Improving Docker Registry Design based on Production Workload Analysis

Ali Anwar, *Virginia Tech;* Mohamed Mohamed and Vasily Tarasov, *IBM Research—Almaden;*
Michael Littley, *Virginia Tech;* Lukas Rupprecht, *IBM Research—Almaden;*
Yue Cheng, *George Mason University;* Nannan Zhao, *Virginia Tech;* Dimitrios Skourtis,
Amit S. Warke, and Heiko Ludwig, and Dean Hildebrand, *IBM Research—Almaden;*
Ali R. Butt, *Virginia Tech*

https://www.usenix.org/conference/fast18/presentation/anwar

This paper is included in the Proceedings of the
16th USENIX Conference on File and Storage Technologies.
February 12–15, 2018 • Oakland, CA, USA

ISBN 978-1-931971-42-3

Open access to the Proceedings of
the 16th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

Improving Docker Registry Design based on Production Workload Analysis

Ali Anwar[1], Mohamed Mohamed[2], Vasily Tarasov[2], Michael Littley[1], Lukas Rupprecht[2], Yue Cheng[3]*, Nannan Zhao[1], Dimitrios Skourtis[2], Amit S. Warke[2], Heiko Ludwig[2], Dean Hildebrand[2]†, and Ali R. Butt[1]

[1]*Virginia Tech,* [2]*IBM Research–Almaden,* [3]*George Mason University*

Abstract

Containers offer an efficient way to run workloads as independent microservices that can be developed, tested and deployed in an agile manner. To facilitate this process, container frameworks offer a registry service that enables users to publish and version container images and share them with others. The registry service plays a critical role in the startup time of containers since many container starts entail the retrieval of container images from a registry. To support research efforts on optimizing the registry service, large-scale and realistic traces are required. In this paper, we perform a comprehensive characterization of a large-scale registry workload based on traces that we collected over the course of 75 days from five IBM data centers hosting production-level registries. We present a trace replayer to perform our analysis and infer a number of crucial insights about container workloads, such as request type distribution, access patterns, and response times. Based on these insights, we derive design implications for the registry and demonstrate their ability to improve performance. Both the traces and the replayer are open-sourced to facilitate further research.

1 Introduction

Container management frameworks such as Docker [22] and CoreOS Container Linux [3] have established containers [41, 44] as a lightweight alternative to virtual machines. These frameworks use Linux *cgroups* and *namespaces* to limit the resource consumption and visibility of a container, respectively, and provide isolation in shared, multi-tenant environments at scale. In contrast to virtual machines, containers share the underlying operating system kernel, which enables fast deployment with low performance overhead [35]. This, in turn, is driving the rapid adoption of the container technology in the enterprise setting [23].

The utility of containers goes beyond performance, as they also enable a *microservice* architecture as a new model for developing and distributing software [16, 17, 24]. Here, individual software components focusing on small functionalities are packaged into container *images* that include the software and all dependencies required to run it. These *microservices* can then be deployed and combined to construct larger, more complex architectures using lightweight communication mechanisms such as REST or gRPC [9].

To facilitate the deployment of microservices, Docker provides a *registry service*. The registry acts as a central image repository that allows users to publish their images and make them accessible to others. To run a specific software component, users then only need to "pull" the required image from the registry into local storage. A variety of Docker registry deployments exist such as Docker Hub [5], IBM Cloud container registry [12], or Artifactory [1].

The registry is a data-intensive application. As the number of stored images and concurrent client requests increases, the registry becomes a performance bottleneck in the lifecycle of a container [37, 39, 42]. Our estimates show that the widely-used public container registry, Docker Hub [5], stores at least hundreds of terabytes of data, and grows by about 1,500 new public repositories daily, which excludes numerous private repositories and image updates. Pulling images from a registry of such scale can account for as much as 76% of the container start time [37]. Several recent studies have proposed novel approaches to improve Docker client and registry communication [37, 39, 42]. However, these studies only use small datasets and synthetic workloads.

In this paper, for the first time in the known literature, we perform a large-scale and comprehensive analysis of a real-world Docker registry workload. To achieve this, we started with collecting long-span production-level traces from five datacenters in IBM Cloud container registry service. IBM Cloud serves a diverse set of customers, ranging from individuals, to small and medium businesses, to large enterprises and government institutions. Our traces cover all availability zones and many components of the registry service over the course of 75 days, which totals to over 38 million requests and accounts for more than 181.3 TB of data transferred.

We sanitized and anonymized the collected traces and then created a high-speed, distributed, and versatile Docker trace replayer. To the best of our knowledge, this is the first trace replayer for Docker. To facilitate future research and engineering efforts, we release

*Most of this work was done while at Virginia Tech.
†Now at Google.

both the anonymized traces and the replayer for public use at https://dssl.cs.vt.edu/drtp/. We believe our traces can provide valuable insights into container registry workloads across different users, applications, and datacenters. For example, the traces can be used to identify Docker registry's distinctive access patterns and subsequently design workload-aware registry optimizations. The trace replayer can be used to benchmark registry setups as well as for testing and debugging registry enhancements and new features.

We further performed comprehensive characterization of the traces across several dimensions. We analyzed the request ratios and sizes, the parallelism level, the idle time distribution, and the burstiness of the workload, among other aspects. During the course of our investigation, we made several insightful discoveries about the nature of Docker workloads. We found, for example, that the workload is highly read-intensive comprising of 90-95% pull compared to push operations. Given the fact that our traces come from several datacenters, we were able to find both common and divergent traits of different registries. For example, our analysis reveals that the workload not only depends on the purpose of the registry but also on the age of the registry service. The older registry services show more predictable trends in terms of access patterns and image popularity. Our analysis, in part, is tailored to exploring the feasibility of caching and prefetching techniques in Docker. In this respect, we observe that 25% of the total requests are for top 10 repositories and 12% of the requests are for top 10 layers. Moreover, 95% of the time is spent by the registry in fetching the image content from the backend object store. Finally, based on our findings, we derive several design implications for container registry services.

2 Background

Docker [22] is a container management framework that facilitates the creation and deployment of containers. Each Docker container is spawned from an *image*—a collection of files sufficient to run a specific containerized application. For example, an image which packages the Apache web server contains all dependencies required to run the server. Docker provides convenient tools to combine files in images and run containers from images on end hosts. Each end host runs a daemon process which accepts and processes user commands.

Images are further divided into *layers*, each consisting of a subset of the files in the image. The layered model allows images to be structured in sub-components which can be shared by other containers on the same host. For example, a layer may contain a certain version of the Java runtime environment and all containers requiring this version can share it from a single layer, reducing storage and network utilization.

2.1 Docker Registry

To simplify their distribution, images are kept in an online *registry*. The registry acts as a storage and content delivery system, holding named Docker images. Some popular Docker registries are Docker Hub [5], Quay.io [20], Artifactory [1], Google Container Registry [8], and IBM Cloud container registry [12].

Users can create *repositories* in the registry, which hold images for a particular application or system such as Redis, WordPress, or Ubuntu. Images in such repositories are often used for building other application images. Images can have different versions, known as *tags*. The combination of user name, repository name, and tag uniquely identifies an image.

Users add new images or update existing ones by pushing to the registry and retrieve images by pulling from the registry. The information about which layers constitute a particular image is kept in a metadata file called *manifest*. The manifest also describes other image settings such as target hardware architecture, executable to start in a container, and environment variables. When an image is pulled, only the layers that are not already available locally are transferred over the network.

