Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

The Design and Experimental Use of CReB, a Container Registry Benchmark

Author: Petar Galic (2571287)

1st supervisor: dr. Alexandru Iosup
daily supervisor: ir. Erwin van Eyk

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

December 14, 2020

Abstract

Cloud technology is the key to the digital growth of a society. Government organizations, startups, and legacy companies are utilizing cloud infrastructure to deliver the products and services that people depend on. Cloud technology adopted containers, the lightweight processes that enable efficient sharing of hardware and system-level resources across heterogeneous infrastructure. To facilitate containerization, there is a need for a service that enables distribution and storage of container blueprints – the container images. The container image registry is a service that offers storage and sharing of container images. Quick delivery of container images to the source of deployment is critical for minimizing deployment downtime and for the general performance. Measuring the performance of container registries is necessary to identify performance bottlenecks and evaluate performance optimizations.

Open-source performance measuring tools exist, though they are tied to the container registry academia. This makes them unusable for the evaluation of the industry state-of-the-art. There is a lack of extensive performance benchmarks of the production-grade registries.

To fill this gap, we present the design of a Container Registry Benchmark, CReB. It improves the existing state-of-the-art in registry performance evaluation, and adds new features that enable experiments on the real-world, production-grade container registries. CReB allows fine-grained experiment configuration with real-world and synthetic workload, reporting performance metrics relevant for container registry systems: latency, throughput, and cost.

We implement CReB and evaluate performance of the set of the most popular container registries. In this work we simulate a hybrid cloud environment. We evaluate registries from a fixed source external to the evaluated registries.

We present experiment results for three large experiments: two real-world workload experiments using IBM production traces, and one long running, synthetic

workload experiment. We identify performance differences between three large public cloud registries within geographically similar regions, and present the performance degradation patterns observed over time.

We open-source CReB, along with data artifacts to facilitate further research and collaboration in the field of container registry evaluation.

Preface

I am very happy to present this document. It is a culmination of 7 months of work, starting from a basic idea to a full blown project. I am thankful for this project as it kept me sane (or insane) and busy during the coronavirus crisis. There were a lot of crisis points where I doubted the relevance and the contribution value of the project, but as the time went on I realized I am just scraping the surface of an interesting and developing field.

I would like to thank my 1st supervisor, Alexandru Iosup for his motivating comments and never ending ideas that really pushed me to always consider how to improve every step of the process. I would also like to thank my daily supervisor Erwin van Eyk for his eagerness and support even when I did not believe in myself and the project. He was involved in the project from day one and helped me find a road to excellence. Also their effort in highlighting every missing article in my text without getting annoyed is much appreciated! I hope that the collaboration with them will not stop with the end of this project.

I would also like to thank my parents and my sister for their support throughout my education. Without their support I would not be where I am today.

And lastly, I would like to thank my girlfriend Alexia. Your patience and support from day to day helped me more than you can imagine.

Honorable mention: my cat, Mr. Noodles. Thanks for sitting on my laptop, jumping in the video calls, and cuddling, mostly when hungry.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

As the cloud platforms are delivering better performance, more organizations are offloading their critical processes to the cloud. This trend is happening due to many reasons, such as ease of deployment and delegation of the infrastructure availability concerns to the cloud platform provider.

Cloud platforms and the services built on top of them are running inside containers [1]. Containers at its core are processes running on the host system in an isolated environment. Containers do not need an additional layer of virtualization as virtual machines. Container process is run from the container image, a bundle of application code and related dependencies necessary for application to run. Container startup time consists of fetching missing pieces of container image, preparing the isolated environment and running the process inside the container.

Containers are at the center of the cloud native, distributed architecture because of their lightweight nature, portability, and performance isolation (1). The performance of containers, and the services for container management and image distribution have become critical to the performance of the cloud technologies in general.

The container image registry, in short *container registry*, is the service for storage and distribution of container images. It allows users to upload and download images and categorize them into repositories by image names and tags. It is a critical component in the DevOps process[2] of every containerized system, and it is also a key component in the development process (2). Containers have the same behaviour on all OS and hardware, therefore there is less uncertainty around misbehavior in production environment.

[1] google.com/containers

[2] DevOps is a set of practices that combines development process and IT operations. At its core it is the process of building software code into artifacts and deploying the artifacts in a way that ensures high software quality.

The performance of a container registry has consequences on different functionalities of the service that it interacts with. Uber developed Kraken, an image distribution service to handle workloads such as 1,000 concurrent different image-pulls, or 10,000 same image pulls for updating a Kubernetes cluster in the private cloud (3). Another example is serverless computing, where the platform prepares a container for user-defined code to run inside. In this use case container startup time needs to be minimized such that the user does not observe any meaningful latency compared to running containers themselves. Container registry is the components that fulfills the function registry functionality of delivering images and other artifacts necessary for a serverless function to run(4). In this case the interaction with the container registry is a standard functionality of the serverless platform, and therefore its performance directly influences every day performance of the serverless platform.

We can see from these examples that evaluating performance of container registry can provide useful insight for a containerized service. It is also important for identifying bottlenecks in the system. In the case of Kraken, a high number of concurrent requests exposed the registry throughput as the bottlenecks. This showcases the need for a registry benchmark.

We haven't been able to identify any well-known container registry benchmark instruments present today, despite the industry's interest. There does not exist any comprehensive performance analysis, as well. We have identified previous work related to the more general topic of cloud benchmarking (5, 6), and cloud storage benchmarking (7, 8, 9, 10). Though useful to highlight the general concerns in regards to benchmarking, these benchmarks do not capture well the unique characteristics of container registries.

In this thesis we present CReB: a new Container Registry Benchmark that evaluates and reports most relevant user-level metrics: latency, throughput, and cost. These metrics are reported for the key user interactions with a container registry. Along with the benchmark instrument, we present the analysis of performance of well-known container registries.

1.1 Context

The cloud revolution is moving at a fast pace. Companies are delegating increasingly more infrastructure layers to cloud providers, as the cloud service costs are dropping and performance is increasing (11). Companies are able to significantly reduce their DevOps costs and quickly move to the market. This trend is observed for large organizations such

as Netflix, Twitch, and Facebook spend tens to hundreds of millions of dollars on AWS services (12), and emerging startups such as Stripe(13) or Slack(14).

How is it possible that so many companies can seamlessly switch, or start working from scratch with the public cloud platform? Transitioning and change for large companies used to take years, and now the tech companies are moving as fast as the cloud platform that is servicing it. The answer is, among others, the container technology.

The container is a lightweight virtualization technique that bundles code and related dependencies together such that the software can be ran on any kind of infrastructure. There is no need for operating system virtualization, as containers are isolated processes that run natively on Linux. The container process is run using a container image, which is a list of files (layers) that contains code and dependencies necessary to run the containerized process. We dive deeper in container technology in Chapter 2.

As the cloud environment usually implies heterogeneous hardware and operating systems, containers grew in popularity due to their ability to run on any operating system and hardware. Organizations across the world started containerizing their applications, and they needed a service to store images, and distribute them across the machines and makes in an automated and standardized way.

Containers are often compared to virtual machines, and both have advantages and disadvantages. Containers have a performance advantage over virtual machines due to the ability to run directly on the host operating system, without the need for a hypervisor process in between [1]. Containers are however considered more insecure than virtual machines. In case of the attacker gaining access to the containerized process, there are number of security concerns that can result in privilege escalation and attacker gaining access to the container host system (15).

This service is *container image registry*. Docker Hub was the first registry that gained popularity due to it being integrated in Docker runtime flow, and retains the lead today (16). We observe the trend of platforms offering their own container registries to centralize all software artifacts in the same location, to reduce context switching. Organizations can also choose to run a private on-premise registry for the enhanced security. Today, half of all containers are managed by a cloud provider and third-party registry (17).

Container performance optimizations are an interesting research topic because of the impact it causes on the general performance of cloud applications, and especially latency sensitive ones. As research shows, pulling the container image from a container registry takes 76% of the container startup time (1). Further research in academia has been done in

[1] wikipedia.org/hypervisor

optimizing container registries to minimize pull times (18, 19). Organizations such as Uber (3), Alibaba (20), and Facebook (21) are also innovating to improve artifact distribution times.

1.2 Problem Statement

The emerging area of container registries is missing a *large-scale performance evaluation of existing registries from large cloud providers*, which can provide insight into performance and cost differences between these providers. We have also identified that the area of container registry benchmarking lacks a *comprehensive instrument* able to evaluate and report experiment metrics. Experiments should be based on parameters that we later refer to as *key aspects*.

The current performance evaluations of container registries (1, 18, 22) lack comparison to real-world registry services in a realistic environment. These papers succeeded in proposing optimizations in regards to improving cache hit ratios and subsequently improving latencies, but we do not know if these optimizations exist in production-grade registry services.

A large evaluation of production-grade, hosted cloud registry services is valuable both to academia and industry. It highlights how big of a priority do registry services put on raw performance, and if it is inversely proportional to the number of features they advertise in a container registry (such as geo-replication, vulnerability scanning, storage of other OCI artifacts, etc.). Key aspects of such a container registry benchmark are the following: *workloads*, *metrics*, and *System-under-Test sets*.

Along with the evaluation we hypothesize the benchmark instrument will give organizations interested in improving their image distribution times a way to compare various deployments and configurations and make decisions accordingly.

1.3 Research Questions

To create a relevant benchmark and to provide an insightful performance evaluation of cloud registries, we decomposed the problem into the following three research questions:

RQ1. What are the relevant key aspects of a container registry benchmark?

To build a relevant benchmark instrument it is necessary to research the current registry performance evaluations and registry systems to identify performance and

cost metrics that are relevant for cloud registry users. It is also necessary to identify workloads, both synthetic and real-world, that test registry behaviors and various optimizations. Lastly, it is important to identify user-preferred platforms to incorporate in the study. No such analysis exists in the world today, therefore a systematic survey that will encompass all of the aforementioned topics needs to be performed.

RQ2. How to design a registry benchmark that supports relevant key aspects?

Using the aspect types identified in the RQ1, a benchmark should be designed, according to the state-of-the-art approach defined in (5). For the benchmark instrument to support extensive experiment configurations, it needs to run both real-world and synthetic workloads, support configuration of at least 15 System-under-Test's (registries) at the time. The benchmark instrument needs to be designed taking into account requirements from the envisaged stakeholders.

RQ3. What are the cost and performance differences between production-grade, hosted registry platforms?

To answer this research question in a sufficient way a large scale performance benchmark needs to be performed. The benchmark should consist of at least three different experiments, evaluating a set of at least 15 registries and reporting at least two metrics in the process. The experiments should test at least push and pull operations for manifests and layers. The benchmark should consist of both real-world and synthetic workloads.

1.4 Approach

RQ1. To answer this research question we launch *an extensive literature and system survey* in the area of container registry systems. This will help identify current features, performance optimizations – and more importantly how they are measured and what are the workloads used. We need to analyze real-world registry systems to grasp the performance bottlenecks in real production systems, and what metrics are relevant for end users.

RQ2. We perform a requirements analysis to identify stakeholders and their concerns regarding the new benchmark instrument. From there we formalize the design of the instrument architecture. Then we perform the evaluation of architecture according to the user requirements and general benchmarking requirements. Finally we build a prototype according to the design.

RQ3. To answer this question we use the instrument that is the product of answering RQ2. We vary configuration parameters defined for the benchmark instrument to create specific experiment configurations. We perform the experiments on a set of registries using synthetic and real-world workloads. In the experiment we simulate the real performance of a number of clients sending requests to a registry.

1.5 Main Contributions

In this thesis we provide the following *Technical*, *Conceptual*, and *Experimental* contributions, mapped to the research questions that it answers:

1. *(Conceptual, RQ1)* Multivocal literature and system survey on container registries proposed in research and current industry state-of-the-art (Chapter 3).
2. *(Conceptual, RQ1)* Analysis of the critical issues with current state-of-the-art instrument and comparative analysis with the improved version (Chapter 4).
3. *(Conceptual, RQ2)* Requirements analysis and design of CReB, a container registry benchmark instrument (Chapter 4).
4. *(Technical, RQ2)* Container registry benchmark instrument CReB, open-sourced (Chapter 4).
5. *(Experimental, RQ3)* Design and deployment of experiments using CReB, on the DAS cluster (Chapter 5).
6. *(Experimental, RQ3)* Quantitative results and analysis of the large scale container registry evaluation using CReB (Chapter 5).

2

Background

In this section we dive in the technology that is the basic building block of today's cloud services (23). At its core containers are a virtualization technology, and they are often evaluated in comparison to another virtualization technology, virtual machines. The popularity of containers has grown with cloud popularity in parallel. Containers raised to prominence when it became apparent that the overhead of managing virtual machines is too big for many cloud services. The rise of Agile and service oriented architecture in software contributed to this trend, since organizations preferred technologies that enable rapid application development, and quick deployment cycles (24).

However today we see that new use cases emerge due to the advantages the containers bring, especially in the cloud native ecosystem. Container orchestration systems like Kubernetes [1] and Docker Swarm [2] have enabled a further abstraction from containers into nodes and clusters, which in turn enabled organizations to build robust, highly available and fault-tolerant distributed systems.

2.1 Containers

What is a container? Container is a lightweight virtualization that allows applications to be quickly ran regardless of the underlying OS and hardware. This is enabled by packaging the application code together with all the configuration, dependencies and packages needed for the application to run. By using Linux kernel isolation features application has its own resource and network space, and delivers almost bare-metal performance due to no need for an OS virtualization.

[1] kubernetes.io
[2] docker.com/swarm

A modern container technology has its roots in the Linux container technology (LXC) (25), which was released 12 years ago, on August 6th 2008. The concept of containers has been around for even longer, and can be traced to the introduction of `chroot` command [1]in Unix around 1982. Before introduction of LXC, there were other similar concepts implemented in Solaris(26) in 2004, and FreeBSD(27) in 2000. In 2013, due to the introduction of namespaces and cgroup support in Linux kernel, Docker emerged as the new solution for the rapid Linux application deployment inside containers.

To understand the container ecosystem we provide the analysis of the Docker ecosystem. Docker is a set of services for building, running, and deploying containers, and since it appeared in 2013 it became the industry standard. During years other container runtimes emerged but according to the 2019 Sysdig report (28) 79% of the users use Docker runtime for containers.

In Section 2.1.1 we explain what is running under the hood of Docker containers, and in Section 2.3 we explain how Docker services interact to manage the container lifecycle. Finally, in Section 2.2 we go over the other container runtimes and formats present in the industry.

2.1.1 Container Internals: an Example of Docker

Docker containers are based on features in Linux kernel that provide isolation property necessary for containers. The main kernel features that enable Docker containers are *namespaces* and *cgroups*.

Namespaces are the Linux kernel feature used to provide isolation and virtualization of system level resources. There are many namespaces that each isolate a different system resource. For example, a PID namespace is used to isolate a subtree in the Linux process tree and give it a PID 1 in this namespace, which creates a separate process tree where other processes that are in this process tree are isolated and do not see processes outside it. Another example of a namespace is filesystem namespace, which uses the same approach to create an isolated are inside a filesystem tree. Same approach allows for users outside of the isolated environment to be mapped to users with different privilege level inside the isolated area. These and other namespaces are used to provide a fine-grained isolation per resource which is used for containers.

Cgroups are the Linux kernel feature that allows for isolation of hardware resources. Cgroups are used to provide containers with processing power, so the container itself sees

[1]wikipedia.org/wiki/chroot

only these resources available and cannot consume more resources than are made available by the Linux cgroup.

These features enable the container to be independent from the host system and have its own set of isolated resources that it can use without impacting the rest of the system. Now we will look at the features that are specific to Docker containers.

The first feature that distincts Docker container from the likes of LXC and prior technologies are *stackable image layers*.

Docker container is run from a Docker image, which is a collection of layers (blobs) that contain all packages, configurations, dependencies necessary for the application to run, together with the application code. Docker image is built from a Dockerfile, which is an image configuration file that specifies the aforementioned artifacts for packaging. Every command in the Dockerfile results in a layer created, only if there are changes (*Copy on Write* behaviour). The layers are connected with a manifest, a file that contains pointers to all layers that make up an image, along with metadata (such as image name and tag), and configuration file pointers. Image tag corresponds to the image version.

When a Docker container is run from a Docker image, a writable layer is created on top of the read-only image layers which is used to make changes to container. Since image layers are immutable, different Docker images that share basic OS layers or language runtime dependencies often contain same layers, therefore the layer storage is reduced by sharing layers among Docker images.

The image layer sharing is made possible by utilizing *Another Union Filesystem* – a Docker implementation of a Union File System. Union Filesystem is a filesystem used for displaying different file directories as they are a single directory. Each container image layer corresponds to a folder in the filesystem which is also a branch of AUFS.

The final piece of the Docker puzzle is the networking. To facilitate the communication between containers and to provide network access to outside Docker creates a virtual network bridge. The containers have their own network interfaces inside network namespaces. These interfaces are mapped to virtual network interfaces for each container that are connected to a virtual network bridge which connects to a host network interface.

These are the key features and technologies behind Docker containers. As we can see Docker utilizes existing features of Linux kernel to provide a lightweight virtualization that we know as containers.

2.2 Container Runtime

The interaction with kernel to run containers is performed by the *container runtime*.

We distinguish between low-level container runtimes and high-level container runtimes (29). Low-level runtimes handle the core of running containers – using necessary kernel features to run containers. On the other hand high-level container runtimes are user-facing, offering API's and client applications. High-level container runtimes need to either have the low-level container runtime capabilities, or delegate running the containers to a low-level container runtime.

Docker was initially built on `lxc` runtime, which we already mentioned in the introduction of this chapter as the initial container support introduced in Linux. Docker switched to `libcontainer` and later to `runc` container runtime. `runc` was developed as a core part of Docker, however it was spinned off and donated to the open-source community (see Section 2.5). It is the most widely used low-level container runtime. Currently Docker is using a high-level container runtime `containerd` also developed as part of Docker initially and then spinned off. `containerd` provides API and client application but uses `runc` for Linux kernel interactions (30). It is important to note that Docker uses customized versions of `runc` and `containerd` which are enhanced with features like volume and network management for containers.

However, other container runtimes have been developed at the same time. Example of a low-level container runtime is `lmctfy`[1] developed by Google. It was abandoned in favour of `containerlib` and `runc`.

Examples of a high-level container runtime are `rkt` and `cri-o`. `rkt`[2] is a container runtime developed by CoreOS that performs both high-level and low-level container runtime tasks. It builds, manages and fetches container images using cli commands. It used a deprecated Application Container format however it is in the process of switching to the standardized OCI image format [3]

On the other hand, `cri-o`[4] is the high-level container runtime developed by Red Hat. It is specifically developed for Kubernetes as a lightweight alternative to Docker runtime. It contains a low-level container runtime that works with OCI runtime tools[5]. Other components of `cri-o` allow users to manage storage, image build, networking, and monitoring.

