# 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.

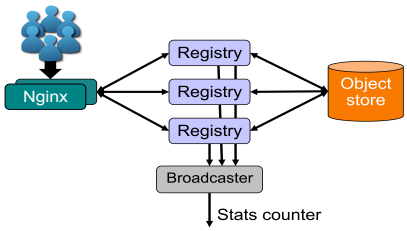



**Figure 1:** IBM Cloud Registry architecture. Nginx receives users requests and forwards them to registry servers. Registry servers fetch data from the backend object store and reply back.

**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.

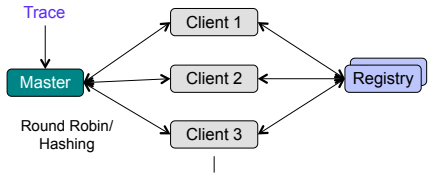



**Figure 3:** Trace replayer. Master parses the trace and forwards request to one of the clients either in round robin or applying hash to the `http.request.remoteaddr` field in the trace.

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.

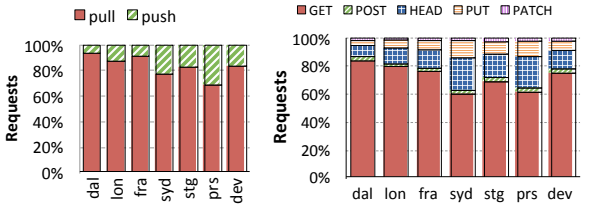
The requests are issued by the trace replayer in two modes: 1) "as fast as possible", and 2) "as is", to account for the timestamp of each request. The master side of the trace replayer is multithreaded and each client's progress is tracked in a separate thread. Once all clients finish their jobs, aggregated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

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).