In this study we use Docker Registry's version 2 API which relies on the concept of content addressability. Each layer has a content addressable identifier called *digest*, which uniquely identifies a layer by taking a collision-resistant hash of its data (SHA256 by default). This allows Docker to efficiently check whether two layers are identical and deduplicate them for sharing between different images.

Pulling an Image. Clients communicate with the registry using a RESTful HTTP API. To retrieve an image, a user sends a `pull` command to the local Docker daemon. The daemon then fetches the image manifest by issuing a `GET <name>/manifests/<tag>` request, where `<name>` defines user and repository name while `<tag>` defines the image tag.

Among other fields, manifest contains name, tag, and fsLayers fields. The daemon uses the digests from the fsLayers field to download individual layers that are not already available in local storage. The client checks if a layer is available in the registry by using `HEAD <name>/blobs/<digest>` requests.

Layers are stored in the registry as compressed tarballs ("blobs" in Docker terminology) and are pulled by issuing a `GET <name>/blobs/<digest>` request. The registry can redirect layer requests to a different URL, e.g., to an object store, which stores the actual layers. In this case, the Docker client downloads the layers directly from the new location. By default, the daemon downloads and extracts up to three layers in parallel.

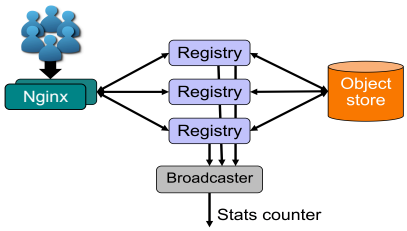


Figure 1: IBM Cloud Registry architecture. Nginx receives users requests and forwards them to registry servers. Registry servers fetch data from the backend object store and reply back.

Pushing an Image. To upload a new image to the registry or update an existing one, clients send a `push` command to the daemon. Pushing works in reverse order compared to pulling. After creating the manifest locally the daemon first pushes all the layers and then the manifest to the registry.

Docker checks if a layer is already present in the registry by issuing a `HEAD <name>/blobs/<digest>` request. If the layer is absent, its upload starts with a `POST <name>/blobs/uploads/` request to the registry which returns a URL containing a unique upload identifier (`<uuid>`) that the client can use to transfer the actual layer data. Docker then uploads layers using *monolithic* or *chunked* transfers. Monolithic transfer uploads the entire data of a layer in a single `PUT` request. To carry out chunked transfer, Docker specifies a byte range in the header along with the corresponding part of the blob using `PATCH <name>/blobs/uploads/<uuid>` requests. Then Docker submits a final `PUT` request with a layer digest parameter. After all layers are uploaded, the client uploads the manifest using `PUT <name>/manifests/<digest>` request.

2.2 IBM Cloud Container Registry

In this work we collect traces from IBM's container registry which is a part of the IBM Cloud platform [11]. The registry is a key component for supporting Docker in IBM Cloud and serves as a sink for container images produced by build pipelines and as the source for container deployments. The registry is used by a diverse set of customers, ranging from individuals, to small and medium businesses, to large enterprises and government institutions. These customers use the IBM container registry to distribute a vast variety of images that include operating systems, databases, cluster deployment setups, analytics frameworks, weather data solutions, testing infrastructures, continuous integration setups, etc.

The IBM Cloud container registry is a fully managed, highly available, high-performance, v2 registry based on the open-source Docker registry [4]. It tracks the Docker project codebase in order to support the majority of the latest registry features. The open-source functionality is extended by several microservices, offering features such as multi-tenancy with registry namespaces, a vulnerability advisor, and redundant deployment across availability zones in different geographical regions.

IBM's container registry stack consists of over eighteen components. Figure 1 depicts three components that we trace in our study: 1) Nginx, 2) registry servers, and 3) broadcaster. Nginx acts as a load balancer and forwards customers' HTTPS connections to a selected registry server based on the requested URL. Registry servers are configured to use OpenStack Swift [18, 25, 26] as a backend object store. The broadcaster provides registry event filtering and distribution, e.g., it notifies the vulnerability advisor component on new image pushes.

Though all user requests to the registry pass through Nginx, Nginx logs contain only limited information. To obtain complete information required for our analysis we also collected traces at registry servers and broadcaster. Traces from registry servers provide information about request distribution, traces from Nginx provide response time information, and broadcaster traces allow us to study layer sizes.

The IBM container registry setup spans five geographical locations: Dallas (`dal`), London (`lon`), Frankfurt (`fra`), Sydney (`syd`), and Montreal. Every geographical location forms a single Availability Zone (AZ), except Dallas and Montreal. Dallas hosts Staging (`stg`) and Production (`dal`) AZs, while Montreal is home for Prestaging (`prs`) and Development (`dev`) AZs. The `dal`, `lon`, `fra`, and `syd` AZs are client-facing and serving production workloads, while `stg` is a staging location used internally by IBM employees. `prs` and `dev` are used exclusively for internal development and testing of the registry service. Out of the four production registries `dal` is the oldest, followed by `lon`, and `fra`. `Syd` is the youngest registry and we started collecting traces for it since its first day of operation.

Each AZ has an individual control plane and ingress paths, but backend components, e.g. , object storage, are shared. This means that AZ's are completely network isolated but images are shared across AZ's. The registry setup is identical in hardware, software, and system configuration across all AZs, except for `prs` and `dev`. `prs` and `dev` are only half the size of the other AZs, because they are used for development and testing and do not directly serve clients. Every AZ hosts six registry instances, except for `prs` and `dev`, which host three.

3 Tracing Methodology

To collect traces from the IBM Cloud registry, we obtained access to the system's logging service (§3.1). The logging service collects request logs from the different system components and the log data contains a variety of information, such as the requested image, the type of request and a timestamp (§3.2). This information is sufficient to carry out our analysis. Besides collecting the

Aavailablity Zone	Duration (days)	Trace data (GB)	Filtered and anonym. (GB)	Requests (millions)	Data ingress (TB)	Data egress (TB)	Images pushed (1,000)	Images pulled (1,000)	Up since (mm/yy)
Dallas (dal)	75	115	12	20.85	5.50	107.5	356	5,000	06/15
London (lon)	75	40	4	7.55	1.70	25.0	331	2,200	10/15
Frankfurt (fra)	75	17	2	1.80	0.40	3.30	90	950	04/16
Sydney (syd)	65	5	0.5	1.03	0.29	1.87	105	360	04/16
Staging (stg)	65	25	3.2	5.90	2.41	29.2	327	1,560	-
Prestaging (prs)	65	4	0.5	0.75	0.23	2.45	65	140	-
Development (dev)	55	2	0.2	0.34	0.01	1.44	15	70	-
TOTAL	475	208	22.4	38.22	10.54	170.76	1289	10280	-

Table 1: Characteristics of studied data. `dal` and `lon` were migrated to v2 in April 2016.


```
{
    "host": "579633fd",
    "http.request.duration": 0.879271282,
    "http.request.method": "GET",
    "http.request.remoteaddr": "40535jf8",
    "http.request.uri": "v2/ca64kj67/as87d65g/blobs/b
        26s986d",
    "http.request.useragent": "docker/17.04.0-ce go/
        go1.7.5..)",
    "http.response.status": 200,
    "http.response.written": 1518,
    "id": "9f63984h",
    "timestamp": "2017-07-01T01:39:37.098Z"
}
```


Figure 2: Sample of anonymized data.