[1] google/lmctfy
[2] coreos.com/rkt
[3] `rkt` OCI roadmap
[4] cri-o.io
[5] opencontainers/runtime-tools

These are the most prominent container runtimes that were used over the years. Again we need to take into account the Sysdig 2019 report (28) indicates 79 % of users use the Docker container runtime, and 18% of users use the `containerd` runtime which is the base runtime of Docker as well.

2.3 Docker Services

Now that we know what is happening under the hood of the container management platforms, we can look at the wider set of services that Docker is composed of. These services go beyond just running containers, and extend to managing the whole lifecycle from building images to deploying containers, and providing a user interface for interacting with the set of features.

In the Figure 2.1 we can see an example of a Docker architecture, along with the artifacts that the components are managing.

The core of Docker is the local Docker engine, as indicated by the combination of Docker Client, REST API, and Docker Daemon. Docker Daemon is a long running server that that exposes REST API for clients. Docker Daemon is also the location of `containerd` runtime. Docker Daemon is responsible for managing all aspects of the container lifecycle – it builds images from Dockerfiles, it fetches missing layers from the external registry and stores them locally. It is also responsible for creating and destroying volumes for data persistence, and networks used by the containers (as the responsibility of the container runtime). It also performs the core functionality: running containers.

In the Figure 2.1 we see the component responsible for storage and distribution of container images: *Container Image Registry*, or shorter *Container Registry*. We highlighted two types of container registries in the figure: local and external. We discuss container registries and different use-cases for different types in-depth in Section 2.4.

2.4 Container Image Registries

Container Image Registry is a service for storage and distribution of container images. In the development of containerized systems where container image is the key software artifact, container registry is the critical component in the software development and deployment process.

Container registry organizes container images in repositories, similar to how software code is organized in versioning platforms such as Github. All image versions, distinguished

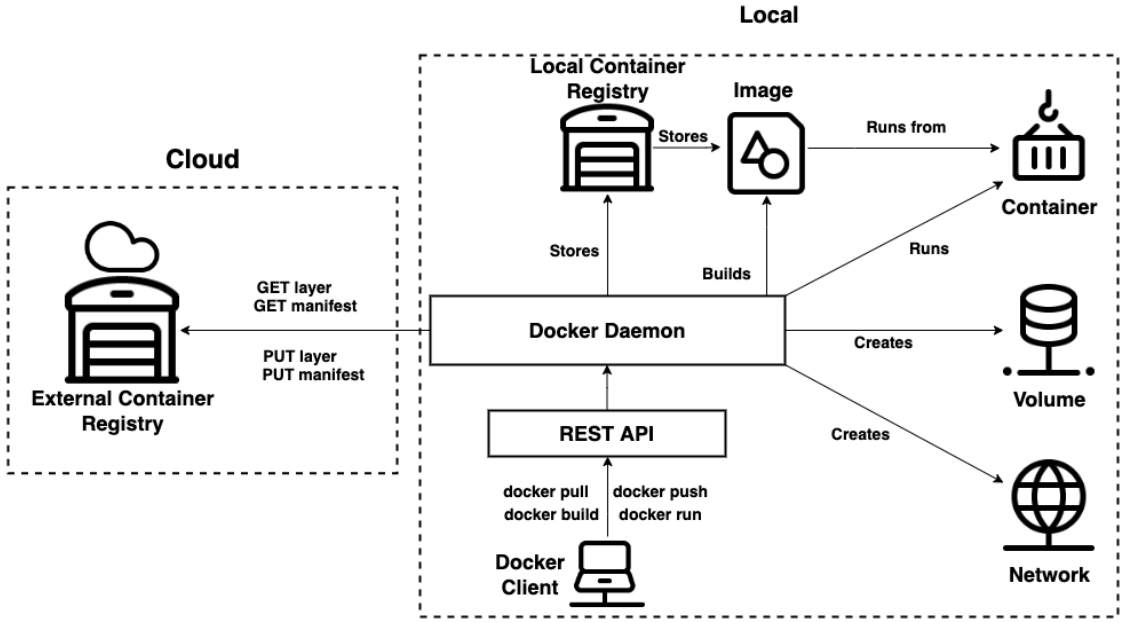

Figure 2.1: An example of Docker services interaction.

by the image tags are grouped inside one repository. Developers push images with new tags to the container registry which correspond to new application versions. Repository organization enables fine-grained access control for container images.

Organizations can use managed versions of container registry, or deploy the container registry themselves. Public registry platforms are registry services where users and organizations can store images publicly for anyone to pull. This is a common way of container image distribution for the open-source projects. Examples of the public registry platforms are Docker Hub, Quay, and Gitlab.

Large cloud providers offer container registry services on their platforms. For organizations that have their services deployed inside the cloud platforms, using the native platform registry brings many advantages. Keeping the container registry close to the deployment reduces the container image pull and push latency. The cloud registries can be deployed in a quick manner, and are integrated with other cloud native container management services such as managed Kubernetes. Cloud container registry instance is private and separated from other cloud container registry instances on the same platform. Cloud container registries offer other functionalities out-of-the-box such as vulnerability analysis, geo-replication, webhook integration, etc. Examples of cloud platform registries are AWS Elastic Container Registry, Google Cloud Registry, and Azure Container Registry.

Another option for the organizations is to deploy a container registry themselves. The container registry can be deployed in the cloud as a cloud hosted service, or on-premise for enhanced security. On-premise registries are typical for the organizations that have a private cloud and want to minimize external dependencies, especially for the critical components of the deployment process. On-premise container registry services support additional functionalities such as geo-replication, vulnerability analysis, webhook integration, however additional configuration and setup is necessary. There is a number of open-source container registries that can be deployed on-premise. Examples are Harbor, Quay, Docker Registry, and JFrog Artifactory.

Container registry can be deployed locally to act as a *pull through cache*. In this deployment type the local container registry acts as a proxy cache. Every image pull request goes through the local registry, and if the registry contains the requested image layers it serves it. In case the registry does not contain the image layers needed, it forwards the request to an external registry. The image layers served from the external registry are stored in the local registry and returned to the source of the request. This type of deployment is shown in Figure 2.1.

2.4.1 Container Registry Interactions

Interactions with container registry are standardized with the Docker Registry V2 protocol [1]. This protocol defines the API for registry operations. The registry API defines common format and URI conventions for operations such as authentication, image pull, and image push. Docker Registry V2 API is implemented by all well known container registries and clients that manage image distribution. Usually a client is used to automate the registry operations. Client interacts with the container runtime to determine what layers to pull or push. An example of a client is the Docker client. When running commands such as `docker run` or `docker pull`, based on the availability of layers in the local machine, the client interacts with the container runtime and the container registry.

The two core standardized registry operations are *image pulling* and *image pushing*. The sequence of registry requests is opposite for the two operations.

When pulling the image, first the request for image manifest is sent to the following URL: `GET /v2/<name>/manifests/<reference>`. The `name` and `reference` URI parameters are required to identify the image. The `name` corresponds to the image name, and the `reference` is the image tag or digest. Once manifest is retrieved, the layers that

[1] docker.com/registry/spec/api

need to be pulled can be determined by looking at the layer digests specified in the manifest. The clients request layer digests that are missing in the local storage. The layer GET request is issued in the following format: `GET /v2/<name>/blobs/<digest>`, where `name` is the image name, and `digest` is layer digest. The image push operation works in the opposite order. First, a POST request is issued with the format `POST /v2/<name>/blobs/uploads/`, where `name` is the image name. If the POST request is successful, a response will contain the `Location` field with the upload URL of the following format: `/v2/<name>/blobs/uploads/<uuid>`. This upload URL is used to push an image layer to the registry. To push the layer, the client issues a PUT request to the following URL: `/v2/<name>/blobs/uploads/<uuid>?digest=<digest>`. After all the image layers are pushed, the client pushes the image manifest with the following URL: `PUT /v2/<name>/manifests/<reference>`, where `name` is the image name and `reference` is the image tag or manifest digest.

We described the two most common registry operations, and many more are described in the Registry V2 API. For example listing images, listing repositories, checking whether image exists, authentication, etc.

2.5 Efforts to Standardize Containerization: Open Container Initiative

Open Container Initiative (OCI) is an open governance founded by leaders in the container industry. It is the culmination of attempts to introduce standardization around container runtimes and container formats.

The culmination of the OCI efforts are the OCI Runtime and Image Format Specifications. The OCI runtime specifications outline how to run the containers once the image bundle is unpacked on disk. It defines 5 principles of Standard Containers: *standard operations*, *content-agnostic*, *infrastructure-agnostic*, *designed for automation*, and *industrial-grade delivery*. These principles guide the further technical specifications of the container runtime.

The specifications are further expanded to define platform-specific configurations (for example, namespace mappings and cgroups path), filesystem bundle, and other technical aspects of container technology. The basic implementation of these specifications is defined as OCI runtime.

The image format specifications define the OCI image, consisting of the manifest, filesystem layers, image index, and a configuration. Each component of the OCI image is further

specified to deliver a standard format. Image format specifications specify other types of OCI images. The other image format types are Helm [1] chart (used for deploying Kubernetes pods)and Singularity [2] image (single file container image format).

Docker spinned off `runc` and `containerd` projects and donated them to OCI. This was done with the goal of making these projects OCI compliant. We also mentioned that `rkt` switched from the App Container (appc) image format to the OCI image format, therefore we conclude that the OCI standards are reaching widespread adoption.

Open Container Initiative leads the effort for the standardization of container registries to support all OCI artifacts (Helm charts, Singularity images). The culmination of these efforts is ORAS [3]: OCI Registry As Storage. ORAS is a project developed by Microsoft and donated to OCI. ORAS is a CLI tool and Go module that can be used to push OCI artifacts to OCI compliant registries.

[1] helm.sh/charts
[2] singularity.lbl.gov/about
[3] github.com/oras

3

MLR Survey of Container Registry Systems

Following a general introduction to concepts that are key to this thesis in Chapter 2, in this chapter we launch an extensive literature and system survey of state-of-the-art in the field of container registries.

We have identified scientific articles on the topic of container registries, mostly focused on performance improvements of the various parts We have not identified any systematic system surveys in the scientific literature. We hypothesize this is partly due to the novelty of the field, and also due to the dominance of hosted, closed-source platforms. However, we can observe an abundance of blog posts, articles and videos on this topic. This highlights the community interest, and shows the need for a more systematic approach to the topic.

Figure 3.1 shows the current state of the field, as taken from the Cloud Native Computing Foundation website [1]. Out of twelve container registries shown in the figure, only six are open source.

In Section 3.1 we present the method we used to collect relevant literature and other sources of information, such that the literature survey is clear and reproducible. Along with this we provide statistical information on the data we obtained using these queries.

In Section 3.2 we discuss and compile our results to produce a mapping of current state-of-the-art to the categories identified in Section 3.1.

Lastly, in Section 3.3 we analyze the mapping and answer the questions we posed at the start of the survey.

[1] cncf/container-registries

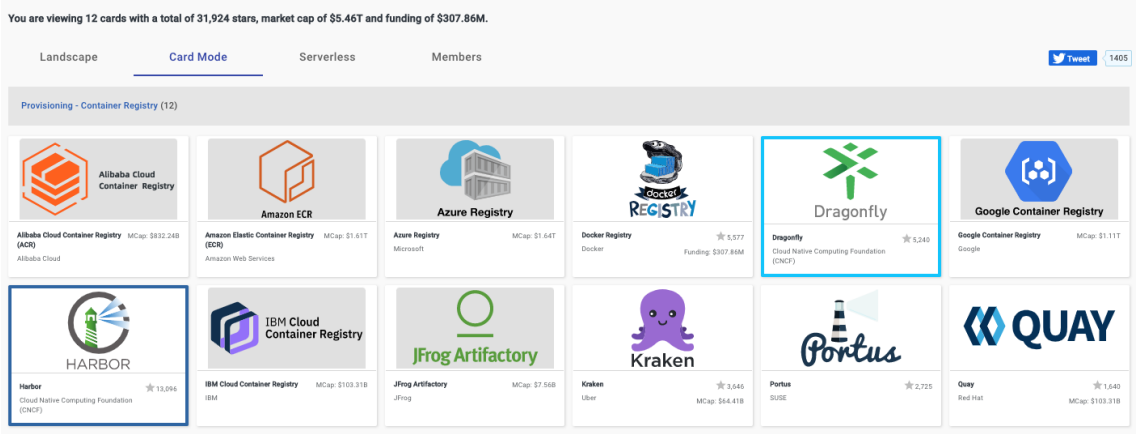

Figure 3.1: An informal view on cloud registry landscape, as of November 27th, 2020.

3.1 Methodology

Due to the system-focused nature of this survey, we identified several methods of resource gathering. The state-of-the-art method identified is the Systematic Literature Review (SLR) by Kitchenham and Charter (31). It defines a sequence of steps for a reproducible literature survey. Other literature survey methods that we identified are snowballing, and unguided traversal of the material. Unguided traversal of the material is an extensive process in which the researcher is trying to obtain as much material as possible, starting from the initial set of articles. In the case of snowballing, the researcher launches a process similar to the unguided traversal, but in this case with a pre-defined selection criteria. The issue with these two methods is reproducibility and the stoppage criteria. Two different researchers launching the same literature survey may end up with a completely different literature sets.

The SLR method solves this by presenting a set of steps that will lead to a reproducible literature survey. It splits the review process into three steps: planning the review, conducting the review, and reporting the review. The first step concerns developing a clear review protocol, defining research questions and query terms for the literature survey, and evaluation of the protocol itself. In the second step the selection of the literature based on search queries is performed. In the third step, the synthesis of data is performed, along with the report of the findings.

The SLR method is well adjusted to the goal of finding data from the primary literature (scientific articles, research papers), however due to the system-focused nature of this survey and the fact that the resources on existing platforms cannot be found in the scientific

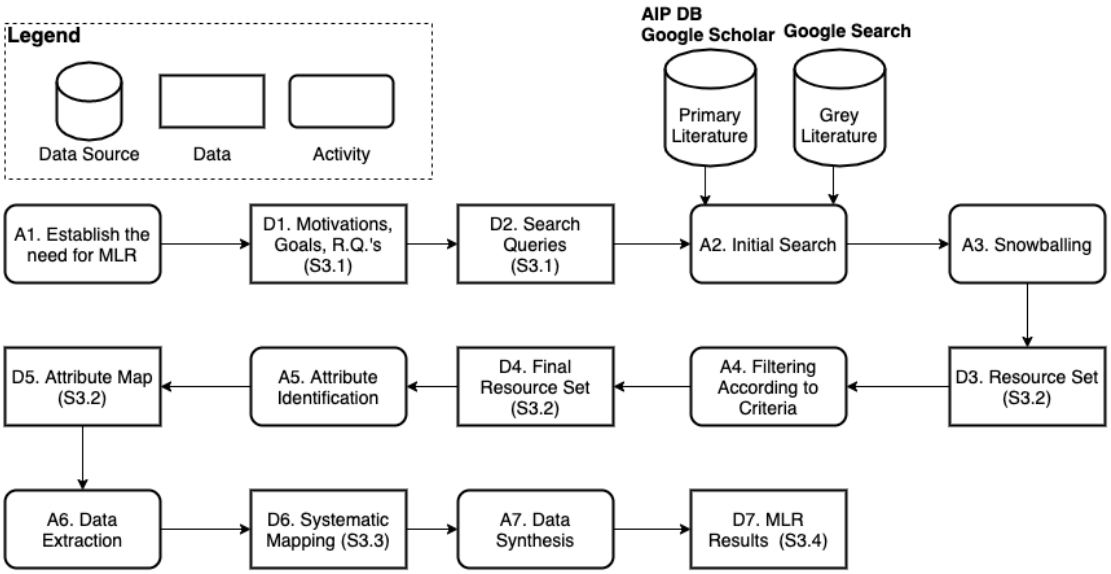

Figure 3.2: Multivocal Literature Review process.

literature, there is a need to extend the range of resource types to grey literature (GL). Grey literature is defined as literature that *is produced on all levels of government, academics, business and industry in print and electronic formats, but which is not controlled by commercial publishers, i.e., where publishing is not the primary activity of the producing body*(32). Examples of grey literature are government documents, videos, lectures, data sets, blogs, etc.

We identified Multivocal Literature Review (MLR) method (33) as the state-of-the-art method for conducting the survey on both primary and grey literature. This method is an adaptation of the SLR method, extending it with a set of requirements for the selection of grey literature resources. The MLR method includes both the SLR method, and the Grey Literature Review (GLR) method, which aims to review only sources from grey literature.

Figure 3.2 shows the version of the MLR method we applied to this literature survey. We have excluded the evaluation activities conducted after forming the methodology and after obtaining the set of resources due to the time pressure and extensiveness of these procedures. In the figure we identified data artifacts (D) that are produced after each of the activities (A).

Earlier in this section we identified the lack of a systematic survey of container registry systems, and the community interest in the topic. We have also explained why there is a specific need for the application of the MLR method (A1). We now formulate the specific

goals, and research questions (D1):

G1. Bridging the gap between industry and academia. There is a growing interest in the field of containers as a whole, and along with that also the container registries. We identified scientific literature concerned with the topic of utilizing container image properties to deliver better performance. On the other hand, we have also identified a plethora of conference talks, blog posts and community articles written about container registries.

G2. Gain insight into current advancements and concerns. We want to gain a better understanding of the direction in which the industry is heading. We want to know what are the container registry bottlenecks the organizations are observing, and what are the solutions.

G3. Obtain knowledge about relevant performance metrics and workloads. We want to continue from the knowledge obtained in this survey to create a container registry benchmark that reports relevant metrics to the user. To do that, we need to gain knowledge on performance evaluation of the current state-of-the-art. It is crucial to understand what is the final aim of the latest performance optimizations in both academia and industry.

The main research question that we draw from the formulated goals is *What is the current state-of-the-art in the field of container registries?* (SRQ1). This question is directly related to all three goals we mentioned before. It is formulated broadly enough to encompass our interest in performance optimizations, features, and current concerns. We can further specify this question in two sub-questions: *What is the current state-of-the-art in the research on container registries?* (SRQ1a), which we answer by surveying the topic of container registries in the scientific research, and *What is the current state-of-the-art in the field of production-grade container registry systems?* (SRQ1b). We answer the question by surveying the container registry systems in the industry. We combine both of these surveys into a Multivocational Literature Survey which result in a mapping of container registry systems to the predefined attributes.

We have formulated initial queries with which we have aimed to obtain current state-of-the-art in scientific literature, and to form an initial seed set of grey literature which help us identify platforms to expand the set of queries and to start the snowball.

