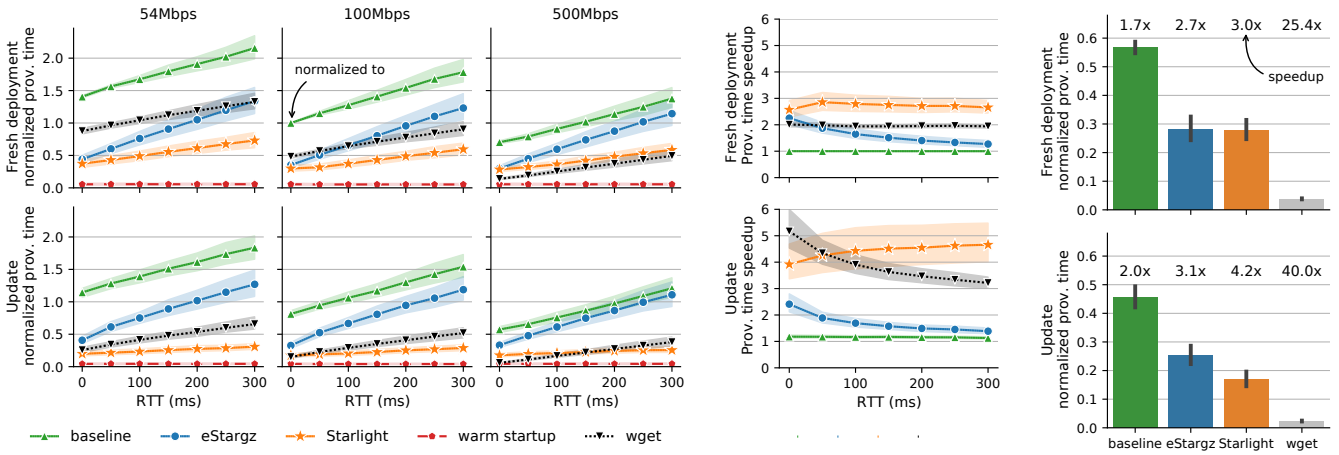
Effect of Latency When RTT is very low (i.e., inside a single datacenter), Both Starlight and eStargz are significantly faster than the baseline. However eStargz scales poorly when RTT grows due to its pull-based design that requests "out-of-order" files on-demand (§3). As latency grows, delays due to these requests add up: eStargz's provisioning time at RTT=300ms grows by 3.7× when going from RTT of 0ms to 300ms on a 500Mbps network. In comparison, the baseline provision time only doubles. For high bandwidth, high latency networks (e.g., satellite links) eStargz performance is close to the baseline approach, especially for updates.

Starlight, on the other hand, is far less sensitive to latency than the other approaches: its provisioning time grows at a slower rate than the baseline, eStargz, and wget. Starlight scales well not because its prediction of file access order is perfect (it is not), but rather due to its push-based design. Unlike eStargz, Starlight avoids flooding the registry with HTTP requests when containers open files "out of order", and instead waits for the file to arrive.

Deploying Updates Since updates are a common operation (§3.3), we also consider the provisioning time for updating containers on existing workers.

Starlight is very successful in optimizing updates: provisioning updates using Starlight (bottom row of Figure 4) is on average 1.7× faster than an equivalent fresh deployment (top row) using Starlight, and 2.5× faster than baseline fresh deployment.[7] The other approaches only show modest

[7] Harmonic mean of speedups across all bandwidths and latencies.



(a) Edge and WAN.

(b) Speedup with 100Mbps.

(c) Cloud time and speedup.

Figure 4: (a) Normalized provisioning time for different methods, round-trip times, and network bandwidth, aggregated across containers in Table 2. Solid line shows the geometric mean [19]; shaded areas show 95% confidence intervals. Top row shows fresh deployment, bottom row shows updates. Time is normalized to fresh deployment of the same container using the baseline approach with RTT of 0ms and a 100Mbps connection. (b) Speedups over baseline with 100Mbps (solid line shows harmonic mean). (c) Provisioning time and speedup in the cloud (RTT approximately ∼0.15ms, no bandwidth restriction).

improvement when provisioning updates: average update provisioning time for baseline and eStargz are close to those of fresh deployment. Additionally, we observe that Starlight update provisioning scales much better than the two other approaches as RTT grows. Finally, Starlight's transfer volume is smaller: the size of a median Starlight update is 30% that of a fresh update using the size of a baseline fresh deployment, while for eStargz and the baseline updates are 99% (figure omitted for space).