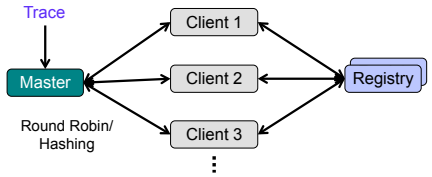


Figure 3: Trace replayer. Master parses the trace and forwards request to one of the clients either in round robin or applying hash to the `http.request.remoteaddr` field in the trace.

traces, we also developed a trace replayer (§3.3) that can be used by others to evaluate, e.g., Docker registry's performance. In this paper we used the trace replayer to evaluate several novel optimizations the were inspired by the results of the trace analysis. We made the traces and the replayer publicly available at:

`https://dssl.cs.vt.edu/drtp/`

3.1 Logging Service

Logs are centrally managed using an "ELK" stack (ElasticSearch [7], Logstash [14] and Kibana [13]). A Logstash agent on each server ships logs to one of the centralized log servers, where they are indexed and added to an ElasticSearch cluster. The logs can then be queried using the Kibana web UI or using the ElasticSearch APIs directly. ElasticSearch is a scalable and reliable text-based search engine which allows to run full text and structured search queries against the log data. Each AZ has its own ElasticSearch setup deployed on five to eight nodes and collects around 2 TB of log data daily. This includes system usage, health information, logs from different components etc. Collected data is indexed by time.

3.2 Collected Data

For trace collection we pull data from the ElasticSearch setup of each AZ for the "Registry", "Nginx", and "Broadcaster" components as shown in Figure 1. We filter all requests that relate to pushing and pulling of images, i.e. `GET`, `PUT`, `HEAD`, `PATCH` and `POST` requests. Table 1 shows the high-level characteristics of the collected traces. The total amount of our traces spans seven availability zones and a duration of 75 days from 06/20/2017 to 09/02/2017. This results in a total of 208 GB of trace data containing over 38 million requests, with more than 180TB of data transferred in them (data ingress/egress). Next, we combine the traces from different components by matching the incoming HTTP request identifier across the components. Then we remove redundant fields to shrink the trace size and in the end we anonymize them. The total size of the anonymized traces is 22.4 GB.

Figure 2 shows a sample trace record. It consists of 10 fields: the `host` field shows the anonymized registry server which served the request; `http.request.duration` is the response time of the request in seconds; `http.request.method` is the HTTP request method (e.g., `PUT` or `GET`); `http.request.remoteaddr` is the anonymized remote client IP address; `http.request.uri` is the anonymized requested url; `http.request.useragent` shows the Docker client version used to make the request; `http.response.status` shows the HTTP response code for this request; `http.response.written` shows the amount of data that was received or sent; `id` shows the unique request identifier; `timestamp` contains the request arrival time in UTC timezone.

3.3 Trace Replayer

To study the collected traces further and use them to evaluate various registry optimizations, we designed and implemented a trace replayer. It consists of a master node and multiple client nodes as shown in Figure 3. The master node parses the anonymized trace file one request at a time and forwards it to one of the clients. Requests are forwarded to clients in either round robin fashion or by hashing the `http.request.remoteaddr` field in the trace. By using hashing, the trace replayer maintains the request locality to ensure all HTTP requests corresponding to one image push or pull are generated by the same client node as they were seen by the original registry service. In some cases this option may generate workload

skewness as some of the clients issue more requests than others. This method is useful for large-scale testing with many clients.

Clients are responsible for issuing the HTTP requests to the registry setup. For all `PUT` layer requests, a client generates a random file of corresponding size and transfers it to the registry. As the content of the newly generated file is not same as the content of the layer seen in the trace, the digest/SHA256 is going to be different for the two. Hence, upon successful completion of the request, the client replies back to the master with the request latency as well as the digest of the newly generated file. The master keeps track of the mapping between the digest in the trace and its corresponding newly generated digest. For all future `GET` requests for this layer, the master issues requests for the new digest instead of the one seen in the trace. For all `GET` requests the client just reports the latency.

The trace replayer runs in two phases: warmup and actual testing. During the warmup phase, the master iterates over the `GET` requests to make sure that all corresponding manifests and layers already exist in the registry setup. In the testing phase all requests are issued in the same order as seen in the trace file.

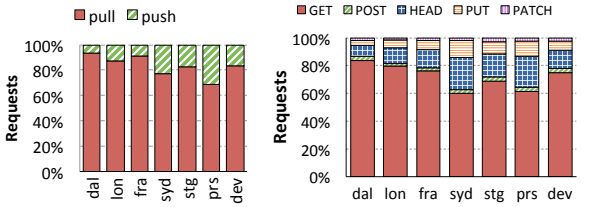
The requests are issued by the trace replayer in two modes: 1) "as fast as possible", and 2) "as is", to account for the timestamp of each request. The master side of the trace replayer is multithreaded and each client's progress is tracked in a separate thread. Once all clients finish their jobs, aggregated throughput and latency is calculated. Per-request latency and per-client latency and throughput are recorded separately.

The trace replayer can operate in two modes to perform two types of analysis: 1) performance analysis of a large scale registry setup and 2) offline analysis of traces.

Performance analysis mode. The Docker registry utilizes multiple resources (CPU, Memory, Storage, Network) and provisioning them is hard without a real workload. The performance analysis mode allows to benchmark what throughput and latency can a Docker registry installation achieve when deployed on specific provisioned resources. For example, in a typical deployment, Docker is I/O intensive and the replayer can be used to benchmark network storage solutions that act as a backend for the registry.

Offline analysis mode. In this mode, the master does not forward the requests to the clients but rather hands them off to an analytic plugin to handle any requested operation. This mode is useful to perform offline analysis of the traces. For example, the trace player can simulate different caching policies and determine the effect of using different cache sizes. In Sections §5.3 and §5.4 we use this mode to perform caching and prefetching analysis.

Additional analysis. By making our traces and trace replayer publicly available we enable more detailed analysis in the future. For example, one can create a module for the replayer's performance analysis mode that analyzes request arrival rates with a user-defined time granularity. One may also study the impact of using content delivery networks to cache popular images by running the trace replayer in the performance analysis mode. Furthermore, to understand the effect of deduplication on data reduction in the registry, researchers can conduct studies on real layers in combination with our trace replayer. The relationship between resource provisioning vs. workload demands can be established by benchmarking registry setups using our trace replayer and traces.



(a) Push vs. Pull. (b) Request Ratio.

Figure 4: Image pull vs. push ratio, and distribution of HTTP requests served by registry.