Search Type	ID	Query
Initial Search	Q01	`container registry`
Initial Search	Q02	`Docker registry`
Snowballing	Q03	`Uber Kraken`
Snowballing	Q04	`Alibaba Dragonfly`
Snowballing	Q05	`Harbor`
Snowballing	Q06	`Quay`
Snowballing	Q07	`Portus`
Snowballing	Q08	`ECR/Elastic Container Registry`
Snowballing	Q09	`ACR/Azure Container Registry`
Snowballing	Q10	`GCR/Google Cloud Registry`
Snowballing	Q11	`ICR/IBM Container Registry`
Snowballing	Q12	`Alibaba Container Registry`
Snowballing	Q13	`JFrog Artifactory`
Snowballing	Q14	`Docker Hub`
Snowballing	Q15	`Github Container Registry`
Snowballing	Q16	`Gitlab Container Registry`
Snowballing	Q17	`Facebook Location Aware Distribution`

Table 3.1: Initial and snowballing queries.

3.1.1 Queries

The set of queries used for initial search and snowballing is displayed in Table 3.1. This data artifact is depicted in the Figure 3.2 under D2. Since we started with the empty set of registries, we aimed to populate it using the snowballing process (unguided search with stoppage criteria). We structured snowballing process in the following way. We obtained an initial sample of registries based on the initial queries. We added these registries to the registry set. We then used these registries as search queries, and from there we obtained new registries (usually from the comparative analyses in blog posts or videos), and repeated the procedure until the stoppage criteria. We stopped the search once we did not find any new registries in an iteration of the snowballing process.

For the primary literature we selected two sources of information: Google Scholar and the AIP DB (34) that combines resources from three prominent sources of primary literature: DBLP, Semantic Scholar, and AMiner.

For the grey literature we selected one main source: basic Google search. However, when performing the snowballing process (A3), we also used Youtube as the source for talks and

webinars about registry systems.

The product of this process is shown in the Figure 3.2 under D3.

3.1.2 Selection Criteria

We have constructed a separate selection criteria for primary and grey literature. For the primary literature, we select all articles published in journals and conferences. We exclude secondary studies, such as theses, as they usually do not have enough credibility and do not go through the same validation process as the articles published in top tier journals. We specifically exclude scientific articles that do not concern primarily with container registry systems or container registry optimizations, but focus on introducing new extensions, such as (35, 36).

As for the grey literature, the selection criteria needs to be rigorous as the validation is either not conducted or it is vague. We have decided to include conference talks about specific systems architecture, talks or webinars about features of the system only if they origin from the well known organization (such as CNCF, Red Hat, platform company itself). We have also decided to include Github and original platform documentation, as usually the review process is more structured. We also include blog posts by registry developers. We also include blog posts if they are from the well known technology portal or consultancy company, only if its concerned with comparisons between different platforms (however, we do not include comparisons created by platform organizations itself, as they tend to be biased). We include Stack Overflow as its community has a good reputation for rating valid answers highly and discarding false answers. We specifically exclude medium posts (if they do not fall under any category mentioned above), unverified blog posts, and all the other resources not mentioned. We use this selection criteria for filtering an initial set of resources obtained from query search and snowballing (Activity A4 in the Figure 3.2).

The applied filtering is shown in Figure 3.3. In regards to the container registry system search we mostly filtered out the guides on registry deployment as these resources did not satisfy the relevance and source validity requirements. The product of this process is labeled as D4 in the Figure 3.2.

3.2 Results

In this section we present the results from reading the obtained resources. We first describe the attributes we defined for characterization of the registry services in Section 3.2.1. This activity is indicated as A5 in the MLR process. Then we present the mapping of registry

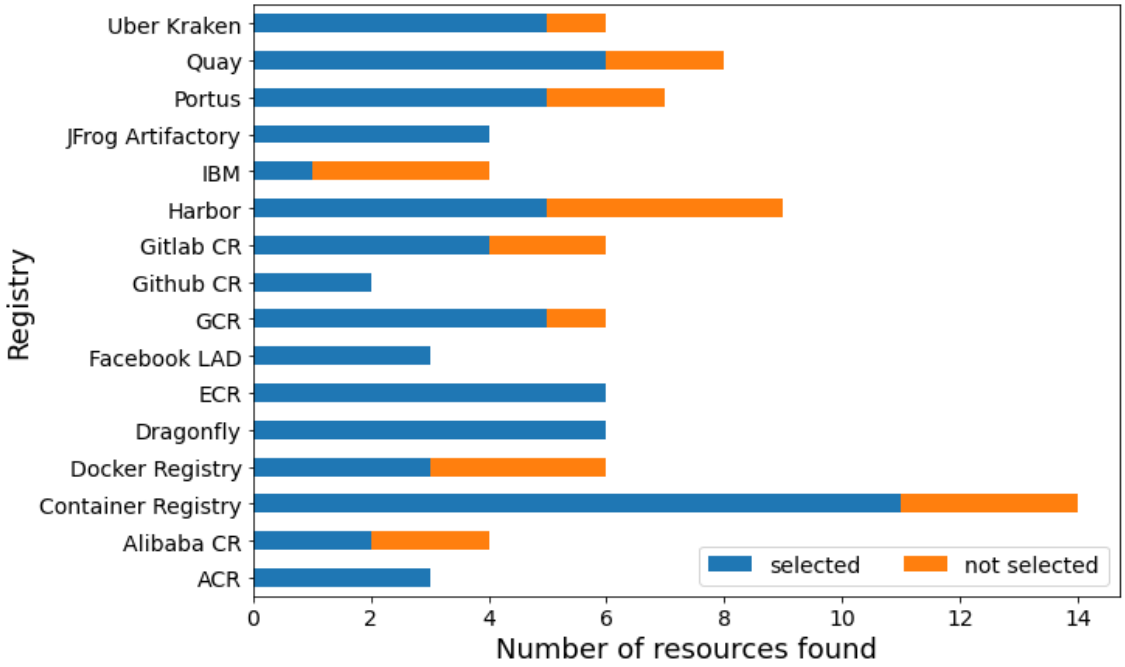

Figure 3.3: MLR resource gathering and selection results.

services to the attributes in Section 3.2.2. With these results we aim to answer research question SRQ1 posed in Section 3.1.

3.2.1 Attributes

A key challenge in defining a set of attributes for the surveyed services is that in the set of services there are both open-source and closed-source platforms. We have also identified that the goal of the platforms is either to provide as wide set of features as possible, or to act as a thin file distribution layer that is completely focused on performance.

To present a consistent mapping that gives insight about both closed-source and open-source services, we defined the following attributes:

A1. Open-source: Whether the registry code is open-source. This is a main point of distinction between the systems described here.

A2. Offered as a Managed Service: Whether there exists a managed instance of the registry, open for public. This presents the ease of use for the user, since they do not have to worry about deployment.

A3. Focus on Performance: Whether the registry design is optimized for performance first. We determine this by looking at the amount of resources about the registry

performance: 3 or more means service is focused on performance, 1 or 2 means the service is partly focused on performance, and none means it is not focused on performance.

A4. Community Recognition: Whether the registry has an active community (for example, number of Github stars), or there is a lot of community articles and blog posts online. Open source registries are community recognized if they have more than 10,000 Github stars, partly community recognized if they have more than 4,000 Github stars, and less than that means registries are not community recognized.

A5. Focus on Security: Whether the registry puts extra focus on the security, through offering many authentication and authorization options. If the registry offers more than three authentication techniques and vulnerability scanning it means its focus is fully on security, if it offers one or two authentication techniques and vulnerability scanning then it is partly focused on security, and if it doesn't contain vulnerability scanning or has just one authentication technique then its focus is not on security.

A6. Geo-Replication: Whether the registry supports geo-replication.

A7. Artifact Registry Whether the registry supports OCI artifacts, such as helm charts or Singularity. We define registry as a full artifact registry if it supports OCI image format and other artifacts within the same service, and if it is supported by ORAS (OCI Registry As Storage) tool. We define registry as a partly artifact registry if it supports OCI image format and there is a service within the same platform that offers storage and distribution of other artifacts. We define registry as a container image only registry if it doesn't support OCI image format and there is no other service within the same platform that handles other artifacts.

The attribute map is the data artifact indicated under D5 in the Figure 3.2.

3.2.2 Attribute Mapping

In Table 3.2 we present the mapping for the set of registries selected. This attribute mapping is created to answer the research question SRQ1b: *What is the current state-of-the-art in the field of production-grade container registry systems?*.

When analysing the registries and mapping them to the registries defined in the previous subsection we performed additional selection of registry platforms. The reason for the additional selection is that we do not know too much about certain registries such as

Registry Platform	A1	A2	A3	A4	A5	A6	A7
Docker Registry	●	◐	○	◐	◐	○	●
Kraken	●	○	●	◐	○	○	○
Dragonfly	●	○	●	◐	○	○	○
Harbor	●	○	○	●	●	◐	●
Quay	●	●	○	○	●	◐	◐
Portus	●	○	○	○	●	○	○
ECR	○	●	-	●	●	●	●
ACR	○	●	-	●	●	●	●
GCR	○	●	-	●	●	●	◐
ICR	○	●	-	◐	◐	○	○
Alibaba CR	○	●	●	●	◐	○	●
Gitlab CR	○	●	-	●	◐	◐	◐
JFrog Artifactory	●	○	-	●	●	●	●
Docker Hub	◐	●	○	●	◐	○	◐

Table 3.2: Mappings of registries to the identified registry attributes. Full circle: attribute present, half circle: attribute partly present, empty circle: attribute absent. ECR is Elastic Container Registry, ACR is Azure Container Registry, GCR is Google Container Registry, and ICR is IBM Container Registry.

Facebook LAD. Also, Github CR is a newly introduced registry and its characteristics based on the attributes we defined cannot be sufficiently determined at this time. The full list of registries found during the search process is added to appendix. This process is indicated in the Figure 3.2 under activity A6.

This mapping is indicated in the Figure 3.2 under data artifact D6.

In the mapping we can see the clear distinction based on the attribute **A1: Open-Source**. Docker Registry, Kraken, Dragonfly, Harbor, Quay, and Portus are all open source project, with public repositories available at Github. We have marked Alibaba CR and Docker Hub as partly open sourced, as we know from the resources obtained that they are based on Dragonfly, and Docker Registry respectively. Regarding the other projects, we got information from the Azure Container Registry team that the ACR was initially based on the Docker Registry project, however it diverged to support different features needed by the users.

A2: Offered as a Managed Service, is an attribute that is part of every evaluated registry with exception of Portus, Kraken, Harbor, and JFrog Artifactory. We do not know the extent of divergence between Alibaba CR and Dragonfly, therefore we cannot

make connections between these two projects. We applied the same logic to the Docker Registry, as we know many of the hosted registries are based on this project. Kraken is the project by Uber and it is used in their deployment, however it is not hosted publicly as a standalone project. Harbor and Portus are projects aimed for private clouds, and also do not offer a hosted platform, and none of the public hosted registries are based on these projects.

A3: Focus on Performance was the most challenging to determine for the closed-source registries. This is one of the reasons for this project, and we evaluate performance of these platforms in Chapter 5. We identified Dragonfly and Kraken projects as the projects which primary goal is to tackle the registry throughput bottleneck for large deployments. We also marked Alibaba CR as the platform focused on performance due to its connection to Dragonfly. Docker Registry (Docker Hub), Harbor, Quay, and Portus are on the other hand registries which primary goal is to integrate a wide range of features that enhance the dev-ops cycle and reduce the amount of components the user needs to connect in the dev-ops pipeline. For these registries performance is the secondary concern, which we also validate in Chapter 5.

A4: Community Recognition, the purpose of this attribute is to rank the registries according to the community involvement, and their usage. It is again hard to evaluate the closed-source registries on this attribute since we cannot just take a look at their Github stars, but we tried doing it by counting the number and types of the resources created by the community. Out of all open-source projects, we mark Harbor as the one with the largest community (13.5k stars on Github). After that, we identified Docker Registry, Kraken, and Dragonfly as partly recognized projects (around 5k Github stars). Lastly we labeled Quay and Portus as projects that lack community recognition (around 2k Github stars). In the closed-source registry group, we stated before that Docker Hub is the absolute leader for the number of registries hosted. We also label Gitlab CR, Alibaba CR, GCR, ECR, ACR, JFrog Artifactory as projects that have good community recognition due to the amount of grey resources we found. On the other hand, we mark IBM CR as partly recognized, as we failed to obtain many relevant resources on it, although its traces are used in the registry research community (19).

A5: Focus on Security creates a distinction between registries whose primary focus is a wide range of security features and access control. Many registries from this set are now offering image vulnerability analysis on push. There are also other features such as RBAC, different forms of authentication, etc. Many of the open source projects are focused on security, as this is a primary concern for the customers that host their images in a self

hosted registry. Specifically Harbor, Quay, and Portus are all offering a wide range of security features such as LDAP user authentication, OAuth, application tokens etc. On the other hand, the registries offered by the public cloud providers retain the security features that are present across all cloud services. This gives users security and trust in the safety of their images.

A6: Geo-replication is offered in two distinctive categories. The first category are managed registries that offer geo-replication as the one-click feature. The registry needs to be configured in each region, and the storage costs are incurred per registry instance. The second category are the private self-hosted registries that support geo-replication, however geo-replicated storage needs to be configured. We mark registries as partly supporting geo-replication if there is a managed version of the registry that does not support geo-replication, but it can be configured in the self-hosted version. ECR, ACR, and GCR offer geo-replication as a one-click feature. Harbor offers replication endpoint configuration. Quay.io doesn't offer georeplication, however the self-hosted Quay version supports geo-replication. Gitlab CR supports geo-replication as part of the self-hosted premium version, but its hosted version which is part of the Gitlab platform doesn't. Docker Hub, Alibaba, IBM, Portus, Kraken, and Docker Registry do not support geo-replication.

A7: Artifact Registry relates to the OCI compliance, and its supported by most of the registries. It is outside of the scope of Kraken, Portus, and Dragonfly as their use case is different than other registries. Quay partially supports this attribute as we did not find explicit support for Helm charts and Singularity images.

3.2.3 State-of-the-Art in Research

Due to the novelty of the container registry systems, we have obtained only a few resources from the primary literature that are concerned with proposing new systems or optimizations to the existing ones.

The paper by Harter et al. (1) is the oldest primary literature resource that satisfied our selection criteria. It is focused on improving container startup times by optimizing the interaction between the Docker client and the registry, by creating an NFS server in which both registry and the clients continuously maintain a connection. They present their work as Slacker, a Docker storage driver that improves pull times from a Docker registry by lazily fetching layers when necessary, instead of pulling all the image related data immediately. This paper also presents a container registry benchmark HelloBench, which measures performance for complete docker commands.

Another paper we selected is the paper by Anwar et al. (19) which was done in collaboration with IBM. This paper is influential for later work in the container registry research field, as it presents an open-source collection of traces recorded from IBM production and staging registries. These traces, along with the replaying tool, are used across all the later papers we identified in this research field. The tool can be used for benchmarking local or remote registry implementations, though it lacks some key features such as authentication and manifest generation to better simulate real-world conditions.

Along with these traces they proposed registry optimizations. The first optimization is the two-level caching which exploits the registry workload characteristic of storing very small and very large files. The small files are kept in memory and the large files are kept in SSD. This design had a high cache-hit ratio in the evaluations they presented. Another optimization they proposed is prefetching layers based on the patterns observed in trace, for example prefetching layers that were recently pushed as there is a high probability they will be pulled immediately.

This paper was followed by another paper from the collaboration by Virginia Tech and IBM, which presented the container registry system Bolt (18). Bolt is a distributed registry where each registry node has a local storage instead of a separate storage component from which it fetches the necessary data. Every node stores a subset of the registry data, and the location of a certain file is determined by its SHA256 hash. There is a Zookeeper node that coordinates all the nodes in the registry cluster. This design eliminates the centralized storage as the bottleneck, and reduces the amount of hops the request needs to take, effectively reducing latency and increasing registry throughput when comparing with the docker registry client.

The final paper we selected is another collaboration between Virginia Tech and IBM Research, resulting in DupHunter (22), an extension of Bolt. DupHunter is presented as an aim to introduce deduplication optimization for the Docker registry. Since Docker images are composed of several layers, a lot of images share basic layers that represent language environments and common package configurations. However, existing deduplication techniques will not result in a performance gain due to specific patterns of access for Docker image layers. Therefore, DupHunter exploits these access patterns for layers, by replicating the most frequently accessed layers, and deduplicating the less frequently accessed layers. This optimization does result in performance gains but requires fine tuning of deduplication configuration. As Bolt, DupHunter uses the IBM traces to evaluate the performance gains.

This analysis, together with the discussion in 3.3 is indicated in the Figure 3.2 under A7.

3.3 Discussion

Based on the knowledge obtained from the MLR survey we hope to answer the research question RQ1 stated in the Section 1.3: *What are the relevant key aspects of a container registry benchmark?*

Based on the metrics, workloads, and registries we extracted and synthesized from the scientific and grey literature, we selected types of metrics and workloads, and registry sets that we take into account for the benchmark instrument design in the Chapter 4, and design experiments in the Chapter 5.

The resulting selection of metrics, workloads, and registries is the final data artifact D7 in the Figure 3.2.

3.3.1 Metrics

With regards to the performance metrics observed in the literature and in the system resources we identified the following three metrics that can describe registry performance: *latency*, *throughput*, and *cache hit ratio*. These three metrics were used for evaluating registry optimizations proposed in the scientific literature. Throughput and latency were also used in talks about Dragonfly and Uber for reasoning about performance improvements.

Since it is not possible to measure cache hit ratio when testing production-grade systems as a black box, we exclude this metric from the key metric selection that we use in later chapters.

3.3.2 Workloads

We have seen that all state-of-the-art in container registry scientific research uses IBM registry workload traces presented in (19) to evaluate container registry optimizations. These traces present two opportunities. First, by using the standardized workload in the container registry research we contribute to creating a common ground for a standardization in container registry benchmark field. Secondly, given this trace format there is a possibility of auto-generating traces with certain behavior patterns that could be replayed by the same tools used for replaying these traces [1].

[1] github.com/chalianwar/docker-performance

3.3.3 Registry Selection

We surveyed the state-of-the-art in the container registry industry, and with the knowledge of the current state of the industry field we can now define the list of the production-grade hosted registries that are widely used and recognized in the community. We excluded the registries that are not offered as a managed public service: Docker Registry, Kraken, Dragonfly, Portus, and JFrog Artifactory. We categorize the registries in two categories: *public registries* and *private registries*, which we used for performance comparison and further selection:

- **Public registries:** Quay.io, Gitlab, Docker Hub.
- **Private registries:** ECR, ACR, GCR, ICR, Alibaba CR.