As we discuss in §3, layer reuse is low in real-world containers, and even the on-demand "lazy" approach of eStargz must still fetch file metadata from all layers. Conversely, Starlight optimizes updates at a finer file-level granularity, and also stores all file metadata at the beginning of the delta bundle. The result is that Starlight is much better able to exploit redundancy in updates, significantly outperforming the benchmark approaches.

Effect of Bandwidth Can increasing bandwidth help mitigate slow provisioning time? We find that higher bandwidth does not provide a corresponding improvement in provisioning time at higher RTT, even for the baseline approach at 0.15ms. This is not surprising: container provisioning is not purely bandwidth-bound task, since we must also decompress and start containers.

Very low bandwidth We repeated our experiments with a 5Mbps network. At such low bandwidth, transmission time overwhelms the effect of latency: normalized provisioning time for fresh deployments is 9–10.5× higher (compared to 100Mbps network with 0.15ms RTT) for baseline and wget, while eStargz and Starlight reduce it to 2.5–4×. For updates, the baseline is 8–9×, wget and eStargz are 2.5–3×, and Starlight the fastest at 1× across the range of RTT values.

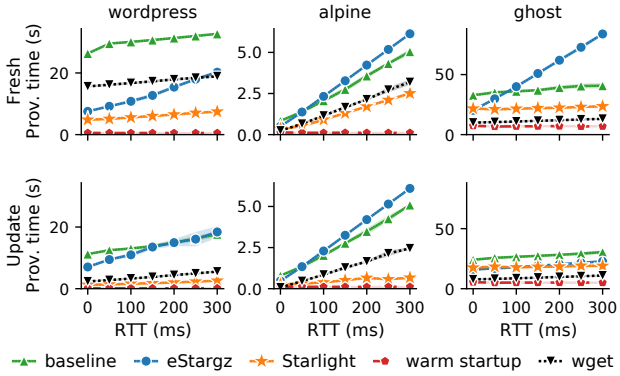


Figure 5: Provisioning times versus round-trip latency for selected containers. Shaded areas show standard deviation.

Interestingly, the network is so slow that Flink class loader times out when opening one of the class files when provisioning with eStargz and Starlight. This is the only case we have found of timeout due to on-demand downloading. Indeed, such timeouts are very rare in practice since most software does not timeout on read-only `open()` calls, and software that does must handle timeouts correctly to function with NFS mounts and other distributed filesystems. Nevertheless, we could mitigate such issues by automatically or manually sorting these files earlier in the delta bundle. Starlight's on-demand optimizer makes this straightforward.

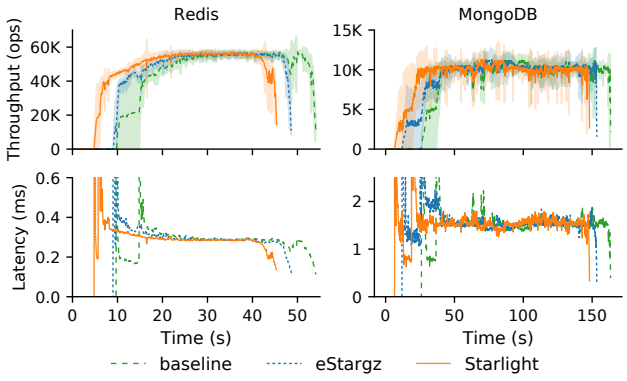


Figure 6: Redis (left) and MongoDB (right) performance during provisioning. Shaded areas show standard deviation.

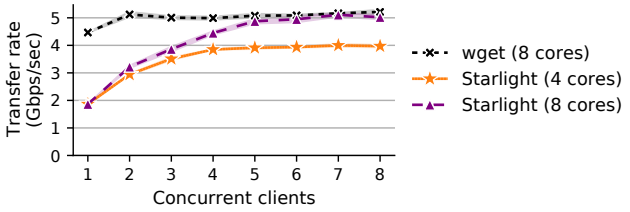


Figure 7: Scalability of Starlight proxy as the achieved transfer rate for different number of concurrent workers. Network bandwidth is capped at 5Gbps, and RTT is ∼0.15ms.

Individual Analysis Figure 5 shows provisioning time of selected containers across a range of latencies at 100Mbps.