4 Workload Characterization

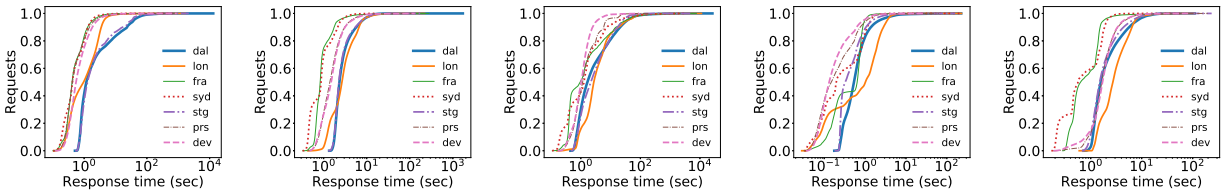
To determine possible registry optimizations, such as caching, prefetching, efficient resource provisioning, and site-specific optimizations, we center our workload analysis around the following five questions:

1. What is the general workload the registry serves? What are request type and size distributions? (§4.1)
2. Do response times vary between production, staging, pre-staging, and development deployments? (§4.2)
3. Is there spatial locality in registry requests? (§4.3)
4. Do any correlations exist among subsequent requests? Can future requests be predicted? (§4.4)
5. What are the workload's temporal properties? Are there bursts and is there any temporal locality? (§4.5)

4.1 Request Analysis

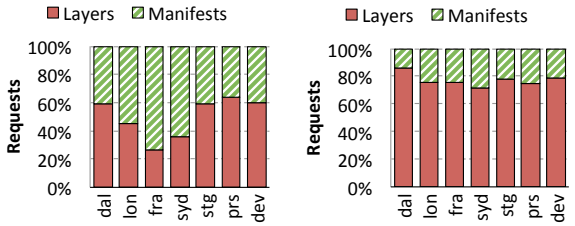
We start with the request type and size analysis to understand the basic properties of the registry's workload.

Request type distribution. Figure 4(a) shows the ratio of images pulled from vs. pushed to the registry. As expected, the registry workload is read-intensive. For `dal`, `lon`, and `fra`, we observe that 90%–95% of requests are pulls (i.e. reads). `Syd` exhibits a lower pull ratio of 78% because it is a newer installation and, therefore, it is being populated more intensively than mature registries. Non-production registries (`stg`, `prs`, `dev`) also demonstrate a lower (68–82%) rate of pulls than production registries, due to higher image churn rates. Each push



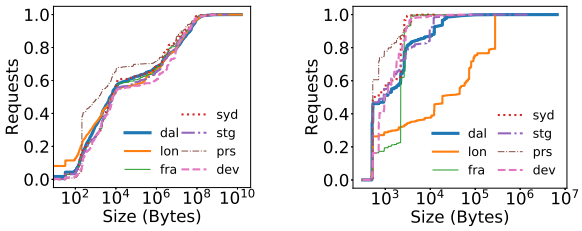
(a) GET layers. (b) PUT layers. (c) PATCH layers. (d) GET manifests. (e) PUT manifests.

Figure 8: CDF of response time for `GET`, `PUT`, `PATCH` requests to layers and `GET` and `PUT` requests to manifests.



(a) Pull. (b) Push.

Figure 5: The ratio of requests that access either an image manifest or a layer.



(a) Layers. (b) Manifests.

Figure 6: CDF of manifest and layer sizes for GET requests.

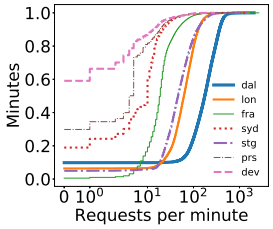


Figure 7: CDF of requests per minute.

or pull consists of a sequence of HTTP requests as discussed in §2. Figure 4(b) shows the distribution of different HTTP requests served by the registry. All registries receive more than 60% of `GET` requests and 10%–22% of `HEAD` requests. `PUT` requests are 1.9–5.8× more common than `PATCH` requests because `PUT`s are used for uploading manifests (in addition to layers) and many layers are small enough to be uploaded in a single request.

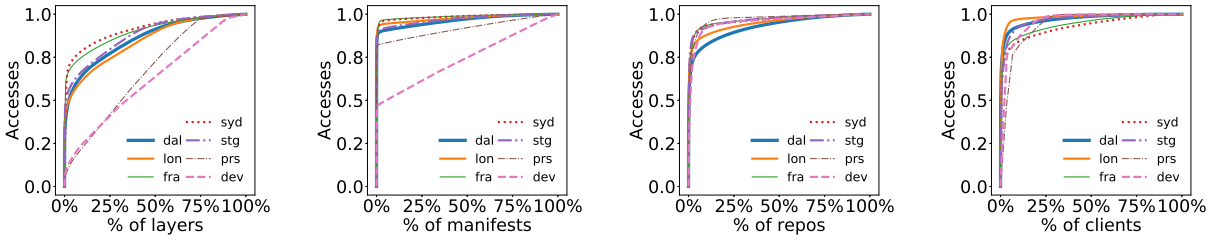
Figures 5(a) and 5(b) show the manifest vs. layer ratio for pull and push image requests, respectively. We include `GET` requests in pull count, while pushes include `PUT` *or* `HEAD` requests to account for attempts to upload the layers that are already present in the registry. For pulls we observe that, except for `syd` and `fra`, 50% or more requests retrieve layers rather than manifests. This is expected as a single manifest refers to multiple layers. Our investigation revealed that the divergent behavior of `syd` and `fra` is caused by their clients trying to pull images that they have already pulled in the past. This results into many `GET` requests to the manifests without subsequent `GET` requests to the layers. For pushes, we see that accesses to layers dominate accesses to manifests.

Request size distribution. Figure 6 shows the CDF of manifest and layer sizes for `GET` and `PUT` requests. In Figure 6(a) we observe that about 65% of the layers are smaller than 1 MB and around 80% are smaller than 10 MB. In Figure 6(b), we find that the typical manifest size is around 1 KB for all AZs except for `lon` where 50% of the GET requests are for manifests larger than 10 KB. For `lon`, a large number of requests are for manifests that are compatible with the older Docker version, hence increasing their size. We observe similar trends for `PUT`s for all the AZs (not shown in the Figures).

4.2 Registry Load and Response Time

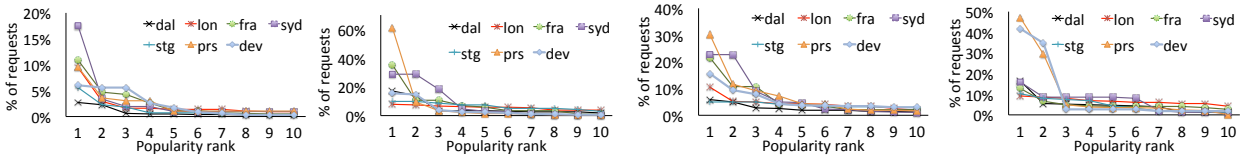
Load distribution. Figure 7 shows the CDF of received requests per minute over time. `dal` has the highest overall load and services at least 100 requests per minute more than 80% of the time. `lon` and `stg` are second and third, followed by `fra`, `syd`, `prs`, and `dev`, in descending order. This trend is consistent across the different request types (not shown). The ordering of AZs by the load yields two main observations. First, development and pre-staging registries experience low utilization. `dev`, for example, does not receive any requests 57% of the time. Second, registry load increases with its age. In our traces `dal` and `lon` have been running the longest while `fra` and `syd` have only been deployed recently.

Response time distribution. Figure 8 shows the CDFs of response time of different requests to layers and to



(a) Layer popularity. (b) Manifest popularity. (c) Repository popularity. (d) Client popularity.

Figure 10: CDF of access for layers, manifests, repositories, and clients.



(a) Layers popularity (b) Manifests popularity (c) Repos popularity (d) Clients popularity

Figure 11: Popularity of top ten layers, manifests, repositories, and clients.