Public registries are the registries that are focused on offering both public and private repositories and they allow a certain number of public and private repositories per user or organization. For the private registries the main focus is on the data and storage size. They offer registries deployed in the regions, which allows for having services close to each other geographically, which in turn results in lower latencies. These registries do not focus on numbers of repositories as part of their pricing model.

3.4 Threats to Validity

In this section we analyze possible issues with this literature and systems survey. Firstly, a possible threat is the selection criteria. We have not defined a clear set of websites and publishers that we accept, rather the selection criteria is based on multiple factors defined in 3.1.2. We have not evaluated the selection criteria to determine whether the requirements are too lenient or too strict.

Another threat to validity is based on the attribute selection. Due to the novelty of the field and variety of registries with different goals we have tried to select attributes that provide an informative overview and create a common ground for these registries. However, due to the focus on creating a common ground there is a danger of not sufficiently describing all registry features, which can cause some registries to appear as they are missing features compared to others.

4

Design and Implementation of the Container Registry Benchmark

This chapter answers the research question RQ2: *How to design a registry benchmark that supports relevant key aspects?*

In Chapter 3 we have identified state-of-the-art in the field of container registry benchmark. Our observation is that the current registry performance evaluation tools and instruments are directed towards evaluating registry optimizations in research. This has a key consequence for the general usability of these tools, as the optimized registries that are used in these evaluations are stripped of the real-world features such as authentication.

This observation is tied to the fact that there was no relevant work observed in the evaluation of public cloud registries. To fill in this gap we create an instrument that evaluates a wide range of public and private container registries using generated synthetic workloads. Additionally, we integrated a customized version of the Trace Replayer tool from (19) to support performance evaluation using real-world workloads.

In this chapter we present a container registry benchmark instrument that aims to bridge the container registry performance evaluation gap between research and industry. In Section 4.1 we describe the components that are part of the System Under Test (SUT). In Section 4.2 we first present a requirements analysis to formalize core goals of the instrument. In Section 4.3 we describe the design of the instrument, both through analysis of each individual component and through various diagrams. In Section 4.4 we use the general requirements for a successful benchmark defined in (5) as a checklist to ensure the relevance of our benchmark. In Section 4.5 we discuss the key implementation choices and we present the open source repository for the instrument. Lastly, in Section 4.6 we discuss

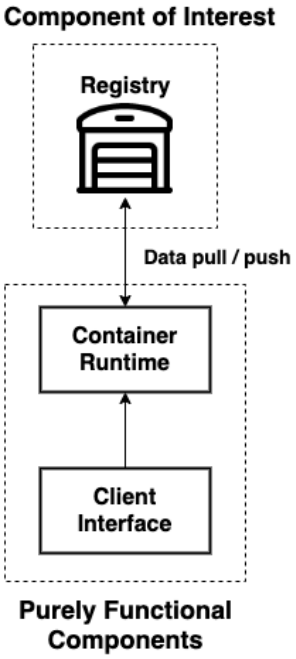

Figure 4.1: The high-level components of a typical registry System under Test.

in-depth the state-of-the-art container registry performance measurement tool, the issues with it that we observed and the analysis of improvements to the state-of-the-art.

4.1 System Under Test

System Under Test (SUT) is a system that is composed of components, which is of interest to the benchmark(5). However, the benchmarks are usually concerned with only a certain component within the SUT. These components are labeled as *components of interest*. The rest of the components are known as *purely functional components*. In our case we define the SUT as the container management system.

We describe the SUT components with a high-level diagram of the container management service in Figure 4.1. As highlighted in Chapter 2, the container management service is composed of different components. In this diagram we isolated the components that take part in the management of container images.

We have labeled the Registry component as the Component of Interest, and the rest of the components as Purely Functional Components, meaning that we do not consider their performance in the benchmark. We try to isolate the interaction between the Container

Runtime and Registry, without interacting with the other components, to achieve a more fine grained control over the operations performed by the registry.

4.2 Requirements Analysis

In this section we expand on the insight gained in the Chapter 3 to structure stakeholders, their concerns, and requirements for the benchmark instrument to answer the research question RQ2 in a satisfying way. The full requirement analysis helps to guide the design and implementation of the benchmark instrument according to the software and benchmark best practices.

We first describe stakeholders in Section 4.2.1. From there we describe the expected use cases in which these stakeholders will use the instrument in Section 4.2.2. Based on the use cases we define functional (Section 4.2.3) and non-functional (Section 4.2.4) requirements.

4.2.1 Stakeholders

The main stakeholders we identified are *performance engineers*, *scientific researchers*, and *DevOps engineers*. These stakeholders are envisioned to be the primary users. There are other stakeholders involved, for example managerial team in a company, but their involvement is secondary and their interests are beyond the scope of this instrument (producing report sheets, including complex fine-grained cost reporting etc.).

S01. Performance Engineers are the direct users of this instrument. They come from the container registry development team that is interested in comparing the performance of their registry to other registries. The performance engineers can also come from companies that develop latency sensitive applications with containers.

S02. Scientific Researchers are the stakeholders from the academia, and also direct users of the instrument. For the researcher it is important that the instrument can be easily extended with new functionalities. It is also important that the instrument can be deployed in distributed setting, and run both real-world and synthetic workloads. An important requirement for the academia is that the instrument must be open-sourced for scientific researchers to examine and reason about the instruments correctness, but also to test the validity.

S03. DevOps engineers are interested in reducing deployment time. The aim is to reduce the downtime which is also an important business goal as it relates directly to the availability guarantee in the Service Level Agreement (SLA).

4.2.2 Use Cases

We define general use-cases for the benchmark instrument, which help us in the requirements specification phase. The use-cases are drawn from aforementioned stakeholder concerns. As with the stakeholders, the list of all the possible use cases for the benchmark instrument is exhaustive and can be expanded to cover other specific use-cases for different types of performance and general cloud system evaluations. The three use-cases selected here are *performance monitoring*, *performance optimization evaluation*, and *DevOps process evaluation*:

UC1. Performance monitoring (adresses the stakeholder S01 concerns in Section 4.2.1): This benchmark will be especially important for highly available, latency sensitive applications. Running daily performance tests on each component of the system is important for identifying bottlenecks and delivering promises from service level agreements. In serverless platforms, container registries are a crucial component in the architecture so monitoring its performance is especially important.

UC2. Performance optimization evaluation (S01, S02): Further optimizations to the registry, especially caching optimizations need to be evaluated to determine if the optimizations are indeed valuable. This use case is also related to cloud architects and managers selecting right container registry based on the performance and cost ratio. This use case also encompasses the scientific research, as optimizations in registry design are currently evaluated using similar instruments.

UC3. DevOps process evaluation (S01, S03): This instrument will be useful to perform evaluations of container registries when faced with a high number of requests. In case of image updates in large Kubernetes clusters, every node might request the image from the registry. It is important to report the request throughput metric to provide insight into how the registry copes with bursty loads.

4.2.3 Functional Requirements

We split the functional requirements in two categories: System level (SR), and deployment level (DR). We do this to highlight the difference between must-haves on the level of pure functionality of the system, and the must-have requirements in regards to control over deployment and deployment scenarios the instrument should run. Every requirement starts with “The instrument must“:

SR1. Report latency of pulling and pushing file blobs from/to container registry (Relates to use-cases UC1, UC2 in Section 4.2.2): This is the basic requirement for the container registry benchmark instrument. The instrument must send requests and measure the time until a response is received.

SR2. Report request throughput (UC1, UC2, UC3): The instrument must report request throughput, which is especially relevant for stress experiments. The request throughput is expressed in requests per second or requests per minute.

SR3. Generate layers of arbitrary size. (UC2): The instrument must generate synthetic workloads to replicate certain scenarios and test the registries with it.

SR4. Generate valid manifest for the generated layers. (UC2): In relation to SR3, the instrument must generate manifests that reference generated layers using their digests. Once the manifest is uploaded, the image is recognized by the registry and regular registry image operations can be performed on the generated layers and manifest.

SR5. Offer an option to specify the size and number of layers for image generation. (UC2): The instrument must support synthetic experiments by controlling the size and number of layers to be pushed or pulled. This requirement is directly connected to SR3 and SR4.

SR6. Run on most popular hosted container registries (UC1, UC3): The instrument must authenticate and run on the most popular registries. This is important because it ensures that the instrument will be relevant to the industry.

SR7. Store results in a well known data format (UC1, UC2, UC3): The instrument must record experiment results and store them in a well known data format (such as JSON, or CSV) to be plugged in the existing data processing pipelines.

SR8. Run real-world workloads (UC1, UC2, UC3): For instrument to be relevant both for academia and industry, it needs to read real-world workload traces and replay them to the registry under test.

SR9. Recover parts of experiment in case of failure. (UC1, UC2): The instrument must work in steps, and have an option to replay parts of experiments based on snapshots taken at prior steps.

DR1: Be deployable to any cloud platform (UC1, UC3): The instrument must run on any cloud platform, especially for the industry use case where it might be used for evaluating local performance.

DR2: Extract results back to local machine from the remote machine (UC1, UC2, UC3): The instrument must pipe the results back to the local machine from any type of deployment.

4.2.4 Non-Functional Requirements

We define non-functional requirements with specific thresholds for meeting the requirements. This will ensure basic standards to deliver the quality attributes mentioned.

NFR1. Easy to add new deployment options (Relates to use-case UC2 in Section 4.2.2): It should be possible to extend the instrument within one day for one person to deploy the instrument or parts of it on different research clusters. (*Extensibility*)

NFR2. Easy to support new registries (UC1, UC2): The instrument should work with new registries out of the box, or with modifications that take less than four hours for one software engineer. (*Extensibility, Interoperability*)

NFR3. Specify all experiment and deployment parameters inside one configuration file. (UC1): The instrument should be easy to configure and run, with all the necessary info present it should take less than half an hour to configure the experiments. (*Usability*)

NFR4. Offer an option to run an experiment iteratively across the list of specified registries and store the results together (UC1, UC2): The instrument should run a list of more than ten registries without any intervention between experiments. (*Usability*)

NFR5. Simple set of steps (UC1): Each step of the instrument can be explained with one short sentence. (*Simplicity*)

4.3 Design of an Architecture for Container Registry Benchmark

In this Section we present the design of the architecture for the instrument. The instrument architecture is composed of the *Harness Controller* and *Harness Tooling*.

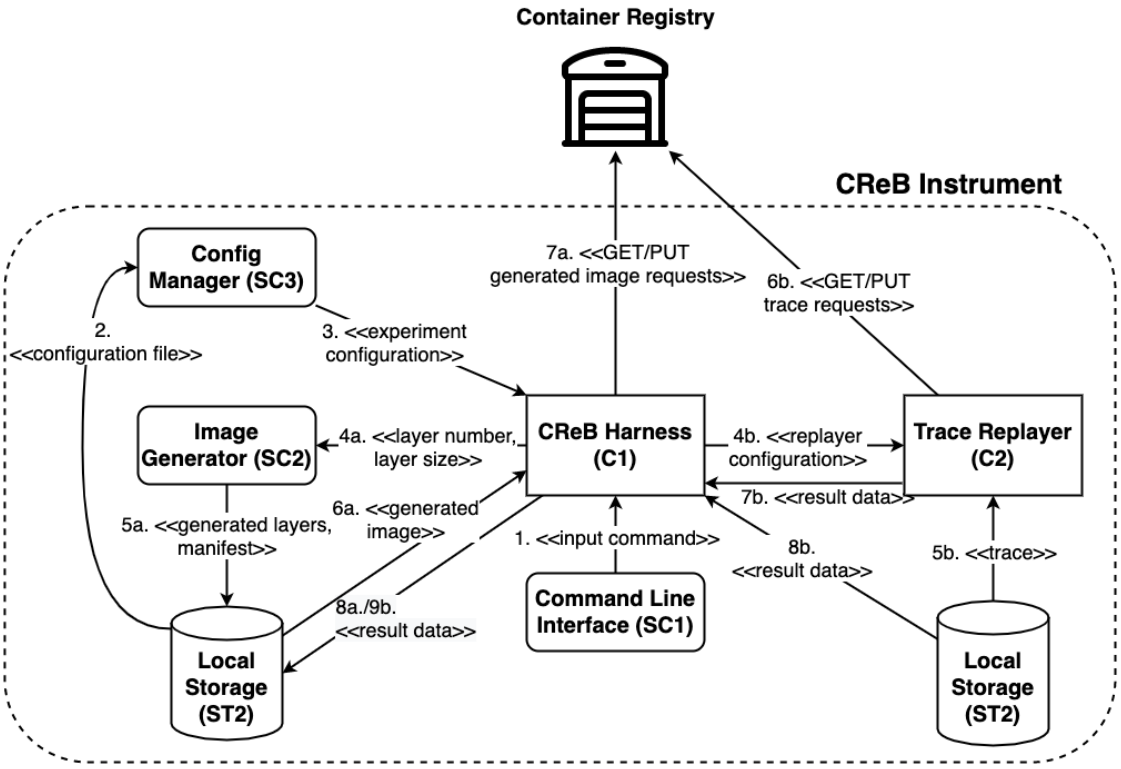

Figure 4.2: Architecture and data flow diagram for CReB.

We integrate the customized *Trace Replayer* tool in the instrument. This is an external tool developed by IBM and Virginia Tech that we discussed in Chapter 3.2.3. We adapted the Trace Replayer with support for Registry V2 API authentication and reading parameters from a common file rather than an internal `yaml` file. We analyse the adapted Trace Replayer in the Section 4.6.

The instrument is the component that receives parameters through configuration and the command line interface, and based on that invokes the right components to perform the task (experiment) specified. We have divided the instrument in two logical groups: Harness Controller and Harness Tooling. The high-level architecture of the benchmark instrument is displayed in the Figure 4.2. In this diagram we excluded subcomponents SC4 and SC5 for readibility purposes. All subcomponents are presented in component diagram with the dependencies between different packages as shown in the Figure 4.3.

4.3.1 Harness Controller

The Harness Controller (C1) is the component in the instrument that handles user inputs and orchestrates other instrument components to perform the user specified action. It is composed of the following components:

1. **Command Line Interface** (Component SC1 in Figure 4.2): Component that contains all available commands for the system. It orchestrates other components to perform the tasks that user specified. The specific commands are denoted as subcomponents because they contain experiment related functionality, and for push and pull commands, even experiment execution.

2. **Config Manager** (SC3): Component that reads the configuration yaml files, stores it in data structures that represent logical groupings of configuration per experiment, and provides a functionality for the components to fetch the required configuration. It is also used to create a new environment file per each iteration of the Trace Replayer experiment.

4.3.2 Harness Tooling

The tooling is the set of components that are invoked by the harness controller to perform user commands. The functionalities of these components are used to prepare and run experiments. The following components are part of the Harness Tooling:

1. **Image Generator (ImgGen)** (Component SC2 in Figure 4.2) Responsible for generating layers and manifests for synthetic workloads. It receives layer number and size parameters, and generates 0's and a sequence of 8 random bytes in the end of the file. This technique is adapted from the Trace Replayer layer and manifest generation functionality. Based on the configuration flag per registry the Image Generator also generates a valid manifest file that connects the generated layers into an image.

2. **Trace Replayer Orchestrator** (Component SC4 in Figure 4.3): Responsible for deploying Trace Replayer, either locally or on a DAS cluster, and orchestrating the experiment. Locally the Trace Replayer is orchestrated by executing python commands, and for the DAS cluster the Trace Replayer Orchestrator is first copying the Trace Replayer code via SSH, and then it is scheduling `slurm` jobs on the DAS cluster via SSH.

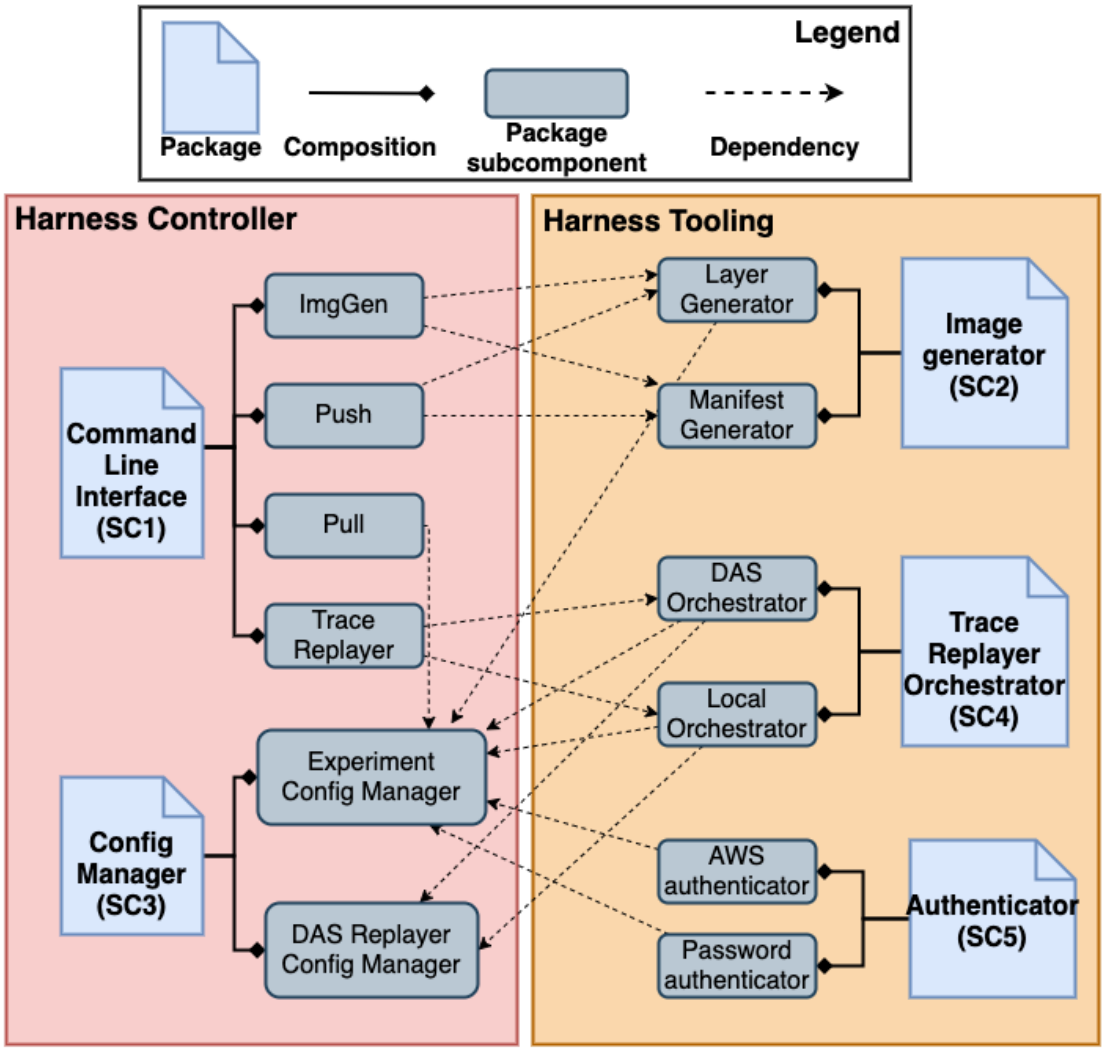

Figure 4.3: Component diagram for CReB.