We find that eStargz is bottlenecked by queuing delays caused by on-demand file downloads, and can be slower than even the baseline for RTT over 50ms. Starlight outperforms both except for ghost at 0ms, which is the worst case for Starlight: a 84K file container whose delta bundle takes 3 seconds to build. See Appendix A.2 for in-depth analysis.

5.3 Performance

We measure application, worker, and proxy performance. Unless otherwise noted, proxy-worker RTT is set to 150ms.

Application Performance Ideally, containers deployed using Starlight would exhibit similar application performance as those deployed using the baseline approach, especially during provisioning when Starlight is decompressing files.

To confirm this, we measure application performance for two databases: Redis (in-memory) and MongoDB (disk-based). We run YCSB [9] Workload A (50% read/write ratio) on a separate `m5a.large` instance as the client while we perform a fresh deployment the containerized application, and measure the throughput and read latency of database operations. We repeat each experiment 5 times; each run consists of 2 million database operations, long enough sufficient to finish provisioning and for application performance to stabilize.

Figure 6 depicts throughput and latency over time for both applications. With Starlight, the worker starts handling requests and finishes processing workload earlier than with the other two methods. Additionally, it reaches the same maximum throughput and minimum query latency.

In summary, Starlight workers exhibit no performance overhead compared to the baseline approach and eStargz, and moreover the time gained by early provisioning directly translates to finishing jobs faster.

Worker CPU Usage and Memory We measured the total CPU time used by the snapshotter and containerd daemons during provisioning of containers in Table 2. CPU usage is largely determined by image size, up to 40 seconds of CPU time for the largest image. Median CPU time was 12 seconds for the baseline, 15.1 seconds for eStargz, and 9.8 seconds for Starlight, since it is more effective in removing cross-layer duplicate files. This is consistent with our finding that containerized application exhibit no performance overhead.

Starlight memory usage, measured as total maximum resident set size of the snapshotter and containerd daemons, is linear in the number of files (140MB plus 9.5KB per file, R^2=0.784) since it maintains file metadata (§4.5 and §4.6). Memory use for both Starlight and eStargz is similar, ranges from under 200MB for most containers to 1GB for ghost – a massive container image with over 84K files. A recent analysis [64] finds that the median container image has 1,090 files, while 70% of images have less fewer 20,000 files – approximately 330MB for Starlight.

Optimization Time Optimizing the delta bundle is by far the most computationally intense operation for the proxy. We find we can compute delta bundles for images of up to 30K files in under one second (figure omitted for space), which includes most of Table 2; the sole exception is ghost at 84K files, which takes three seconds. Similarly, 80% of the container images in Docker Hub [64] have fewer than 30K files, and could therefore be processed within one second. Finally, the time to build delta bundle could be eliminated completely for common deployments by placing a cache in front of the Starlight proxy; we do not do so in any of our experiments.

Scalability We use Apache Benchmark to measure the achievable transfer rate of the Starlight proxy as we increase the number of concurrent clients repeatedly requesting the Redis delta bundle (36.8MB). This is equivalent to the common setup where hundreds of simultaneous Starlight worker requests are load-balanced across multiple replicas of the proxy, and the goal is to saturate the bandwidth – if the proxy is network bound, we are serving as many clients as the network supports. For this experiment, we run with no artificial bandwidth or latency limits. For reference, we request an image of equivalent size from an nginx webserver. Figure 7 shows a Starlight proxy running on a 4-core instance is able to saturate about 80% of the link bandwidth before becoming

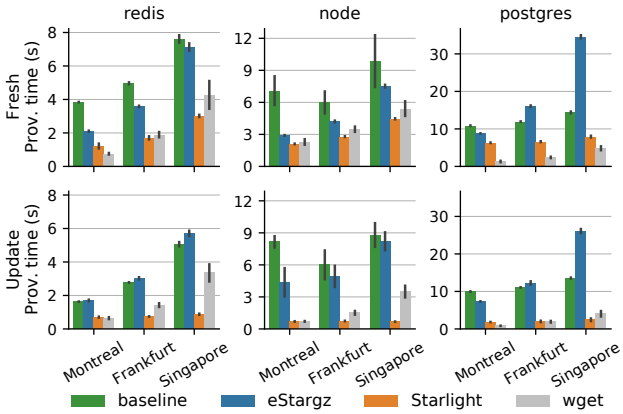


Figure 8: Provisioning time when moving the worker between different datacenters. Errors bars show standard deviation. The registry is located in North Virginia.

bottlenecked due to the need to optimize the delta bundle. Once we switch to 8 cores, it becomes network bound.