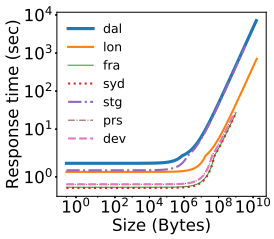


Figure 9: Dependency of response time on the layer size.

manifests. As `dal` is the highest loaded AZ, its request response times are higher compared to other AZs. More than 60% of the `GET` layer requests take more than one second to finish (Figure 8(a)). For the top 25% of requests we see a response time of ten seconds and higher. `fra`, `syd`, `prs`, and `dev` are not highly loaded, so they have the lowest latency in serving the `GET` layer requests. `PUT` and `PATCH` layer requests (Figures 8(b) and 8(c)) follow similar trends. However, `PATCH` requests are visibly slower than `GET`s and `PUT`s as they carry more data. We also analyze the dependency of response time on the layer size (see Figure 9) and find that response times remain nearly constant for layers smaller than 1 MB and then start to grow linearly.

Figure 8(d) and 8(e) show the response time distributions for `PUT` and `GET` requests to manifests, respectively. Since manifests are smaller and cached, we observe significantly smaller and more stable latencies than that of requests serving layers. One interesting observation is that `lon` has the highest response time when serving manifests (300-400 ms more than `dal`). This is because `lon` serves manifests with larger sizes compared to other AZs. This is also consistent with the results shown in Figure 6(b). For the `PUT` manifest requests we observe a more uniform trend across the AZs as the size of the new manifests is similar for all the AZs.

4.3 Popularity Analysis

In this section we study the popularity of layers, manifests, users, repositories, and clients to answer whether image accesses are skewed and produce hot-spots.

Popularity distribution. Figure 10 shows the CDF of the access rate of layers, manifests, repositories, and clients. Figure 10(a) demonstrates that there is a heavy skew in layer accesses. For example, the 1% most frequently accessed layers in `dal` account for 42% of all requests while in `syd` this increases to 59%. However, requests to the `dev` and `prs` sites are almost evenly distributed. The reason is that during testing, developers frequently push or pull images that are not available neither at registry nor at client side. We also observe that the younger AZs experience a higher skew compared to the older AZs. We believe this is due to the fact that accesses become more evenly distributed over a long period of time.

For manifest accesses (Figure 10(b)) skew is more significant than for layers. This confirms that there are indeed hot *images* which can benefit from caching. Repository accesses (Figure 10(c)) reflect this fact but show slightly less skew as manifests are contained in repositories and hence there are less repositories than manifests. The same trend holds for users under which repositories are stored (not shown in Figure 10). Furthermore, we

find that client accesses are also heavily skewed (Figure 10(d)). This means that there are few highly active clients while most of them only submit few requests. This trend is consistent across all AZs. While this does not directly affect the workload, clients can be biased towards a certain subset of images which will contribute to the access skew.

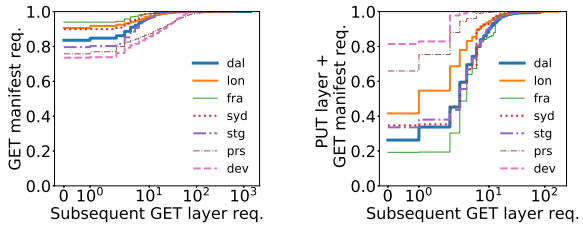
Top-10 analysis. To further understand the popularity distribution of registry items, we analyze the top 10 hottest items in each category. Figure 11(a) shows the access rates for the top 10 layers, which account for 8%–30% of all accesses depending on the registry. The most popular layer (rank 1) in all AZs absorbs 1–10% of all requests while in `syd` it absorbs 19%. The popularity rate drops rapidly as we move from most popular to tenth most popular layer. The relative amount of accesses for the top 10 layers is the lowest for `dal` as it stores the most layers and experiences the highest amount of requests.

For the top 10 manifests (Figure 11(b)), we observe that some container images are highly popular and account for as many as 40% of the requests in `fra` and `syd`, and 60% in `prs`. Note that a manifest is fetched even if the image is already cached at the client side. Hence, a manifest fetch does not necessarily mean that the corresponding layers are fetched (§4.4). Similar to Figure 10, the skew decreases for repository popularity (Figure 11(c)) and user popularity. Part of the reason for the small number of highly accessed images in younger AZ is that registry services in production are tested periodically to monitor their health and performance. For the AZs with a smaller workload (`fra` and `syd`), those test images make up three out of the top five most accessed images. We intentionally did not exclude these images from our analysis as they are typically part of the registry workload in production environments.

Figure 11(d) shows that the most popular client submits around 15% of the total requests. This excludes `prs` and `dev`, which are used by the registry development team for internal development and testing. These two AZs only have a small number of clients, and 2 clients contribute around 80% of all requests.

Overall, the detailed top-10 analysis shows that while there are a few highly popular test images, the popularity of the remaining hot items is decreasing fast and hence, overly small caches will be insufficient to effectively cache data. For example, based on these results, we estimate that a cache size of 2% of the dataset size can provide 40% and higher hit ratios.

We also analyzed the pull count of the top 10 hottest repositories on Docker Hub. We found that the most downloaded repository (*Alpine Linux*) has a pull count of more than 1 billion while the tenth most popular repository (*Ubuntu*) has a pull count of 369 million. This trend



(a) Subsequent GET layers per GET manifest.

(b) Subsequent GET layers per PUT layer + GET manifest.

Figure 12: Relationship between GET manifest and subsequent GET layer requests.

further verifies that caching can be highly effective for increasing the performance of container registries.

4.4 Request Correlation

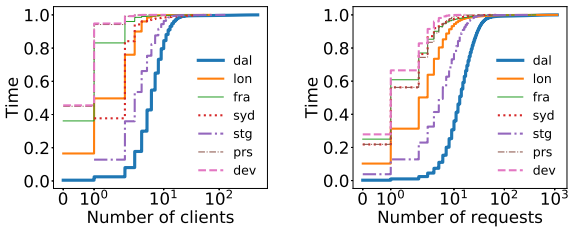
In this section we investigate whether a `GET` request for a certain manifest always results in subsequent `GET` requests to the corresponding layers. Therefore, we define a client *session* as the duration from the time a client connects until a certain threshold. We varied the threshold from 1 to 10 minutes but could not observe significant differences. However, values less than 1 minute dramatically affect the results as that is less than the typical time a client takes to pull an image. We set the session threshold to 1 minute and then count all `GET` layer requests that follow a `GET` manifest request within a session.

Figure 12(a) shows the CDF of the number of times clients issue the corresponding `GET` layer requests after retrieving a manifest. In most cases, ranging from 96% for `dev` to 73% for `fra`, `GET` manifest requests are not followed by any subsequent request. The reason is that whenever a client has already fetched an image and then pulls an image, only the manifest file is requested to check if there has been any change in the image. This shows that there is no strong correlation between `GET` manifest and layer requests.

We then focused only on `GET` manifest request that were received within the session of a `PUT` request to the same repository, from which the manifest is fetched (Figure 12(b)). This leads to a significant increase in subsequent `GET` layer requests within a session for all production and staging AZs. The manifest requests not followed by `GET` layer requests are due to the fact that clients sometimes pull the same image more than once. Overall, our analysis reveals a strong correlation between `GET` manifest and subsequent layer requests if preceding `PUT` requests are considered.

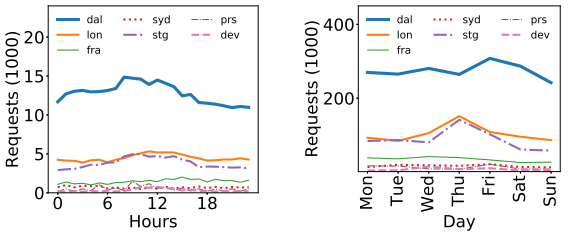
4.5 Temporal Properties

Next, we investigate whether the workload shows any temporal patterns.