3. **Authenticator** (Component SC5 in Figure 4.3): Responsible for obtaining authentication using cloud platform API's or using passwords supplied by the config manager. The authenticator supports any registry that supports registry V2 authentication via user supplied username and password, and it supports authentication with AWS ECR via AWS API.

4.3.3 Control Flow

In this subsection we describe the control flow based on three common commands: *pull*, *push*, and *trace-replayer*. In the case of *trace-replayer*, we try to abstract the deployment

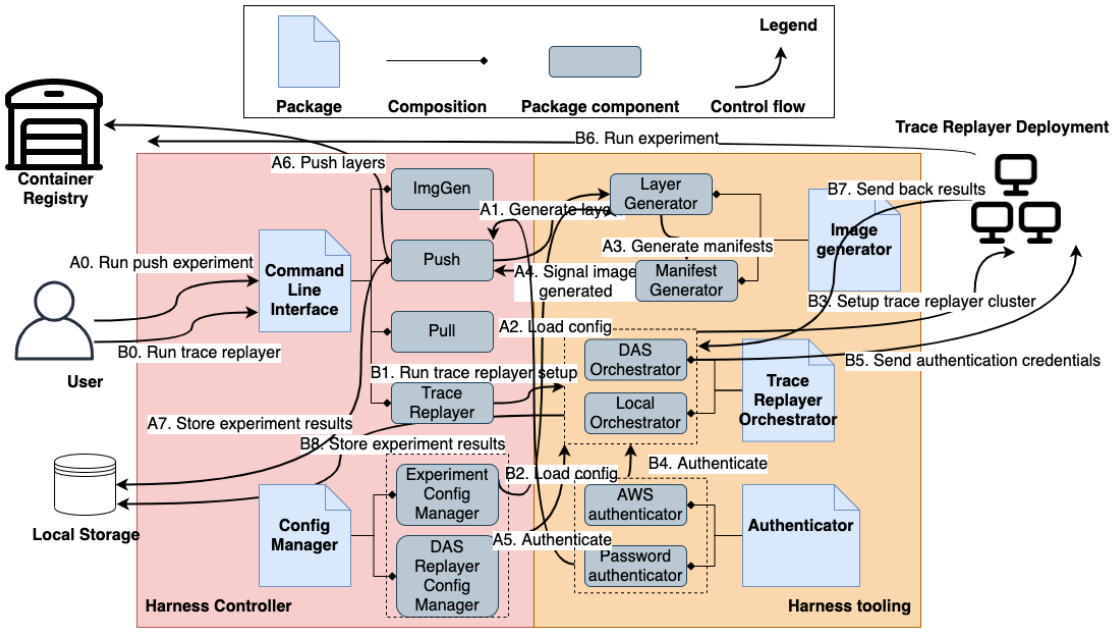

Figure 4.4: Control flow diagram for CReB.

and orchestration logic as we discuss them in-depth in Chapter 5.

4.3.3.1 Push

The push command is used to measure layer push latencies using a synthetic workload. Relevant configuration parameters for this experiment are *layer size*, *layer number*, registry configuration: *username*, *password*, *registry url*, and *repository*.

When we run the experiment push command, the layer generator is invoked to generate the layers for the experiment. The layer number and configuration are fetched from the configuration. The layers are computed, and the SHA256 digest of the file content is computed and stored as the name of the file. Then, the registry client gets initialized with the registry credentials that are fetched from the Config Manager. For each generated layer, the client checks if the layer already exists in the repository, and if it doesn't it initiates the layer push. The response time is measured for the combination of obtaining the layer push url from the registry and pushing the layer. This measurement is appended to the result list, and written to the result file. In the end based on the registry configuration it generates a manifest for the pushed layers.

The action of pushing all generated files is performed iteratively on the list of registries present in the configuration. In the end the result file contains the list of measurements

per registry.

4.3.3.2 Pull

The pull command is used to measure layer pull latencies using a synthetic workload. Relevant configuration parameters for this experiment are *generated layers directory path*, and registry configuration: *username*, *password*, *registry url*, and *repository*.

Pull experiment works in a similar fashion to the push experiment. The pull command depends on the push command, as it looks up the generated image (from the push command) to know which layers to pull. The pull command does not work on its own, as the layers need to exist in the registry to be pulled. Therefore push command needs to run before the pull command.

It obtains the list of digests in the directory with generated layers. For each layer it first does a lookup in the registry. If the file exists, it sends a GET request, and the layer download time is recorded. This measurement is stored in the list, and once all configured registries are tested, a csv file with measurements is produced.

4.3.3.3 Trace Replayer

The Trace Replayer command is used to run the experiment using real-world workloads. It orchestrates the external Trace Replayer tool (C2 in the Figure 4.2 to replay the sample of IBM registry traces. The relevant configuration parameters are trace replayer configuration: *trace path*, *client number*, *client thread number*, *mode (stress/delay)*, *number of threads in master prewarm*, *master run port*, *local result directory*, and if the experiment is run on the DAS cluster: *trace replayer local path*, *workload trace local path*, *DAS credentials*, *jumphost credentials*, *username*, *password*, *slurm job scripts*, and *client nodes*.

For each registry, we first obtain the username and the password for the registry V2 API authentication. Then, based on the CLI flag the harness runs the appropriate orchestrator. In case of the local deployment, it runs the client nodes on the local machine. Then the harness runs master node in the `prewarm` mode to generate and push the layers and manifests from the trace GET requests.

Then the harness runs the master node in run mode, where it instructs the clients to replay the requests from the trace. The results obtained by the master are then stored in the local machine.

The process is similar for the DAS deployment using `slurm` jobs scheduling and SSH. The Trace Replayer code is copied to DAS via `scp` command, together with the `requirements.txt`

file and the environment file that contains registry-specific configuration. These files are used to prepare the environment inside the DAS nodes. Then `slurm` jobs for the clients and the `master prewarm` are scheduled. The client nodes are defined in the configuration because `master run` needs to know the client addresses at startup. Once `master prewarm` is done pushing the layers to registry, the `master run` slurm job is scheduled by the Trace Replayer Orchestrator. `master run` interacts with the client nodes to replay the traces and stores the experiment results in DAS. Trace Replayer Orchestrator via `scp` command fetches the results and stores them on the local machine. When there are more registries in the experiment set the whole process is run again with different registry until all the registries are tested and then the results for all registries are copied to the host machine.

We describe the control flow of the Trace Replayer in-depth in the Section 4.6.

4.4 Analysis of the Design

In (5) cloud benchmark was described in a more specific manner: what it is, and how it relates to the general benchmark field. Three groups of requirements for a successful benchmark were defined: *General requirements* (GR), *Implementation Requirements* (IR), and *Workload Requirements* (WR). We go through the requirements in these groups and argue why this benchmark fulfills all of them.

GR1. Strong Target Audience: In Section 4.2.1 we describe our stakeholders, and with that we show that there is a need for our benchmark across industry and academia.

GR2. Relevant: In Section 4.2.3 based on the analysis of common registry operations we define the minimum operations that the benchmark should perform.

GR3. Economical: This highly depends on the registry pricing model, but we show in the Section 5.2.4 that for reasonably large workloads the cost doesn't explode.

GR4. Simple: In Section4.3 we present the design of the instrument, and in Section4.3.3 we provide in-depth description on the steps taken to run the experiment. The experiments ran are simply testing response times and number of requests per second that are handled by the registries. Simplicity is also a non-functional requirement (NFR5) we define in the requirements analysis.

IR1. Fair and Portable: This benchmark is testing registries using registry V2 API, which is implemented by all major hosted registries, and provides a common ground

for all registries. By running a benchmark with a set of registries we define a single experiment source which allows for choosing a fair location for all registries to reduce the influence of geographical distance on the latency.

IR2. Repeatable: In ideal conditions running the same trace sample for the Trace Replayer produces the same result with the same configuration. We can observe similar trend for the synthetic benchmarks. However depending on the benchmark location and noise in the network the experiment results might differ. We expand on this in Chapter 5.

IR3. Realistic and Comprehensive: The benchmark tests the performance of the most common operations that every registry performs: pulls and pushes of layers and manifests.

IR4. Configurable: In Sections 4.2.3 and 4.2.4 we expressed the requirements for configurability of the benchmark, especially SR3, SR5, and NFR3. Also in Section 4.3 we display how the Configuration Manager component contains the configuration for all functional components inside the Benchmark Harness.

WR1. Representativeness: Inclusion of both synthetic workload generation with layer and manifest generation, and Trace Replayer with the ability to replay IBM registry traces ensures that the set of interactions ranges from real-life scenarios to any type of synthetic scenarios.

WR2. Scalable: The scalability is ensured by the ability to vary number of clients that interact with the registry, but also to configure the number of registries that the benchmark interacts with.

WR3. Metric: The metrics the benchmark reports are simple and understandable: request latency, request throughput, and cost. We define metrics that the benchmark will report in Section 4.2.3.

4.5 Implementation Choices

In this section we discuss implementation choices, how they relate to design and how they accomplish the requirements.

4.5.1 Language Choices

We choose Go as the language of choice for the benchmark harness. The reason for choosing Go is that it supports wide range of features such as concurrency out of the box. This is ideal for managing multiple SSH connections and speeding up the experiment preparation. Go is also the language of choice in many cloud native systems, and it is a language in which the Docker toolset is written. As such, we decided it is mature enough to support our use case.

4.5.2 External Packages

We have chosen Cobra[1] as the CLI management tool. This package was chosen because it offers a rich set of features to manage flags, and it is a go-to tool for many Go CLI tools.

As the registry client we have chosen the Docker registry client developed by Heroku[2]. This registry client was chosen because it allows to break down actions such as `docker pull` and `docker push` into smaller actions, allowing us to use only specific API endpoints of the registry. This is similar to how the registry client in the Trace Replayer works. The registry client also offers methods to generate manifests and authenticate with real registries. We customized features of the client as we observed issues with certain registries.

4.5.3 Data Format Choices

We have decided to work with JSON and CSV as the two main data formats for the results. The configuration is managed by Yaml files. The choice for these three data formats was made because they are the most common data formats, and they are readable by the standard data analysis tools such as Python Pandas and R.

4.6 Trace Replayer

In this section we analyse another critical component of the instrument: the Trace Replayer[3]. This tool was created through the joint work of IBM and Virginia Tech University. It was presented as an addition to a collection of traces that were recorded for 75 days at IBM Cloud registry (19). These traces span 7 different availability zones, and provide insight into usage patterns of container registries. Each request in the trace is anonymized and of the following format:

[1] github.com/spf13/cobra
[2] github.com/heroku/docker-registry-client
[3] github.com/chalianwar/docker-performance

```
{
 "host": "79634854",
 "http.request.duration": 0.258506354,
 "http.request.method": "GET",
 "http.request.remoteaddr": "4053d99b",
 "http.request.uri": "v2/ca64f105/9f625072/manifests/817c8a39",
 "http.request.useragent": "docker/1.12.1 go/go1.6.3",
 "http.response.status": 200,
 "http.response.written": 502,
 "id": "a5449f269e",
 "timestamp": "2017-07-21T01:22:46.944Z"
}
```


The Trace Replayer offers an extensive set of configuration options to sample, run, and measure throughput and latency of the requests inside the trace for the specified registries. It is the tool that has been used to evaluate registry optimizations in (18, 22).

However, we have identified shortcomings of this tool for our use case. In this subsection we explain the general flow of the Trace Replayer tool (Section 4.6.1), then we analyse the technical issues we identified with the tool in Section 4.6.2. After that we discuss how we customized the Trace Replayer and argue about the validity of our implementation in Section 4.6.3.

4.6.1 General Control Flow

As briefly explained in (19) and in the documentation page in their Github repository, Trace Replayer `master` has three modes: *warmup*, *run* and *simulate*. Because we run the experiments for real-world registries, we consider only the first two modes.

The Trace Replayer consists of the master and clients. The master is responsible for preparing the experiment and sending request chunks to clients. Clients are the nodes with the preconfigured number of threads that replay the client trace chunk, and measure response times.

In the **warmup mode** only the master is used. The trace is processed and all the GET requests are extracted. The size of the GET request is read and a blob of the same size is generated. The generated blobs are then pushed to the registry. The returned digests are then mapped to the request URI and the mapping is stored in the intermediate file. This mapping will be used in the run mode.

In the **run mode**, the master and clients work together to perform the experiment. First the master reads the trace again, and assigns a chunk of requests to each client. The master appends metadata to the chunk and sends it to a client address. Clients read the chunks, and executes the request, either with delay or as fast as possible. In case of PUT requests, the blob is generated the same way as in the warmup mode. In case the file to be pushed is layer, first the client requests the url for PUT request from the registry and then executes the PUT request with the generated file. In case of the GET requests the client looks up the digest for the request URI in the mapping file and sends a GET request containing the digest.

After completing a GET or PUT request, the measurements are stored in the list which is sent to the master once all the requests are processed.

There are two request firing modes in the client that significantly influence experiment results: *delay*, and *stress* mode. In the delay mode, the clients are replicating the real trace request times by sleeping between requests. In case of the stress mode, the requests are fired immediately. This has an implication on the throughput measure, which is restricted for the delay mode. The throughput metric is more relevant in case of the stress mode as it measures the ability of a registry to cope with a much larger number of concurrent requests. However the delay mode allows for an experiment with a much larger workload without worrying about the registry request throttling as a DoS mitigation technique.

4.6.2 Issues with State-of-the-Art

Upon inspection of the code inside the Trace Replayer code repository, we have identified two types of issues: missing features (F), and bugs (B) that influence the results:

F1. Authentication: This is the most important feature that is missing, which makes this tool unable to connect to any production registry – all of which require authentication via Registry V2 API[1].

F2. Ability to specify multiple experiments: The Trace Replayer has an option in the configuration to specify multiple registries, but this works in a way that clients during one experiment distribute their requests across all registries.

F3. Generate manifests for pushed layers: In AWS ECR we observed a behavior where the registry does not allow pulling layers that are not referenced by any man-

[1] docs.docker.com/registry/spec/api/

ifest. In this case it is necessary to generate manifests that reference layers pushed in the warmup mode to fetch them in the run mode.

B1. Mislabeling manifest PUT requests: We have observed a bug that causes the PUT manifest requests to be labeled as GET manifest requests, creating a different sequence of requests than what is in the trace. The faulty behavior is observed during trace processing by the master in the run mode. Since for the manifests GET and PUT requests are the URI is the same, it means this URI is present in the intermediate mapping file. At the start of the run mode, the master clears request fields not needed for clients. Master labels every request that has URI present in the intermediate file as GET request. If a manifest has a GET and PUT requests in the trace sample, all requests will be labeled as GET due to this logic.

B2. Wrong request timestamps: This bug is specific to the delay mode. As we mentioned before, the client threads sleep for the time of the request delay, to replicate the temporal pattern of the trace. In case of the GET requests, we observed a bug where the time is recorded before sleep, and that time was added as the request timestamp. This means that if 3 clients had approximately 100 threads, we can observe 300 requests have very similar timestamp, even though the actual timestamps might be different by seconds or even minutes. This bug has a direct influence on the duration of the experiment as the duration is calculated as the largest sum of timestamp and duration. This is not significant for very large experiments as the threads still behave correctly, so the real and observed values have a small difference. However, in small sample experiments the reported throughput can be higher than the actual throughput.

4.6.3 Improvements

To accomplish our goals and deliver a valid benchmark instrument, we forked the Trace Replayer code repository [1] and made the following changes:

1. **Loading configuration from the environment file** (F1, F2): We added this feature to enable Trace Replayer configuration from the benchmark harness. This way we can orchestrate the Trace Replayer, re-run it multiple times, and change the configuration between the experiments to authenticate and replay the traces to a different registry.

[1] github.com/pgalic96/docker-performance

http.request.method	http.request.uri	timestamp (s)
GET	v2/08f6a309/36adcbc6/manifests/817c8a39	0.000000
GET	v2/ca64f105/9f625072/manifests/817c8a39	64.675000
PUT	v2/4715bf52/5da982be/manifests/f05bef95	100.963000
PUT	v2/08f6a309/36adcbc6/manifests/817c8a39	120.000000
GET	v2/4715bf52/5da982be/manifests/f05bef95	137.908000
GET	v2/4715bf52/f130e151/manifests/ef18771e	142.970000
GET	v2/a76c35b3/de29aee4/manifests/406251de	146.050000
GET	v2/a76c35b3/1f707197/manifests/406251de	152.437000

Table 4.1: Test sample, timestamps are normalized based on the first request timestamp.

2. **Add the authentication module** (F2): The underlying client[1] package for registry interaction in the Trace Replayer already supports authentication using HTTP registry V2 API. We have added it to the Trace Replayer.

3. **Add the manifest generation** (F3): As with the previous improvement, we used an existing feature of the underlying registry client to add manifests for the generated blobs.

4. **Fix mislabeling of the manifest PUT requests** (B1): We fixed this bug by using the request method field that was already present in the preprocessed requests.

5. **Fix the wrong request timestamp** (B2): As this bug was observed only in the GET request handling, we fixed it by observing the behavior of the PUT request handling, and fixing the wrong variable assignments.

4.6.4 Improvements Analysis

To validate that our bugs fixed the wrong behavior, we extracted a sample of 8 manifests, with 6 manifest GET requests and 2 manifest PUT requests. The trace sample timestamps are displayed in Table 4.1.

We normalized the timestamp such that every request shows the timestamp relative to the first request. Both of the manifest PUT requests have matching GET requests with the same URI.

We replayed the trace using the original Trace Replayer, and the customized version.

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

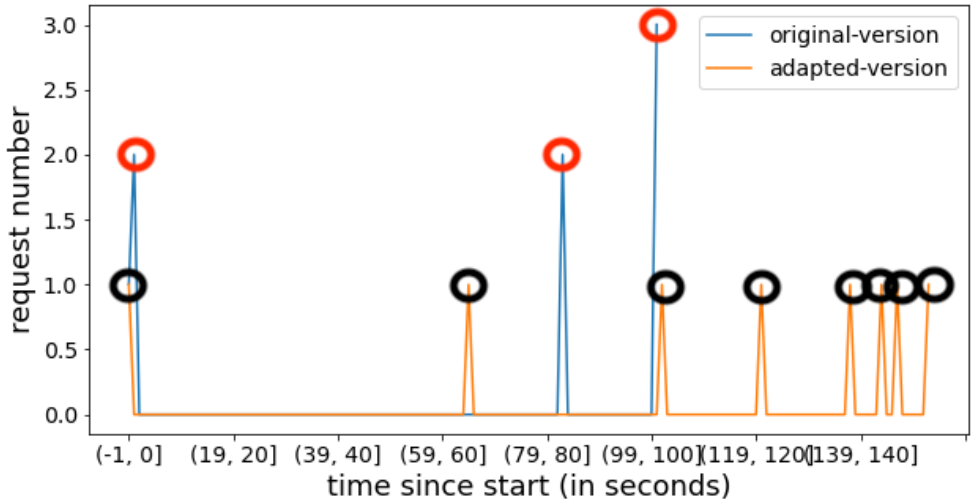

Figure 4.5: Timestamps of requests.