5.4 Geo-Distributed WAN Experiment

Thus far we have evaluated Starlight using controlled experiments in a single AWS datacenter. Here, we Starlight performance in a multi-cloud (wide area network) setup running in multiple datacenters over the real network. We place the registry in `us-east-1` region (N. Virginia) and move the worker to increasingly distant locations: `ca-central-1` (average RTT to registry 14ms, bandwidth 4.24Gbps), `eu-central-1` (89ms, 2.79Gbps), and `ap-southeast-1` (209ms, 1.15Gbps).

Figure 8 shows the provisioning time for fresh and update deployment. Results generally match our previous observations: Starlight substantially outperforms the baseline and eStargz, and in many cases is faster than a simple wget of the delta bundle. eStargz is sensitive to increased latency, in some cases becoming slower than the baesline approach. Finally, Starlight support for container updates is much more effective than the other approaches, and can reduce provisioning time to a fraction of the other approaches.

6 Related Work

There are several streams of work on container provisioning.

On-Demand Download Slacker [28] starts containers early and uses NFS to load files on-demand without requiring the entire container image. CRFS [27] follows a similar idea, but uses a seekable tar gzip format with more efficient compression, allowing it to work with standard registries. DADI [37] also uses on-demand fetching but operates at the block level, which requires a customized image format and registry. eStargz [58] uses collected filesystem traces to identify files needed during provisioning and prefetch them first, before switching to on-demand downloading. Starlight also sorts files based on collected traces, but its push-based design scales better with higher latency. Moreover, Starlight's protocol is file-based rather than layer-based as prior approaches.

Peer-to-peer Some approaches use workers to help provision other workers, Wharf [65] and Shifter [26] propose client-side image sharing: workers act as caches, serving locally stored images to other workers. FID [32], CoMICon [44], and Kraken [31] are P2P docker registries that help reduce registry load by utilizing the bandwidth of workers in the datacenter. Similarly, FaaSNet [60] uses a tree of workers to accelerate provisioning inside datacenters for scsaling Function-as-as-Service workloads inside a datacenter. These approaches tend to focus on single datacenter setting with the goal of reducing registry load. They may not be applicable outside the datacenter or where bandwidth and other worker resources are limited. Conversely, Starlight is focused on accelerating provisioning without increasing worker load.

Registry optimizations Fu et al. [25] and Anwar et al. [4] proposes smart caching and prefetching image layers from the back-end object store to the registry using the production workload, in order to do large scale software provisioning. Starlight is orthogonal to, and compatible with, these works since it does not require changing the registry.

7 Conclusion

Containers have evolved in a single datacenter environment, but are increasingly used in geo-distributed settings such as edge, mobile, and multi-cloud environments. We revisit several of the design decisions behind containers, and show that while they are convenient for developers, they slow down provisioning. Starlight redesigns the provisioning pipeline to support faster container deployment, while maintaining the layer-based structure that makes containers easy to develop and maintain. Empirical evaluation using a large set of popular containers shows Starlight provisioning times are significantly smaller than existing approaches, while incurring no performance overhead. Moreover, Starlight is backwards compatible and makes use of existing registries. Starlight is available as an open-source project at: https://github.com/mc256/starlight.

Starlight's design opens several avenues for improvement. For example, since the delta bundle is optimized on-demand, we can improve it and even tailor it to specific scenarios by collecting traces online during deployment, or by training an ML model to predict which files will be needed first. Another improvement is support for repurposing workers: by modifying the optimizer and extending the delta bundle design, we could optimize switching between arbitrary sets of containers.

References

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

[2] Werner Almesberger. Linux traffic control - implementation overview. Technical report, EPFL ICA, 1998.

[3] Amazon. Amazon Elastic Container Service (Amazon ECS). https://aws.amazon.com/ecs/.

[4] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278, Oakland, CA, February 2018. USENIX Association.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[6] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. Timing the application of security patches for optimal uptime. In *16th Systems Administration Conference (LISA 02)*, Philadelphia, PA, November 2002. USENIX Association.

[7] Containerd. Snapshots design. https://github.com/containerd/containerd/blob/main/design/snapshots.md.