(a) Client concurrency. (b) Request concurrency.

Figure 13: CDF of client and request concurrency.



(a) Hours of day (b) Days of week

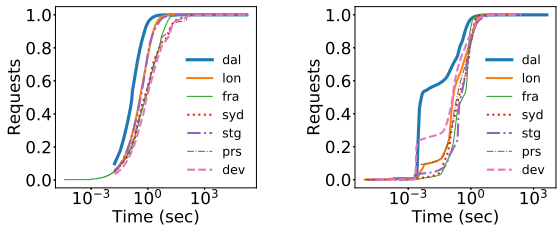
Figure 14: Average number of requests over the tracing period for each hour of the day and day of the week.

Client and request concurrency. We start with measuring how many clients and requests are *active* at a given point in time. Active clients are the clients that maintain a connection to the registry, while active requests are the requests that were received but have not yet been processed by the registry. Figures 13(a) and 13(b) show the results for clients and requests, respectively. Overall, the median number of concurrently active clients is low, ranging from 0.6 clients for `dev` to 7 clients for `dal`. However, there are peak periods during which several hundred clients are connected at the same time. We observe a similar trend for concurrently active requests.

To understand whether these peak periods follow a certain pattern, we plot the average number of requests per hour and day across all traced hours and days in Figures 14(a) and Figure 14(b). For `dal`, we observe that request numbers are decreasing during the night and over the weekend. While other AZs show a similar trend, it is less pronounced at those sites. This suggests that registry resources can be provisioned statically for hours and days. We plan to explore short-term bursts in the future.

Inter-arrival and idle times. Next, we look at request inter-arrival and idle times to study whether the registry experiences longer periods of idleness, during which less resources are required. Inter-arrival time is defined as the time between two subsequent requests. Idle time is the time during which there are no active requests.

Figure 15(a) shows the inter-arrival times. `dal`, `lon` and `stg` experience the highest request frequency with a 99th percentile of inter-arrival time around 3 s while for other AZs it is around 110 s. When looking at idle times (Figure 15(b)), we observe that idle periods are short and in most cases below 1 s. However, the amount of experienced idle periods varies significantly across AZs. Throughout the entire collection time, `dal` saw only approximately 0.1 million idle periods while `lon` experienced more than 1.5 million. While some AZs experience a large amount of idle periods, their duration is short and hence, they are hard to exploit with traditional resource provisioning approaches.



(a) Request inter-arrival time. (b) Request idle time.

Figure 15: CDF of request inter-arrival and idle times

4.6 Analysis Summary

We summarize our analysis in seven observations:

1. `GET` requests are dominant in all registries and more than half of the requests are for layers, opening an opportunity for effective layer caching and prefetching at the registry.
2. 65% and 80% of all layers are smaller than 1 MB and 10 MB, respectively, making individual layers suitable for caching.
3. The registry load is affected by the registry's intended use case and the age of the registry. Younger, non-production registries experience lower loads compared to longer running, production systems. This should be considered when provisioning resources for an AZ to save cost and use existing resources more efficiently.
4. Response times correlate with registry load and hence also depend on the age (younger registries experience less load) and the use case of the registry.
5. Registry accesses to layers, manifests, repositories, and by users are heavily skewed. Few extremely hot images are accessed frequently but the popularity drops rapidly. Therefore, caching techniques are feasible but cache sizes should be selected carefully.
6. There is a strong correlation between PUT requests and subsequent GET manifest and GET layer requests. The registry can leverage this pattern to prefetch the layers from the backend object store to the cache, significantly reducing pull latencies for the client. This correlation exists for both popular as well as non-

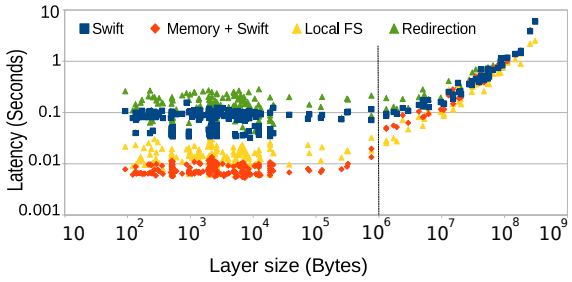


Figure 16: Effect of various backend storage technologies on registry performance.

popular images.

7. While there are weak declines in request rates during weekends, we did not find pronounced repeated spikes that can be used to improve resource provisioning.

5 Registry Design Improvements

In this section, we use the observations from §4 to design two improvements to the container registry: (i) a multi-layer cache for popular layers; and (ii) a tracker for newly pushed layers, which enables prefetching of the newest layers from the backend object store. We evaluate our design using our trace replayer.

5.1 Implementation

We implemented the trace replayer and its performance analysis mode in Python. This mode allows us to study the effect of different storage technologies on response latency. We use Bottle [2] for routing requests between the master and clients and the dxf library [6] for storing and retrieving data in/from the registry. For caching and prefetching, we implemented two separate modules. To implement the in-memory layer cache, we modified the Swift storage driver for the registry (about 200 LoC modified/added). The modified driver stores the small sized layers in memory and uses Swift for larger layers.

5.2 Performance Analysis

The registry is launched on a 32 core machine with 64 GB of main memory and 512 GB of SSD storage, and the Swift object store runs on a separate set of nodes of similar configuration. The trace replayer is started on an additional six nodes (one master and five clients). We made sure that the trace replayer or the object store are never the bottleneck during this analysis. All nodes are connected via 10 Gbps network links. To drive the analysis, the trace replayer is used to replay 10,000 requests from the `dal` trace (August 1st, 2017 starting at 12 am).

We compare four different backends: 1) Swift; 2) memory for layers smaller than 1 MB and Swift for rest of the layers (Memory + Swift); 3) local file system with SSD (Local FS); and 4) Redirection, i.e. the registry replies back with the link to the layer in Swift and the client then fetches the layer directly from Swift. Swift, Local FS, and Redirection are by default supported by the Docker registry.

Figure 16 shows the latency vs. layer size for all backends. We observe that, for small sized layers (i.e. layers less than 1 MB), the response time is the lowest (0.008 s on average) for Memory + Swift. This is followed by Local FS, which yields an average response time of around 0.013 s and Swift with an average response time of 0.07 s. Redirection performs the worst with average response time of 0.11 s.

For large size objects, we observe that Memory + Swift and Local FS are comparable and both beat Swift and Redirection. Moreover, for layers slightly larger than 1 MB, Swift outperforms Redirection. However, for very large layers, Swift and Redirection perform similarly, with average response latencies of 0.63 s and 0.59 s, respectively.

The results highlight the advantage of having a fast backend storage system for the registry, and demonstrate the opportunity for caching to significantly improve registry performance.

5.3 Two-level Cache

In designing our cache, we chose to exploit the high capacity memory as well as SSDs that are present in modern server machines. We also observed that a small fraction of layers are too large to justify the use of memory to cache them. Consequently, we design a two-level cache consisting of main memory (for smaller layers) and SSDs (for larger layers). We do not have to deal with possible cache invalidation as layers are content addressable and any change in a layer also changes its digest. This results in a "new" layer for caching while the older version of the layer is no longer accessed and eventually gets evicted from the cache.