As shown in Figure 4.5, there are three requests that have a timestamp between 0 and 1 (one is exactly at 0 because of normalization). That does not correspond to the real trace data that the second request has delay of 64 seconds. The requests come at spikes, and these spikes correspond to real execution of a certain request after delay. However we see that the last spike is at 100 seconds, and the total duration of the experiment (largest sum of timestamp and request duration) is 101 second as seen in the Table 4.2. These spikes are highlighted with red circles in Figure 4.5. This definitely does not match the trace data as in Table 4.1 the last request has a delay of 150 seconds – so the duration and throughput are miscalculated.

In our customized version of the Trace Replayer, we see eight spikes marked by black circles in the Figure 4.5 corresponding to the firing of each request, where the timestamps match the delay. We can also see that the last one is fired at 152 seconds as indicated in Table 4.3, so it matches the correct throughput and duration that is expected when looking at the trace sample timestamps.

Observing two results we also observe the faulty behavior B1 – mislabeling PUT requests as GET. We see in the Table 4.2 that the original Trace Replayer results now contain 8 GET requests, while the customized one in Table 4.3 resembles the original count. The latency for the push is around twice the latency for GET requests, so we conclude that the two observed bugs influence the experiment results.

timestamp (s)	duration (s)	document_type	method	size
0.000000	0.358306	manifest	GET	535B
0.001116	0.366941	manifest	GET	535B
0.002596	0.393781	manifest	GET	2851B
82.200649	0.367080	manifest	GET	519B
82.202092	0.392813	manifest	GET	2207B
100.926853	0.384099	manifest	GET	513B
100.927618	0.389787	manifest	GET	519B
100.928915	0.372428	manifest	GET	1800B

Table 4.2: Original Trace Replayer results

timestamp (s)	duration (s)	document_type	method	size
0.000000	0.368509	manifest	GET	536B
64.735652	0.417869	manifest	GET	513B
101.058650	0.707077	manifest	PUT	507B
120.095447	0.811669	manifest	PUT	524B
137.984569	0.381654	manifest	GET	518B
143.065468	0.360687	manifest	GET	2206B
146.145522	0.385331	manifest	GET	2850B
152.533439	0.356151	manifest	GET	1800B

Table 4.3: Customized Trace Replayer results.

4.7 Cost Analysis Model

The costs incurred from container registry usage are dependent on a number of factors. Every registry is offering a set of features that drive the costs up if used, and specifically for cloud provider container registries, a big factor is the registry and source location. We can partition the costs in two general categories: *network egress costs*, and *storage costs*. The cloud providers offer the same or similar pricing model for all storage services. That means that the storage and egress costs also depend on the registry region.

The main pricing model difference between Azure Container Registry (ACR), Elastic Container Registry (ECR), and Google Container Registry (ECR) is that ACR charges an instance subscription cost per day. This cost depends on the registry tier. However, ACR also offers a certain amount of storage per registry tier, and the user pays extra fees for the storage only if it exceeds the quota included in the subscription. GCR on the other hand structures the storage costs based on the underlying GCloud Bucket Storage costs. The

costs for this storage are based on two factors: size and number of operations. The number of storage operations is the defining factor, as the storage costs itself are considerably lower than the storage costs for other registries (0.02$ for a Standard Bucket compared to 0.1$ and 0.09$ for ECR and ACR respectively). For the purpose of the cost forecast, we take the Standard Bucket and the number of operations to be 10 1st tier operations (GET,PUT) per GB, which results in the maximum of 2 million operations per 200 TB.

Services such as Quay.io, Gitlab, or Docker Hub have a different pricing model. Their offering is based on a fixed monthly fee, which differs based on the tier selected. The tiers do not differ based on the storage and egress, but on the number of private repositories. This makes these options suitable for small businesses, and single users, as it offers a great flexibility, but as we saw in the experiment results, a worse performance which is not usually a priority for small deployments.

When computing costs for our experiments, we perform calculations in the following way. For every experiment we take the conservative estimation of storage size equal to the egress size. This is due to the inefficiency in the trace replayer where all GET files are pushed to the registry in the `prewarm` mode. We improved this behavior by adding a check in the master node, which is considered per each thread, however there is no coordination between threads to determine whether the file was pushed by another thread, so in the worst case the pushed amount is close to the total egress of the trace sample. We also compute costs based on the pricing tiers, not on the actual costs incurred when running these experiments. The reason for this is that for many of these experiments we used the trial credits received by creating new cloud platform accounts, therefore resulting in much lower costs incurred than the expected.

4. DESIGN AND IMPLEMENTATION OF THE CONTAINER REGISTRY BENCHMARK

5

Experimental Evaluation

In this chapter we present the design of experiments that were used to gain insight into performance differences between registries. We present the sets of platforms that we used for each experiment, which is one of the parameters we varied in experiment configuration. Furthermore we describe the workloads we used for the different experiments, which is another experiment configuration parameter. We present experimental setup and the experiment results, which we use to answer the final research question **RQ3**: *What are the cost and performance differences between production-grade, hosted registry platforms?*

5.1 Experiment Design

We identified a set of experiment parameters we vary to create experiments. Those parameters are *workload*, *trace replayer mode*, *registry set*, and *periodicity*.

5.1.1 Platform Selection

When selecting the registry sets, we aim for a representative selection of real-world, production-grade hosted services. In Chapter 3 we survey the state-of-the-art in the industry. From the registries of interest we identify there, we construct a registry set for each experiment, achieving a different goal for each experiment. The selected registry sets are presented in Table 5.2.

For the experiment E1 we select a wide set of registries, however with the focus on the cloud platform registries. The reasoning for this choice is that we want to gain more insight into performance differences and similarities across public and private registry platforms, since latency is largely influenced by the geographical distances. As for the region choice, we focus mainly on data centers in Europe and North America, as we identified these data

ID	Experiment Name	Workload	Mode	Registry #	Period
E1	Small Size - Real Workload	W1	Delay, Stress	16	once
E2	Large Size - Real Workload	W2	Delay	5	once
E3	Synthetic Long-Running	W3	-	16	every 6 hours, 2 months

Table 5.1: Experiment configurations. Table 5.2 describes registry sets. Table 5.3 describes workload types W1 through W3. Section 4.6 describes trace replayer modes.

centers as the busiest and most relevant to the industry. We also include IBM Container Registry Dallas as a control registry as it is the original source of the traces. We include Docker Hub as its the most widely used public registry today. For the Azure Container Registry, we want to investigate the performance difference between registry tiers (Basic, Standard, Premium) offered.

It is important to note that for any experiment running in the *delay* mode, we also display the original latencies recorded in the trace itself (identified by `ibm-trace-data-dallas`).

For the experiment E2, increasing the size of the workload caused the set of registries to shrink. This was mainly due to the time and cost constraints. We preserved the focus on the public cloud platform registries, with specific focus on AWS Elastic Container Registry, as AWS is the market leader in public cloud computing. We also included Azure Container Registry and Google Container Registry as the services from the second and third most popular cloud provider.

For the experiment E3, we select a wide range of container registries due to a small workload size, with a larger focus on AWS registries but also including more different platforms. We included more hosted registries that we identified in Chapter 3. The reason for this is that the small workload and periodicity allowed us to still keep the usage within free limits of these registry platforms. For that reason, we included Gitlab Registry, Docker Hub, and Quay.io.

5.1.2 Trace Selection

For the two experiments (E1, E2) that use a real-world workload we selected two different samples of workload traces from IBM production registry trace collection. Both samples are selected from the Dallas region, as this is the busiest registry with 62% of all egress and 55% of the total trace data size(19). We now provide the reasoning for the sample selection, together with analysis of the traces at hand. The trace characteristics are presented in Table 5.3. We present two real-world workloads, and one synthetic workload.

ID	Platform	Region	Type	Experiment
ibm-registry-dallas	IBM CR	Dallas	-	E1
ecr-eu-west	ECR	Ireland	-	E1, E2
ecr-eu-central	ECR	Frankfurt	-	E1, E2
ecr-us-west	ECR	North California	-	E1, E2, E3
ecr-us-east-1	ECR	North Virginia	-	E1, E2, E3
ecr-us-east-2	ECR	Ohio	-	E3
ecr-ap-southeast-1	ECR	Singapore	-	E3
ecr-ap-southeast-2	ECR	Sidney	-	E3
gcr-eu	GCR	Europe	-	E1, E2, E3
gcr-us	GCR	US	-	E1, E3
gcr-asia	GCR	Asia	-	E1, E3
acr-premium-eu-west	ACR	Europe West	Premium	E1, E3
acr-standard-eu-west	ACR	Europe West	Standard	E1, E2, E3
acr-basic-eu-west	ACR	Europe West	Basic	E1, E3
acr-standard-us-east	ACR	US East	Standard	E1
acr-standard-us-west	ACR	US West	Standard	E1
acr-standard-eu-central	ACR	Germany West-Central	Standard	E1
acr-premium-eu-central	ACR	Germany West-Central	Premium	E1
docker-hub	Docker Hub	-	Pro	E1, E3
gitlab	Gitlab CR	-	-	E3
quay	Quay.io	-	Free	E3

Table 5.2: Mapping of registries to experiments. The identifiers are used in plots of the results. The type of the registry indicates a subscription tier, if there exists one. Experiment identifiers indicate in which experiments the registry is used.

5.1.2.1 Experiment E1 Workload Analysis

For the experiment E1 we have selected a sample of 405 requests and approximately 3.5 GB of size. The sampling process was the following: we have computed the average number of requests, average egress and average ingress size per day. Then, we have scanned the trace for the day in which none of the three parameter values is further than 10% from the average. From there, we selected a sample of the appropriate size (3.5GB). Keeping a smaller sample size allowed us to include the Trace Replayer stress mode, as much larger workload would result in heavy request throttling.

In Figure 5.1 we show the comparison of total GET and PUT requests in the request sample. This large disparity is similar to the one observed for the whole trace collection

ID	Type	Date-Region	Request #	Egress Size	Timespan
W1	Real-World	21/7-Dallas	405	3.5 GB	3.6 minutes
W2	Real-World	17/8-Dallas	8,250	82 GB	1 hour
W3	Synthetic	-	20	10 MB	-

Table 5.3: Experiments workload characteristics.

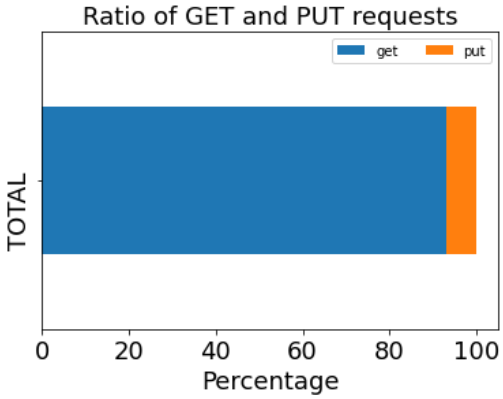

Figure 5.1: Workload W1 GET and PUT request ratio.

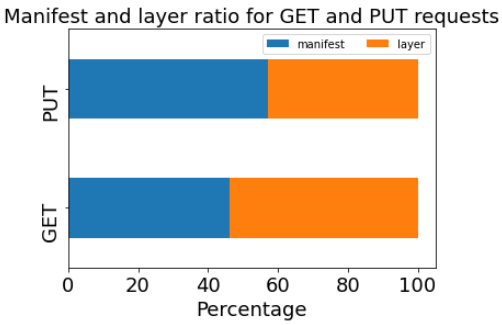

Figure 5.2: Workload W1 Manifest and layer ratio for GET and PUT requests.

(19). This can explain why container registries are optimized for reads. It also explains why cloud provider registries charge egress only.

It is important to note that this means that for small samples such as this one, the number of PUT requests is very low – 28 to be precise.

In Figure 5.2 we can observe the representation of layers and manifest in the sample, for both PUT and GET requests. For GET requests the number of layer requests is larger than the number of manifest requests. This can be explained by the one-to-many relation between a manifest and layers. There are also manifest requests that do not result in subsequent layer pulls, if the client already has all the layers mentioned in the manifest.

In case of image pushes, the manifest is pushed after the layers. However, many manifests are pushed as a part of the re-tagging of existing images for development or deployment purposes. This explains why there is a slightly larger number of manifest PUT requests than layer PUT requests in the trace sample. An in-depth analysis of the full trace is present in (19).

In Figure 5.3 we can see a large difference in size between two types of blobs that are in GET requests. This is the key characteristic of the registry workload. We see that 90% of manifest files are less than 30KB in size, while the layer sizes vary much more with 90% of

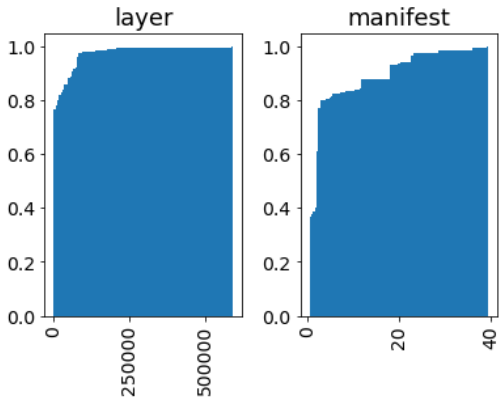

Figure 5.3: Workload W1 CDF of GET blob request sizes (KB).

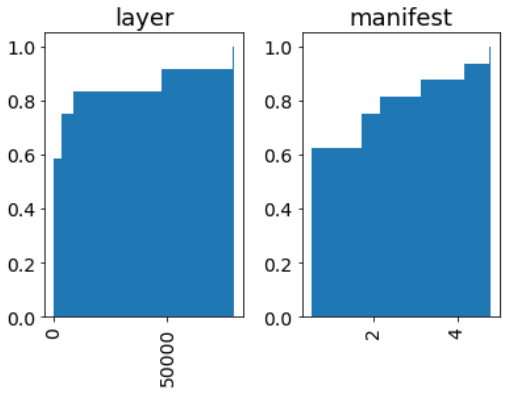

Figure 5.4: Workload W1 CDF of PUT blob request sizes (KB).

them less than 200MB. We observe layers of size up to 600MB. The same file size disparity can be observed in Figure 5.4 as well for the PUT requests. However, due to the small number of PUT requests in this sample the CDF has a stair shape.

5.1.2.2 Experiment E2 Workload Analysis

For the large size experiment, we select a one hour trace sample. We change our trace sampling policy from W1 to avoid spikes of the number of requests sent that our client nodes wouldn't be able to replicate. To do this, we use a sliding window algorithm. The size of the window is 1 hour. For each window, we compute the average throughput per minute. Then, we compute the median, the 5th, and the 95th percentile. We select a window where the distance between 5th and 95th percentile to median is less or equal to 50% of the median value. This value is conservative, we hypothesize the percentile distance could be reduced even further.

We selected the median value due to the properties of the traces that were also observed in (19). There are large temporal patterns observed, both for the week and for the day, so averaging the load parameters introduces the risk of load spikes and dips.

Regarding the trace sample characteristics, we can see a similar pattern as with the W1 sample. However, the differences at scale are much larger. In Figure 5.7 the percentage of PUT requests in the sample has shrinked. We can also see that the trends with manifest and layer ratios for GET and PUT requests have continued and the differences are much larger in this sample.

In Figure 5.7 we can observe a large variation of layer sizes. Most of the requests are still much smaller than 1GB, but the distribution is skewed because of a few very large layers

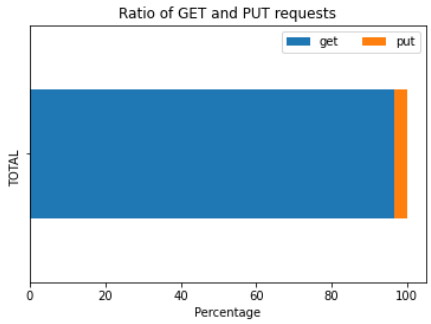

Figure 5.5: Workload W2 GET and PUT request ratio.

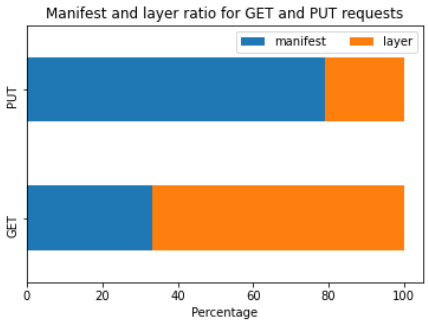

Figure 5.6: Workload W2 Manifest and layer ratio for GET and PUT requests.

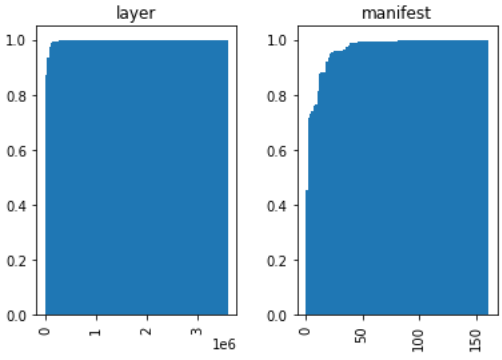

Figure 5.7: Workload W2 CDF of GET blob request sizes (KB).

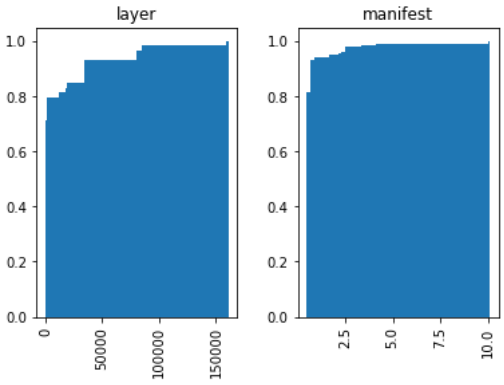

Figure 5.8: Workload W2 CDF of PUT blob request sizes (KB).

(3GB or larger). For the manifests we can still see that 90% of the blob sizes are smaller than 50KB.

Even though the percentage of PUT requests has shrinked, due to the much larger number of PUT requests we can get more insight from the CDF of PUT request sizes from Figure 5.8. We see the layer sizes vary similar to GET requests, however we observe smaller variances.