[8] Containerd. Stargz snapshotter. https://github.com/containerd/stargz-snapshotter.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Lorenzo Corneo, Maximilian Eder, Nitinder Mohan, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, Per Gunningberg, Jussi Kangasharju, and Jörg Ott. *Surrounded by the Clouds: A Comprehensive Cloud Reachability Study*, page 295–304. Association for Computing Machinery, New York, NY, USA, 2021.

[11] Breno Costa, Joao Bachiega, Leonardo Rebouças de Carvalho, and Aleteia P. F. Araujo. Orchestration in fog computing: A comprehensive survey. *ACM Comput. Surv.*, 55(2), January 2022.

[12] Geoff Cumming, Fiona Fidler, and David L. Vaux. Error bars in experimental biology . *Journal of Cell Biology*, 177(1):7–11, 04 2007.

[13] Richard Cziva and Dimitrios P. Pezaros. Container network functions: Bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31, 2017.

[14] Bradley Denby and Brandon Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 939–954, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Docker. Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

[16] Docker. Docker compose. https://github.com/docker/compose.

[17] Docker. Docker documentation. https://docs.docker.com/engine/reference/commandline/dockerd/.

[18] Docker. Empowering app development for developers | docker. https://www.docker.com/.

[19] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[20] The Linux Foundation. Cloud native computing foundation. https://cncf.io.

[21] The Linux Foundation. containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/.

[22] The Linux Foundation. K3s: Lightweight kubernetes. https://k3s.io.

[23] The Linux Foundation. Kubernetes. https://kubernetes.io/.

[24] The Linux Foundation. Open container initiative. https://opencontainers.org/.

[25] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. Fast and efficient container startup at the edge via dependency scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[26] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. Shifter: Containers for HPC. In *Journal of physics: Conference series*, volume 898, page 082021. IOP Publishing, 2017.

[27] Google. CRFS: Container registry filesystem. https://github.com/google/crfs.

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

[29] Stephen Hemminger. Network emulation with NetEm. In *Linux Conf Australia*, pages 18–23, 2005.

[30] Docker Inc. Docker Hub: Container image library | app containerization. https://registry.hub.docker.com/.

[31] Uber Inc. Kraken - p2p docker registry capable of distributing tbs of data in seconds. https://github.com/uber/kraken.

[32] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. Fid: A faster image distribution system for docker platform. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 191–198, 2017.

[33] The kernel development community. Fuse the linux kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[34] kernel.org. Overlay filesystem – the linux kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[35] Petros Koutoupis. Everything you need to know about Linux containers, part i: Linux control groups and process isolation. *Linux Journal*, 2018, 2018.

[36] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

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

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

[39] Scott McCarty. A practical introduction to container terminology, 2018. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction.

[40] Isla Mcketta. How Starlink's satellite internet stacks up against HughesNet and Viasat around the globe, 2021. https://www.speedtest.net/insights/blog/starlink-hughesnet-viasat-performance-q2-2021/.

[41] Microsoft. Azure container instances. https://azure.microsoft.com/en-us/services/container-instances/.

[42] Seyed Hossein Mortazavi, Mohammad Salehe, Moshe Gabel, and Eyal de Lara. Feather: Hierarchical querying for the edge. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 271–284, 2020.

[43] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. CoMICon: A co-operative management system for docker container images. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 116–126, 2017.

[45] Mahesh Nayak, Kumud Dwivedi, and Cheryl McGuire. Azure network round-trip latency statistics, 2021. https://docs.microsoft.com/en-us/azure/networking/azure-network-latency.

[46] The Containers Organization. Buildah: a tool that facilitates building open container initiative (oci) container images. https://buildah.io/.

[47] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. Toward lighter containers for the edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[48] Valerio Persico, Alessio Botta, Pietro Marchetta, Antonio Montieri, and Antonio Pescap. On the performance of the wide-area networks interconnecting public-cloud datacenters around the globe. *Comput. Netw.*, 112(C):67–83, January 2017.

[49] CNCF Distribution Project. Distribution - the toolkit to pack, ship, store, and deliver container content. https://github.com/distribution/distribution.

[50] Prashanth Rajivan, Efrat Aharonov-Majar, and Cleotilde Gonzalez. Update now or later? effects of experience, cost, and risk preference on update decisions. *Journal of Cybersecurity*, 6(1):tyaa002, 2020.