Hit ratio analysis. We perform a simulation-based evaluation of our two-level cache for the registry servers. For these experiments, we mimic the IBM registry server setup. We simulate the same number of servers as there are in each AZ and for each server, we add memory and SSD caches. The registry servers do not share the cache as the Docker registry implementation is non-distributed. However, the setup can be scaled by adding more registry servers behind the Nginx load balancer.

We use the LRU caching policy for both the memory and the SSD level cache. We select cache sizes of 2%, 4%, 6%, 8%, and 10% of the data ingress for each AZ. The data ingress of an AZ is the amount of *new* data stored in that AZ during the 75 days period during which we collected the traces. For the SSD level cache sizes, we select 10×, 15×, and 20× the size of the memory cache. Any object evicted from the memory cache goes first to

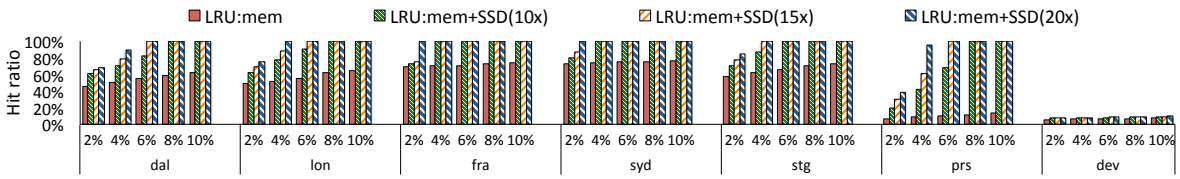


Figure 17: Hit ratio of LRU caching policy for both the memory and the SSD level cache.

the SSD cache before it is completely evicted. We store layers smaller than 100 MB in the memory level cache, while larger layers are stored in the SSD level cache. For our analysis, we iterate over the traces to warm the cache and start calculating the hit and miss ratios upon observing the first eviction from the cache. Given our long trace period, the first eviction happens early relative to the time it takes to replay all traces.

Figure 17 shows the hit ratios. We see that for the production and staging AZs, adding even a single level of LRU-based memory cache yields a hit ratio of 40% for `dal` with a cache size of 2% of ingress data and as high as 78% for `fra` and `syd` with a cache size of 10% of ingress data.

Increasing the cache size increases the hit ratio, until it reaches the max of 78%. This is because we only put layers less than 100 MB in the memory cache. However, when we enable the second level cache, we achieve a combined hit ratio of 100% with 6% cache size for `dal` and 4% cache size for the other four AZs. We observe different results for the `prs` and `dev` AZs. As these two traces represent testing interactions by the registry development team, we do not see any advantage of using the cache in this case.

5.4 Prefetching Layers

Our second design improvement is to enable prefetching of layers from the backend object store by predicting what layers are most likely to be requested. Therefore, we use our observations of the push-pull relationship established in §4.4 to predict what layers to prefetch as shown in Algorithm 1.

In §4.4, we observed that the incoming PUT requests determine which layers will be prefetched when the registry receives a subsequent GET manifest request. When a PUT is received, the repository and the layer specified in the request will be added to a look up table that includes the request arrival time and the client address. When a GET manifest request is received from a client within a certain threshold LM_{thresh}, the host checks if the look up table contains the repository specified in the request. If it is a hit and the client's address is not present in the table, then the address of the client is added to the table and the layer is prefetched from the backend object store. Note that both the amount of time that the entries

Algorithm 1: Layers Prefetching Algorithm.

Input: LM_{thresh}: Threshold for duration between PUT layer and subsequent GET manifest requests, ML_{thresh}: Threshold for duration to keep prefetched layer.

```
1  while true do
2      r ← request received
3      if r = PUTlayer then
           /* Create new entry for layer                          */
4          RepoMap[r.repo] ← NewEntry(r.client, r.layer)
5          RepoMap[r.repo] ← set LM_timer
           /* When LM_timer > LM_thresh, entry is evicted         */
6      else if r = GETmanifest then
7          if r.client not in RepoMap[r.repo] for r.layer then
8              RepoMap[r.repo] ← add r.client
9              PrefetchedLayers ← prefetch r.layer
10             PrefetchedLayers[r.layer] ← set ML_timer
               /* When ML_timer > ML_thresh, layer is evicted     */
11             prefetch++
12     else if r = GETlayer then
13         if r.layer in PrefetchedLayers then
14             serve from PrefetchedLayers[r.layer]
15             prefetch_hit++
16         else
17             serve from object store
```

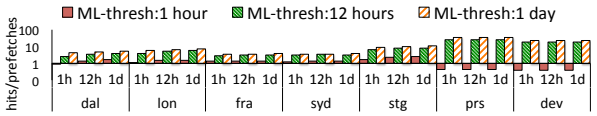


Figure 18: Hits/prefetch ratio.

remain in the look up table and how long the layers are cached at the registry side, defined by ML_{thresh}, are configurable.

Hits/prefetch analysis. We tested our algorithm using different values for retaining look up table entries, LM_{thresh}, and retaining prefetched layers, ML_{thresh}. We use values of 1 hour, 12 hours, and 1 day for each of the threshold parameters. Figure 18 shows the results. Single bars represent ML_{thresh} values while groups of bars are assigned to LM_{thresh} values.

On one hand, we find that increasing ML_{thresh} can significantly increase the hit/prefetch ratio. On the other hand, increasing the retention threshold for the *look up* table entries only marginally increases the hit ratio. This is because the longer an entry persists in the table, the fewer prefetches it serves as the record of clients added to the table increases. We also find that the maximum

amount of memory used by dal, lon, fra, syd, prs, and dev is 10 GB, 1.7 GB, 0.5 GB, 1 GB, 2 MB, and 69 MB respectively. We note that for both prs and dev the maximum amount of memory is low because they experience less activity and therefore contain less PUT requests compared to other cases.

Our analysis shows that it is possible to improve registry performance by adding an appropriate sized cache. For small layers, a cache can improve response latencies by an order of magnitude and achieve hit ratios above 90%. We also show that it is possible to predict the GET layer requests under certain scenario to facilitate prefetching.

6 Related Work

To put our study in context we start with describing related research on Docker containers, Docker registry, workload analysis, and data caching.

Docker containers. Improving performance of container storage has recently attracted attention from both industry and academia. DRR [34] improves common copy-on-write performance targeting a dense container-intensive workload. Tarasov et al. [45] study the impact of the storage driver choice on the performance of Docker containers for different workloads running inside the containers. Contrary to this work, we focus on the registry side of a container workload.

Docker registry. Other works have looked at optimizing image retrieval from a registry side [37, 42]. Slacker [37] speeds up the container startup time by utilizing lazy cloning and lazy propagation. Images are fetched from a shared NFS store and only the minimal amount of data needed to start the container is retrieved initially. Additional data is fetched on demand. However, this design tightens the integration between the registry and the Docker client as clients now need to be connected to the registry at all times (via NFS) in case additional image data is required. Contrariwise, our study focuses on the current state-of-the-art Docker deployment in which the registry is an independent instance and completely decoupled from the clients.

CoMICon [42] proposes a system for cooperative management of Docker images among a set of nodes using peer-to-peer (P2P) protocol. In its essence, CoMICon attempts to fetch a missing layer from a node in close proximity before asking a remote registry for it. Our work is orthogonal to this approach as it analyzes a registry production workload. The results of our analysis and the collected traces can also be used to evaluate new registry designs such as CoMICon.