5.1.2.3 Experiment E3 Workload Analysis

For the long-running experiment, we generate a synthetic workload of 10 layers, each of size 1MB. This gives us a relatively small workload, but due to the long-running nature the total egress and ingress is close to 2 GB per registry. We initially planned to not include manifests, but we encountered a pull restriction in AWS Elastic Container Registry with layers that are not associated to any manifest. To circumvent this restriction, we generat

and push manifests. However, the manifests were not included in the measurements.

5.1.3 Deployment

We describe two separate experiment deployments, one for the experiments that include trace replayer (5.2.1, 5.2.2), and one for the long-running experiment (5.2.3).

We used the DAS cluster (37) to deploy the Trace Replayer component for the real world experiments. We discussed the functionality of the Trace Replayer in Chapter 4. We use the DAS Orchestrator component from harness for deployment of the Trace Replayer to the DAS cluster. DAS nodes are located in Amsterdam, Netherlands. The client nodes are connected to the regular internet via 1 Gbps Ethernet interface through SURFnet educational connectivity institution, which is directly connected to the Amsterdam Internet Exchange (A-IX), one of the largest global internet exchanges.

We explained how harness runs and obtains results from the trace replayer experiment in Section 4.6.1. Specifically for this experiment deployment, the deployment is orchestrated using Slurm workload manager [1]. We have deployed the Trace Replayer client at 3 DAS nodes. We have used one node for master warmup, and one for master run mode. This deployment model was used for all experiments that replayed IBM traces. The deployment diagram is presented in Figure 5.9.

The same DAS configuration is used for both modes in experiment E1. In the case of the stress experiment there is no delays imposed by the trace replayer, so the requests get fired whenever the client thread is done with the prior request. In the experiment deployment we configured 3 clients with 100 threads each. This means that there can be a maximum of 300 requests sent at the same time.

For the synthetic long-running experiment, we have setup a Digital Ocean droplet in the AMS3 (Amsterdam) region with 2GB memory and 60GB of storage. There we setup a `cron` job which runs every 6 hours, firstly pushing layers to the registry, and then pulling the same layers.

In the large scale experiment we test using the delay trace replayer mode. There are multiple reasons for this. First of all, for this workload size (82 gigabytes) in stress mode the request count per minute quickly exceeds throttling limits for ECR, ACR, and GCR registries, making it impossible to obtain relevant results which do not contain a high percentage of failed requests. Secondly, the aim of the large scale experiment is to replicate the real-world environment as closely as possible. The delay mode comes closer to this goal

[1] https://slurm.schedmd.com/

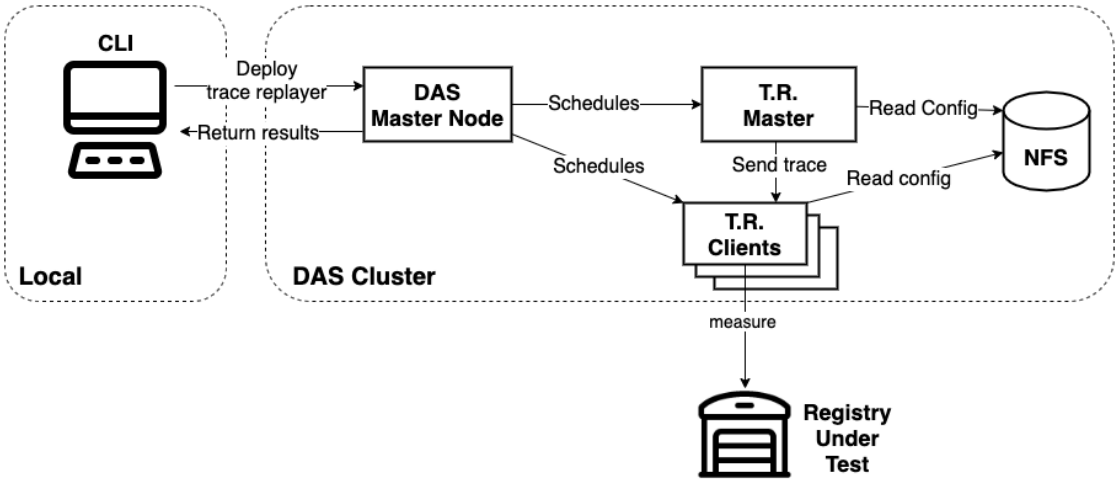

Figure 5.9: Trace Replayer deployment on the DAS cluster.

by imitating real world delays between requests. We haven not experienced issues with running this experiments in the delay mode, as the number of concurrent requests does not activate rate limiting.

By running the long-running experiment we wanted to investigate temporal patterns in the registry performance, and gain insight into performance gains or degradations over time.

As the experiment featured 16 registries, to provide more readable results we group the registries into three categories: *European regional registries*, *US/Asian-Pacific regional registries*, and *public registries*. First and second category concern private registries offered on cloud platforms, while the third category concerns public registries where it is not possible to select a region.

5.2 Experimental Results and Discussion

In this section we present the results of our experiments. We have used Python, specifically Pandas and Seaborn packages to clean and visualize the resulting sets of data. The resulting data sets and Jupyter notebooks are publicly available at Github and Zenodo. We discuss each visualized experiment result and note the most interesting patterns in the results.

5.2.1 Small Size Real-world Experiment Results

In this experiment we evaluated the registries using both Trace Replayer modes. This yields two completely different sets of results as in the delay mode the throughput is capped by

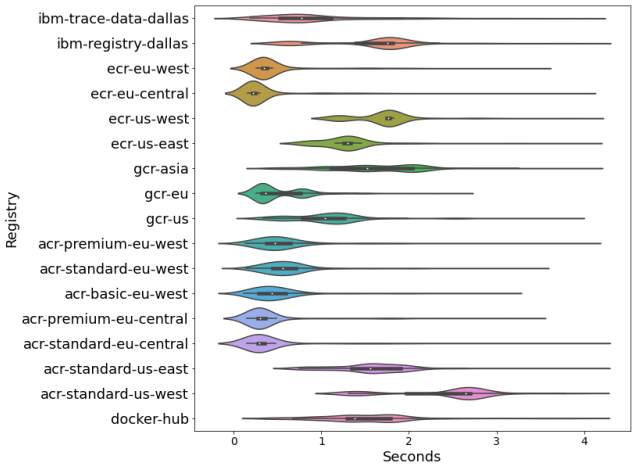

Figure 5.10: E1 Delay: Violin plot of GET request latency measurements for all blob types. Table 5.2 describes registry under test per ID.

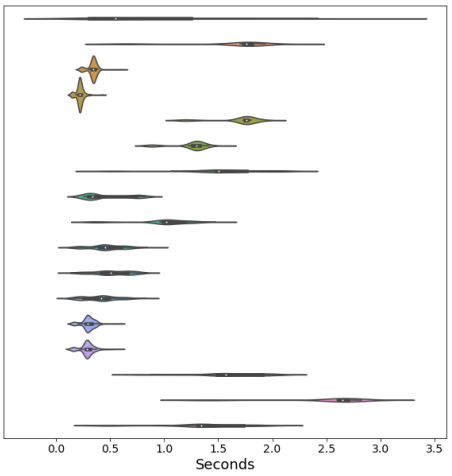

Figure 5.11: E1 Delay: Violin plot for the manifest GET latency.

the delay restriction between requests.

5.2.1.1 Delay Mode Results

We report results in the following categories: *total GET latency*, *layer and manifest GET latency*, *total push latency*, *layer and manifest push latency*, and *average throughput*. **Total GET latency** is the latency for all GET requests, expressed in seconds. **Layer and manifest GET latency** is the GET latency for each type of files fetched from the registry, expressed in seconds. **Total push latency** is the latency for all push operations, while **layer and manifest push latency** is the latency for each type of file pushed to the registry. **Average throughput** is the metric expressed in number of requests handled per second. In Figure 5.10 the visualized results are shown for all GET requests. The results are very different from platform to platform, and even within registries of the same platform, and even geographically close regions. We excluded all outlier latencies larger than 10 seconds to improve readability of the plot.

The two main points of analysis are general performance, and performance variance. As expected, the registries from the regions that are geographically closer to the test machine are performing the best. ECR registries in Frankfurt and Ireland, and ACR Germany West-Central have the lowest performance variance and observed latencies. For ACR EU West the performance variance is larger, as indicated by the wider box-and-whisker plot and more spread out violin. GCR EU registry also has a larger performance variance, although the general performance is similar to the one from ACR and GCR.

The latencies recorded in the IBM trace data (identified by `ibm-trace-data`) are lower than the latencies we measured for IBM Registry Dallas, however that can be attributed to geographical vicinity of the requests to the Dallas registry. We can also observe that the performance of Docker Hub registry has a large variance, and it is comparable to the latencies observed in the US regions, which reveals information about the location of the Docker Hub registry deployment.

Generally ECR registries have low variance, even in geographically distant regions, which cannot always be observed for other platform registries. ACR US east has a very large variance, which can also be observed for the GCR US and Asia. However, GCR US generally outperforms other US registries in latency. Still, the ECR registries in European regions, along with the ACR registries in the geographically similar regions have the lowest latencies and performance variance. We do not see significant differences in the latency between ACR tiers in the same region.

The manifest-only latency measurements visualization is displayed in Figure 5.11. We see the latency distributions are quite different than for the layers. We conclude again that ECR and ACR (Europe) have the lowest performance variance. On the other hand, ACR US East has the largest performance variance. We can again observe a large latency variance in the trace data, which is explained by requests sources being on different geographical locations, which in turn results in large latency differences even for such small files.

IBM Registry Dallas performs similar to the US West registries, despite its geographically closer than the US West. We can also see that ECR West actually outperforms other US West regions, and even has similar performance to the ACR US East, with a smaller performance variance.

For the push latency results, despite the small amount of samples we can still get insight from the Figure 5.12. ACR Germany West-Central has the lowest performance variability. In this graph we can observe quite a lot of long tails, which tells us that there are large layers that skew the violin plot. We can again note that Docker Hub and IBM Registry Dallas have the largest performance variability, even though they are not necessary geographically furthest from the experiment host.

Finally, in the Figure 5.13 we can see the average request throughput per registry, expressed in requests per second. Here we can observe the restriction enforced by the trace replayer mode. However, all registries outperform the original trace measurements, which can be explained by the fact that we measured the throughput by the arrival time, which

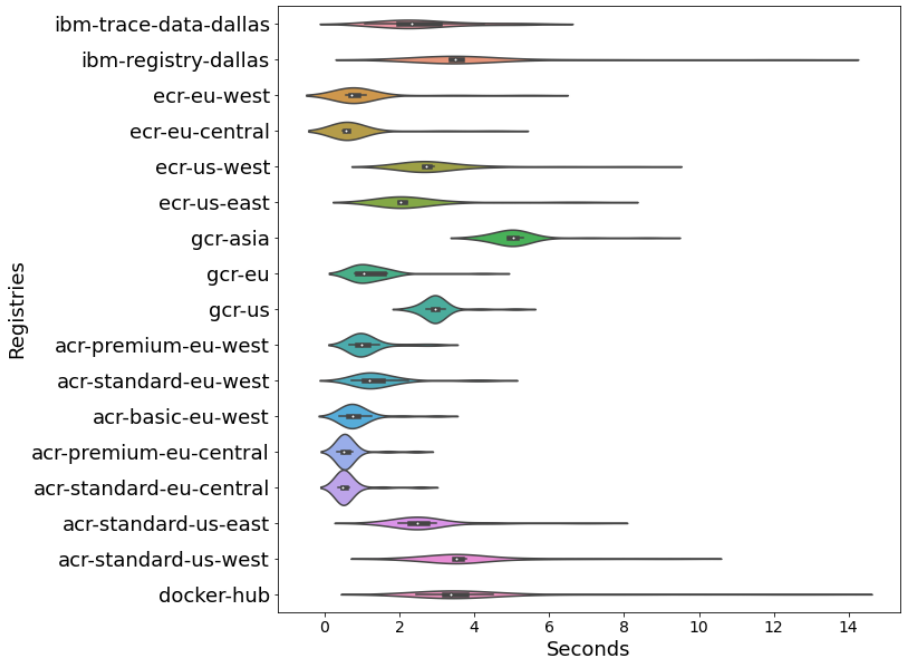

Figure 5.12: E1 Delay: Violin plot for the total push latency.

was worse for the trace replayer data, and we observed a lot of outliers for the trace replayer data (which can be seen in the performance variability).

Apart from that we can see a bit worse performance for ACR US West and US East, which can also be attributed to large outliers, that extended the total time of the experiment. However, for most of the other experiments they reached an upper limit which is enforced by the request delay cap.

5.2.1.2 Stress Mode Results

This experiment is aimed at testing throughput of the registries and testing the throttling mechanisms of registries. Our assumption at first was that 300 requests per second should not cause any throttling, however the results we observed were different. However we have not observed throttling across a wide range of tested registries, therefore we did not adjust the number of clients and threads. Throttling, or rate limiting is one of the key techniques of the container registry, as it determines to what extent the registry can be used in large deployments where thousands of nodes might pull images at the same time. This experiment is useful for testing throughput guarantees of ACR registry tiers.

We analyze the results in this experiment based on the following metrics: *all files GET and push latency*, *layer and manifest get latency*, *throughput per second*, and *average*

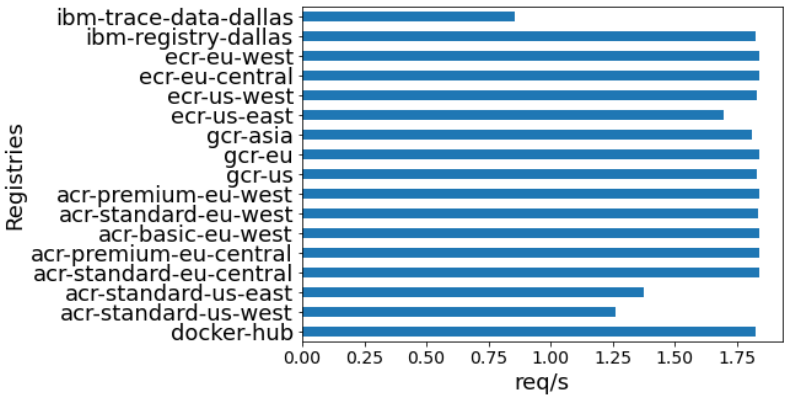

Figure 5.13: E1 Delay: Average request throughput.

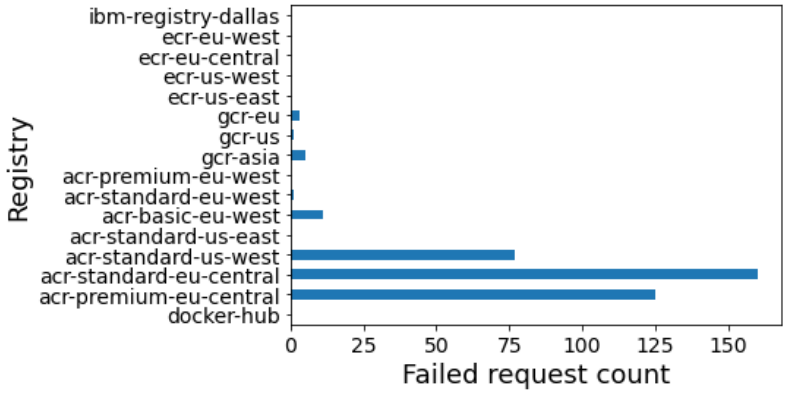

Figure 5.14: E1 Stress: Failed request count.

throughput.

First, in Figure 5.14 we can observe the failed request count per registry in this experiment. We see only the ACR registries observe high number of failed requests, with specifically ACR Germany West-Central having the most. All the requests are met with the response `429: Too many requests`. This indicates throttling of requests, as most of the requests fail in the first seconds of the trace replayer run mode, when the number of sent out requests reaches the 300 requests limit. We have contacted Microsoft Azure team to clarify this behavior, as the throttling is observed only for Germany West-Central and ACR US-West regions. Our assumption was that this is because the datacenters in question might be smaller than other datacenters, therefore their rate limiting thresholds are lower. The Azure team acknowledge these results on their end, and confirmed our assump-

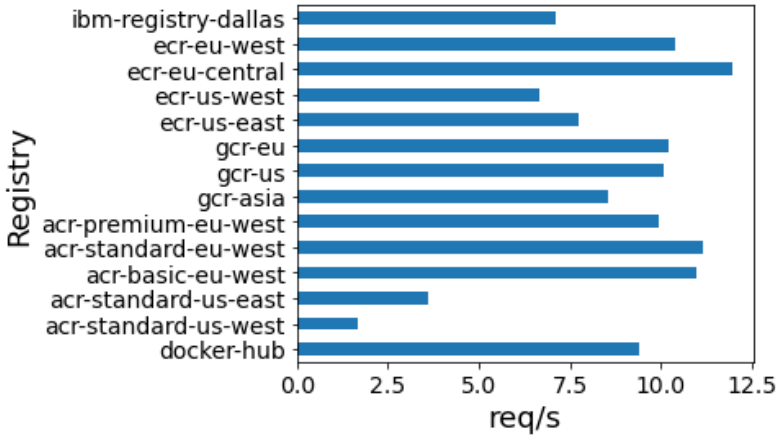

Figure 5.15: E1 Stress: Average throughput.

tions. The ACR uses Nginx [1] for DDoS [2] protection, but also as protection from customers misusing ACR (for example, frequent polling of registry instead of moving the logic client side). In ACR Germany West-Central there is in total 3 Nginx nodes proxying the requests. These 3 nodes get overwhelmed with requests from the same IP Addresses, and start rejecting requests. Since in EU West there is 40 Nginx nodes in front of the registry, the rate limiting threshold is much higher (around 900 requests per second). However, the current behavior still satisfies the minimum throughput guarantees in the SLA for premium tier (166 requests / second). However, this does indicate that users need to choose the region in ACR carefully if their typical workload might be close to or above guarantees provided by the SLA.

In our experiment, due to the large number of failed requests (more than a third), we chose to exclude Germany West-Central ACR region because the results reported were not realistic. A lot of large pulls and pushes failed, therefore causing reported results to be better than if all requests succeeded.

In Figure 5.15 we present results for the average throughput reported per registry. As we can see, the results are quite different than the delay experiment throughput results in Figure 5.13. We can still see the influence of the geographic distance on the results, with registries in European regions having a higher throughput. However, we can also observe some deviations from this pattern. For example, GCR US has a slightly higher average

[1] nginx.com

[2] Distributed denial of service is a type of coordinated network attack where many clients are used for sending large number of requests for the purpose of making the service unavailable.