[51] Brian Ramprasad, Alexandre da Silva Veith, Moshe Gabel, and Eyal de Lara. Sustainable computing on the edge: A system dynamics perspective. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, HotMobile '21, page 64–70, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[53] J. Shah and D. Dubaria. Building modern clouds: Using Docker, Kubernetes & Google Cloud Platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189, 2019.

[54] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[55] Dimitris Skourtis, Lukas Rupprecht, Vasily Tarasov, and Nimrod Megiddo. Carving perfect layers out of docker images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[56] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, Boston, MA, July 2018. USENIX Association.

[57] Abhishek Tiwari, Brian Ramprasad, Seyed Hossein Mortazavi, Moshe Gabel, and Eyal de Lara. Reconfigurable streaming for the mobile edge. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, page 153–158, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Kohei Tokunaga. Startup containers in lightning speed with lazy image distribution on containerd, Apr 2020.

[59] B. Varghese, E. De Lara, A. Ding, C. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele, and P. Willis. Revisiting the arguments for edge computing research. *IEEE Internet Computing*, (01):1–1, jun 5555.

[60] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[61] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[62] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[63] N. Zhao, V. Tarasov, A. Anwar, L. Rupprecht, D. Skourtis, A. Warke, M. Mohamed, and A. Butt. Slimmer: Weight loss secrets for Docker registries. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 517–519, 2019.

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

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

A Appendix

A.1 Container Images Used in Evaluation

Table 2 below lists containers and version tags used in our experiments; combined, they have over 15 billion downloads in Docker Hub.

Category	Images
Linux	alpine:3.13.4
	ubuntu:focal-20210401
Web	memcached:1.6.8
	nginx:1.19.10
	httpd:2.4.43
Data	mysql:8.0.23
	mariadb:10.5.8
	redis:6.2.1
	mongo:4.0.23
	postgres:13.1
	rabbitmq:3.8.13
Services	registry:2.7.0
	wordpress:php7.3-fpm
	ghost:3.42.5-alpine
Dev	node:16-alpine3.11
	openjdk:11.0.11-9-jdk
	golang:1.16.2
	python:3.9.3
Edge	flink:1.12.3-scala_2.11-java8
	cassandra:3.11.9
	eclipse-mosquitto:2.0.9-openssl

Table 2: Container images used in our evaluation.

A.2 Analysis of Selected Containers

Figure 5 shows provisioning time of selected containers across a range of latencies at 100Mbps.

When updating wordpress, the baseline approach is able to reuse 4 out of 18 layers, making it faster in update. eStargz, though faster than the baseline approach in fresh deployments, does not benefit much from this layer reuse since it is bottlenecked by on-demand file downloads. Starlight, on the other hand, is much faster than either approach, reducing update provisioning time by approximately 8×.

For alpine eStargz is slower than the baseline when RTT is above 50ms. This is because the alpine image is small and its file access pattern is not entirely deterministic. Provisioning time is thus dominated by queuing delays due layer downloads and on-demand file downloads. Starlight also suffers somewhat from out-of-order file accesses, but is still able to deploy the container quickly, and is even faster than wget.

Finally, we discuss ghost – the worst case for Starlight. Starlight's provisioning time with low RTT is 10% higher than eStargz's – the only container where this happens. Building a delta bundle takes 3 seconds for this 84K file container. eStargz provisioning time grows quickly with latency, however, and Starlight outperforms it when RTT is above 50ms.

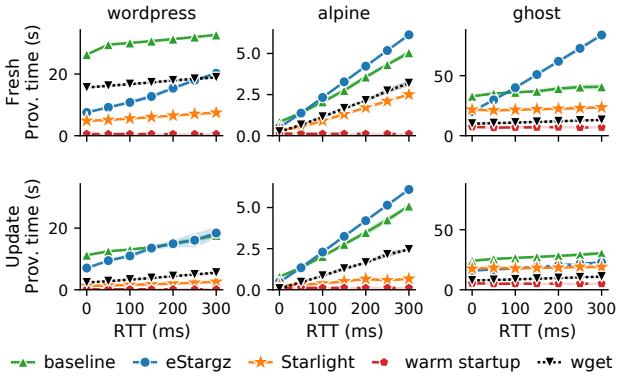


Figure 5: Provisioning times versus round-trip latency for selected containers. Shaded areas show standard deviation. (figure repeated from page 10)