To the best of our knowledge, similar to IBM Cloud, most public registries [5, 8, 19] use the open-source implementation of the Docker registry [4]. Our findings are applicable to all such registry deployments.

Workload analysis studies. A number of works [27, 38] have studied web service workloads to better understand how complex distributed systems behave at scale. Similar studies exist [31, 30] which focus on storage and file system workloads to understand access patterns and locate performance bottlenecks. No prior work has explored the emerging container workloads in depth.

Slacker [37] also includes the HelloBench [10] benchmark to analyze push/pull performance of images. However, Slacker looks at client-side performance while our analysis is focused at registry side. Our work takes a first step in performing a comprehensive and large-scale study on real-world Docker container registries.

Caching and prefetching. Caching and prefetching have long been effective techniques to improve system performance. For example, modern datacenters use distributed memory cache servers [15, 21, 32, 33] to improve database query performance by caching the query results. A large body of research [28, 29, 36, 40, 43, 46, 47] studied the effects of combining caching and prefetching. In our work we demonstrate that the addition of caches significantly improves container registry's performance, while layer prefetching reduces the pull latency for large and less popular images.

7 Conclusion

Docker registry platform plays a critical role in providing containerized services. However, heretofore, the workload characteristics of production registry deployments have remained unknown. In this paper, we presented the first characterization of such a workload. We collected and analyzed large-scale trace data from five geographically distributed datacenters housing production Docker registries. The traces span 38 million requests over a period of 75 days, resulting in 181.3 TB of traces.

In our workload analysis we answer pertinent questions about the registry workload and provide insights to improve the performance and usage of Docker registries. Based on our findings, we proposed effective caching and prefetching strategies which exploit registry-specific workload characteristics to significantly improve performance. Finally, we have open-sourced our traces and also provide a trace replayer, which can be used to serve as a solid basis for new research and studies on container registries and container-based virtualization.

Acknowledgments. We thank our shepherd, Pramod Bhatotia, and reviewers for their feedback. We would also like to thank Jack Baines, Stuart Hayton, James Hart, IBM Cloud container services team, and James Davis. This work is sponsored in part by IBM, and by the NSF under the grants: CNS-1565314, CNS-1405697, and CNS-1615411.

References

[1] Artifactory. https://www.jfrog.com/confluence/display/RTF/Docker+Registry.

[2] Bottle: Python Web Framework. https://github.com/bottlepy/bottle.

[3] CoreOS. https://coreos.com/.

[4] Docker-Registry. https://github.com/docker/docker-registry.

[5] Dockerhub. https://hub.docker.com.

[6] dxf. https://github.com/davedoesdev/dxf.

[7] ElastiSearch. https://github.com/elastic/elasticsearch.

[8] Google Container Registry. https://cloud.google.com/container-registry/.

[9] gRPC. https://grpc.io/.

[10] HelloBench. https://github.com/Tintri/hello-bench.

[11] IBM Cloud. https://www.ibm.com/cloud-computing/.

[12] IBM Cloud Container Registry. https://console.bluemix.net/docs/services/Registry/index.html.

[13] Kibana. https://github.com/elastic/kibana.

[14] Logstash. https://github.com/elastic/logstash.

[15] Memcached. https://memcached.org/.

[16] Microservices and Docker containers. goo.gl/UrVPdU.

[17] Microservices Architecture, Containers and Docker. goo.gl/jsQ1sL.

[18] OpenStack Swift. https://docs.openstack.org/swift/.

[19] Project Harbor. https://github.com/vmware/harbor.

[20] Quay.io. https://quay.io/.

[21] Redis. https://redis.io/.

[22] What is Docker. https://www.docker.com/what-docker.

[23] 451 RESEARCH. Application Containers Will Be a $2.7Bn Market by 2020. http://bit.ly/2uryjDI.

[24] AMARAL, M., POLO, J., CARRERA, D., MOHOMED, I., UNUVAR, M., AND STEINDER, M. Performance evaluation of microservices architectures using containers. In *IEEE NCA* (2015).

[25] ANWAR, A., CHENG, Y., GUPTA, A., AND BUTT, A. R. Taming the cloud object storage with mos. In *ACM PDSW* (2015).

[26] ANWAR, A., CHENG, Y., GUPTA, A., AND BUTT, A. R. Mos: Workload-aware elasticity for cloud object stores. In *ACM HPDC* (2016).

[27] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS* (2012).

[28] BUTT, A. R., GNIADY, C., AND HU, Y. C. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *ACM SIGMETRICS* (2005).

[29] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst. 14*, 4 (Nov. 1996), 311–343.

[30] CHEN, M., HILDEBRAND, D., KUENNING, G., SHANKARANARAYANA, S., SINGH, B., AND ZADOK, E. Newer is sometimes better: An evaluation of nfsv4.1. In *ACM SIGMETRICS* (2015).

[31] CHEN, Y., SRINIVASAN, K., GOODSON, G., AND KATZ, R. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *ACM SOSP* (2011).

[32] CHENG, Y., GUPTA, A., AND BUTT, A. R. An in-memory object caching framework with adaptive load balancing. In *ACM EuroSys* (2015).

[33] CHENG, Y., GUPTA, A., POVZNER, A., AND BUTT, A. R. High performance in-memory caching through flexible fine-grained services. In *ACM SOCC* (2013).

[34] DELL EMC. Improving Copy-on-Write Performance in Container Storage Drivers. https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf.

[35] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IEEE ISPASS* (2015).

[36] GNIADY, C., BUTT, A. R., AND HU, Y. C. Program-counter-based pattern classification in buffer caching. In *USENIX OSDI* (2004).

[37] HARTER, T., SALMON, B., LIU, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Slacker: Fast Distribution with Lazy Docker Containers. In *USENIX FAST* (2016).

[38] HUANG, Q., BIRMAN, K., VAN RENESSE, R., LLOYD, W., KUMAR, S., AND LI, H. C. An analysis of facebook photo caching. In *ACM SOSP* (2013).

[39] KANGJIN, W., YONG, Y., YING, L., HANMEI, L., AND LIN, M. Fid: A faster image distribution system for docker platform. In *IEEE AMLCS* (2017).

[40] LI, M., VARKI, E., BHATIA, S., AND MERCHANT, A. Tap: Table-based prefetching for storage caches. In *USENIX FAST* (2008).

[41] MENAGE, P. B. Adding Generic Process Containers to the Linux Kernel. In *Linux Symposium* (2007).

[42] NATHAN, S., GHOSH, R., MUKHERJEE, T., AND NARAYANAN, K. CoMICon: A Co-Operative Management System for Docker Container Images. In *IEEE IC2E* (2017).

[43] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *ACM SOSP* (1995).

[44] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *ACM EuroSys* (2007).

[45] TARASOV, V., RUPPRECHT, L., SKOURTIS, D., WARKE, A., HILDEBRAND, D., MOHAMED, M., MANDAGERE, N., LI, W., RANGASWAMI, R., AND ZHAO, M. In search of the ideal storage configuration for Docker containers. In *IEEE AMLCS* (2017).

[46] WIEL, S. P. V., AND LILJA, D. J. When caches aren't enough: data prefetching techniques. *Computer 30*, 7 (Jul 1997), 23–30.

[47] ZHANG, Z., KULKARNI, A., MA, X., AND ZHOU, Y. Memory resource allocation for file system prefetching: From a supply chain management perspective. In *ACM EuroSys* (2009).