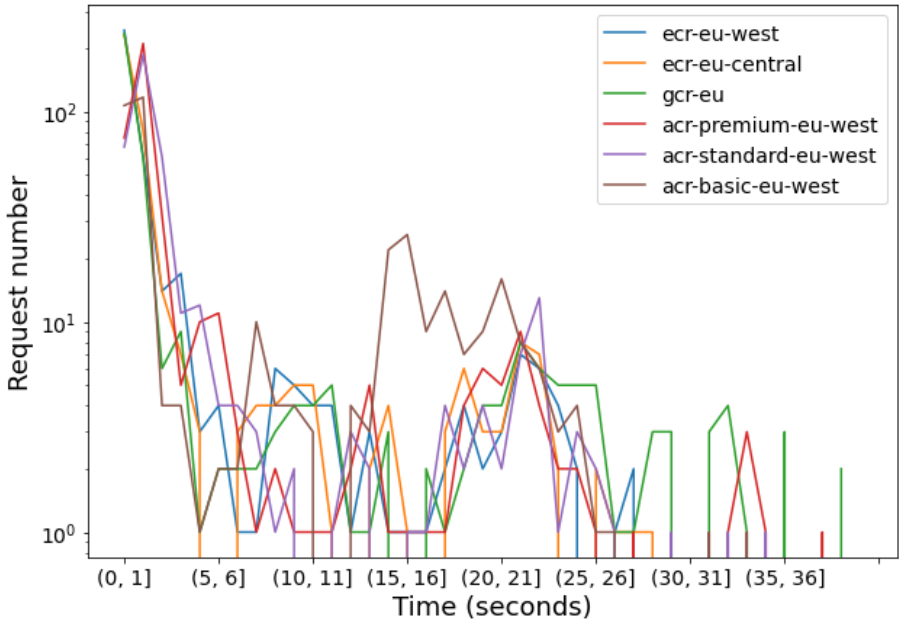

Figure 5.16: E1 Stress: Throughput per second – Europe.

throughput reported than ACR Premium West. Even when comparing average throughput between GCR EU and GCR US there is no significant throughput difference.

On the other hand, ACR US regions reported a very low average throughput. This is mainly due to outlier requests, which took very long time to complete (more than 40 seconds). We can also see that Docker Hub has a very high reported throughput. This shows that even though the registry is not regional (as reported by the latencies in the delay experiment), it is still optimized for handling a large number of requests. We used the paid version of Docker Hub, as this experiment would not be possible with Docker Hub free tier restrictions introduced in November 2020 [1].

Another observation we make is regarding throughput differences between Basic, Standard, and Premium ACR tiers. It appears from this Figure that ACR Premium performs worse than Basic and Standard tier. However, the average throughput is particularly sensitive to outliers when the total experiment time is relatively low (40 seconds), which can be caused by network conditions as well. To better investigate these behaviors, we computed the throughput per second metric, which can be seen in Figures 5.16, 5.17, and 5.18. We are showing only the first 50 seconds, for the readability purposes.

In the Figure 5.16 we highlighted all registries from European regions. We see most of the registries start with a high initial throughput, which correspond to the first second

[1] docker.com/increase-rate-limits

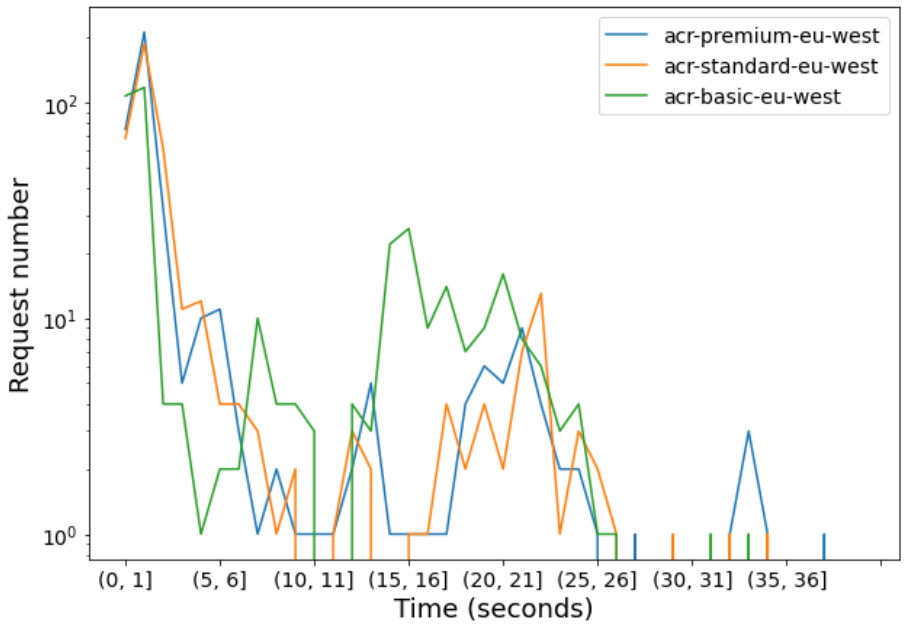

Figure 5.17: E1 Stress: Throughput per second – ACR EU-West.

when the 300 request per second firing maximum limit is reached. We can also see for all registries that there is a pattern of a spike around 20 second mark.

In Figure 5.17 we focus on the ACR tier analysis. We see Premium tier does start with the highest initial throughput which confirms the higher stated throughput for this tier. The Basic tier line has more spikes observed, which indicates requests queuing. Even though the workload request rate exceeds all tier guarantees, the requests still succeed compared to what we observed for other regions in Figure 5.14. This indeed confirms that the throughput guarantees are not linked to the rate limiting imposed, more precisely Nginx nodes in front of registries are not coupled with the registry throughput guarantees, but they are merely used for DDoS protection.

Finally, in the Figure 5.18 we plot throughput per second for US and Asia registries. Here we can observe the initial high throughput pattern as for the European regions, however apart from that the request throughput is highly variable, and we cannot observe any other patterns. We can however note that GCR-Asia does observe highest throughput spikes outside of the initial one. Also, the IBM Registry Dallas has a delayed initial registry spike, indicating somewhat of a "cold start". The initial spike is followed by many smaller spikes later, indicating more of a spread out, queuing behavior.

The main takeaway from these graphs is that some registries implement requests queuing to handle large request load, which is definitely better than simply rejecting requests.

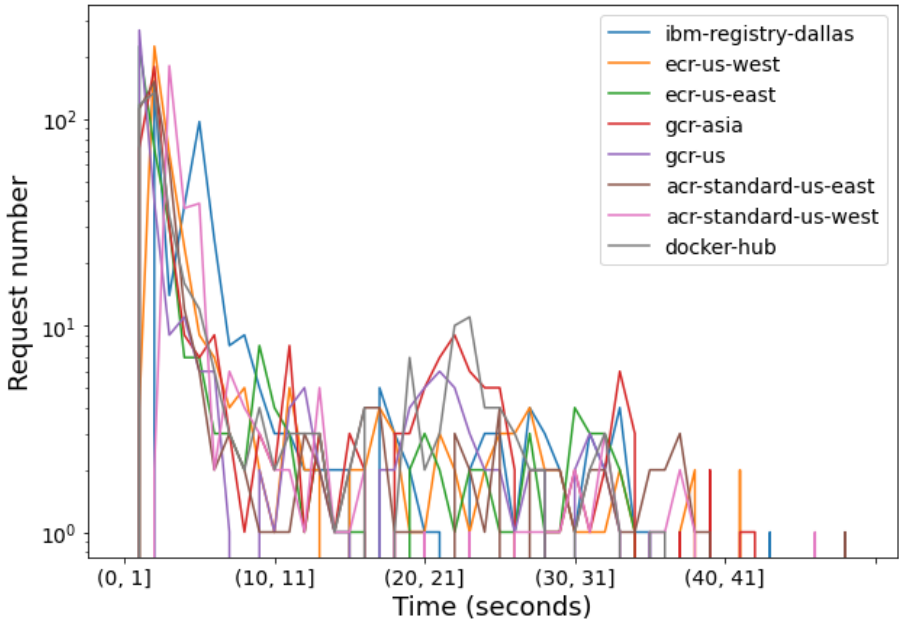

Figure 5.18: E1 Stress: Throughput per second – US/Asia.

However, there is a concern for these registries whether they can be used for deployments where there is a large number of nodes which all request new images at the same time.

In Figure 5.19 we can see the GET latency results without outliers exceeding 6 seconds. We see GCR EU, ECR EU-West, and ECR EU-Central show the lowest performance variance and best average performance. Surprisingly, performance of GCR US is comparable to performance of ACR EU-West. GCR Asia shows the largest performance variance out of all GCR registries, but overall the largest performance variance can be observed for IBM Registry Dallas. Docker Hub has a fairly low performance variance, however the general performance is consistently slower than the european registries, and also some US regions as well. We can still see a generally similar latency performance for all ACR premium tiers, which is explained by the latency not being a distinctive factor in the tier model.

Surprisingly, when comparing plots in Figure 5.20 and Figure 5.21 the general latency for layers and manifests does not differ a lot. This is an interesting finding, which does not correspond to the assumptions of file sizes being the most influential characteristic in the performance of registries. However, we can observe that the performance variance is much smaller for manifests, which corresponds to the fact that there is a smaller variance of manifest sizes compared to layer sizes. This is also caused by the outlier removal, as with layers there is a high number of outliers for every registry, which significantly contribute to average latency. This is confirmed by looking at the average values for layers and manifests

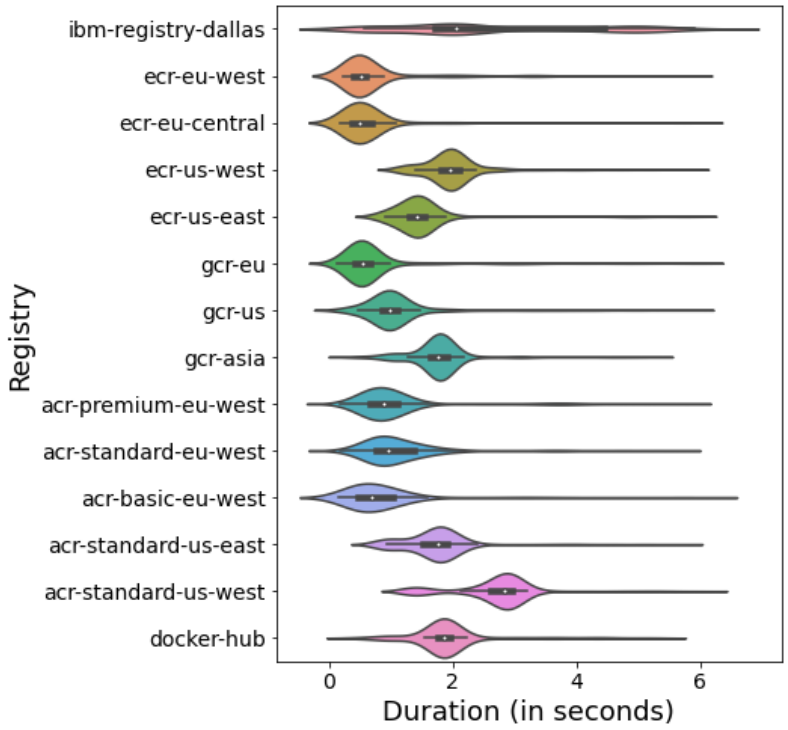

Figure 5.19: E1 Stress: Violin plot of GET request latency.

in Figure 5.22) and Figure 5.23. However, we can note that IBM Registry Dallas observes a high performance variance for all file sizes.

We can conclude that AWS ECR EU Central and EU West performed the best for manifest pulls, with sub 1 second average pull latency. Also interestingly ACR Basic EU-West performed the worst in the manifest latencies, because of the high amount of outliers which skewed the distribution and mean. This behavior cannot be seen in the filtered violin plot.

Finally, in Figure 5.24 we can see the put latencies. Here a larger performance variance for certain registries. There is a number of registries that top the performance in this category, namely ECR EU-West, ACR EU-West Premium and Standard. ECR EU-Central shows much larger performance variability than the aforementioned registries. GCR EU also generally performs well, however the performance variance is much greater than that of ECR EU West and ACR EU-West. Docker Hub again shows a large performance variance, but its general performance is similar to the US regional registries. Some US regional registries, namely ACR US Standard-West, ACR US Standard-East, and ECR US-West and ECR US-East, even though generally exhibiting larger latency for push, still show a

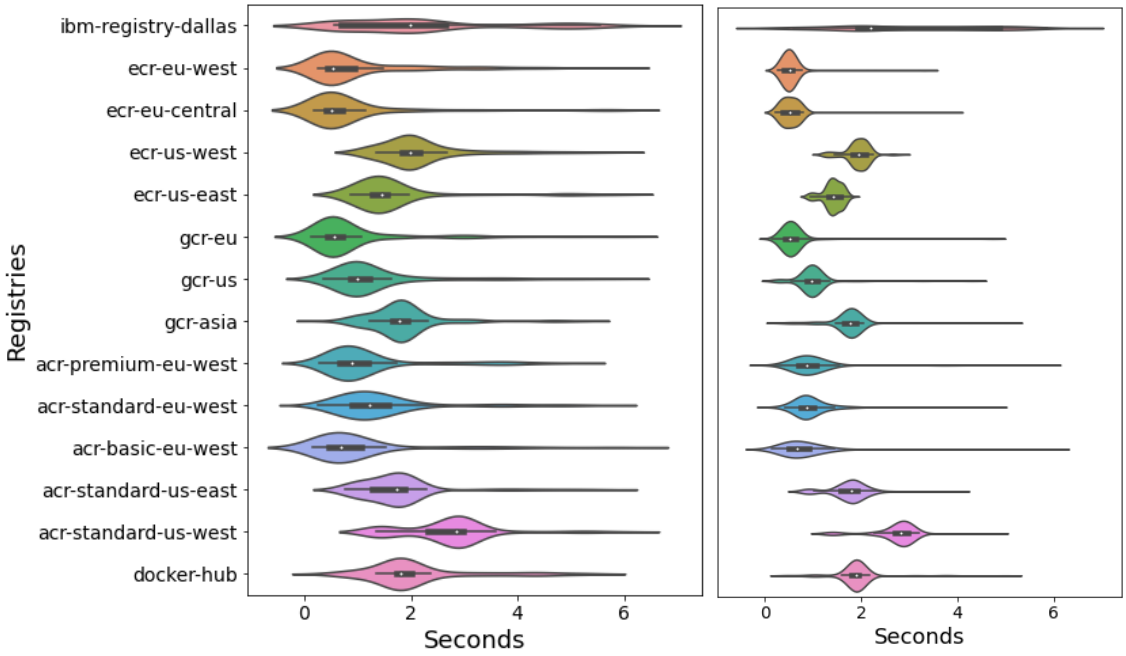

Figure 5.20: E1 Stress: Violin plot of GET layer latency.

Figure 5.21: E1 Stress: Violin plot of GET manifest latency.

very low performance variance.

5.2.2 Large Size Real-world Experiment Results

In Figure 5.25 we can see the violin plot for all GET request latency measurements. The violin and box plot for each registry shows a very low variability and low average latency per registry. This is due to the high number of request, which means that outliers do not influence the distribution and average performance as significantly as in the smaller real-world workload experiment (Figure 5.10). The lowest performance variance is observed for Google Cloud Registry EU, however the difference between all registries is relatively small. Azure EU-West observes a slightly higher performance variance than the other evaluated registries.

A different situation can be observed for manifest latency in 5.26. We see GCR generally has the lowest manifest latency, however with a slightly larger performance variance than ECR. Also GCR has high latency outliers that contribute to the performance variance. Out of all the tested registries, the highest performance variance can be seen in ACR EU-West, however the differences observed are in the tenth of a second range or even smaller.

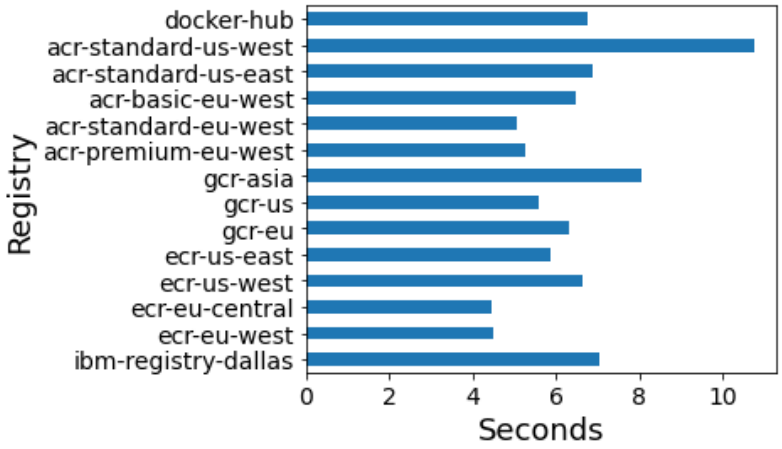

Figure 5.22: E1 Stress: Bar plot of average GET layer latency.

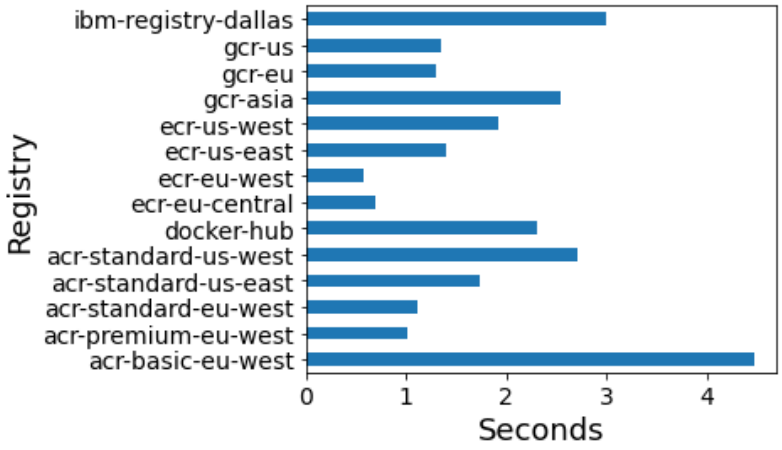

Figure 5.23: E1 Stress: Bar plot of average GET manifest latency.

Due to the size of this experiment, we are able to get more insight from push latency measurements, and analyze them in a more granular way, both for layers and manifests.

In the Figure 5.27 we can see differences in performance variance for layer push. ECR EU-West and Azure EU-West have the highest performance variance, indicated by the box plot length inside the violin plot. On the other hand ECR EU-Central and GCR EU have a very condensed box plot which indicates a very low performance variance. Finally in the Figure 5.28 we observe a difference in performance for the registries. For ECR EU-West, there is an outlier that took more than 14 seconds. This can be compared to the IBM Trace that we analyzed which also exhibits a high performance variance, however with a large amount of request sources. We also observe that all registries exhibit very low performance

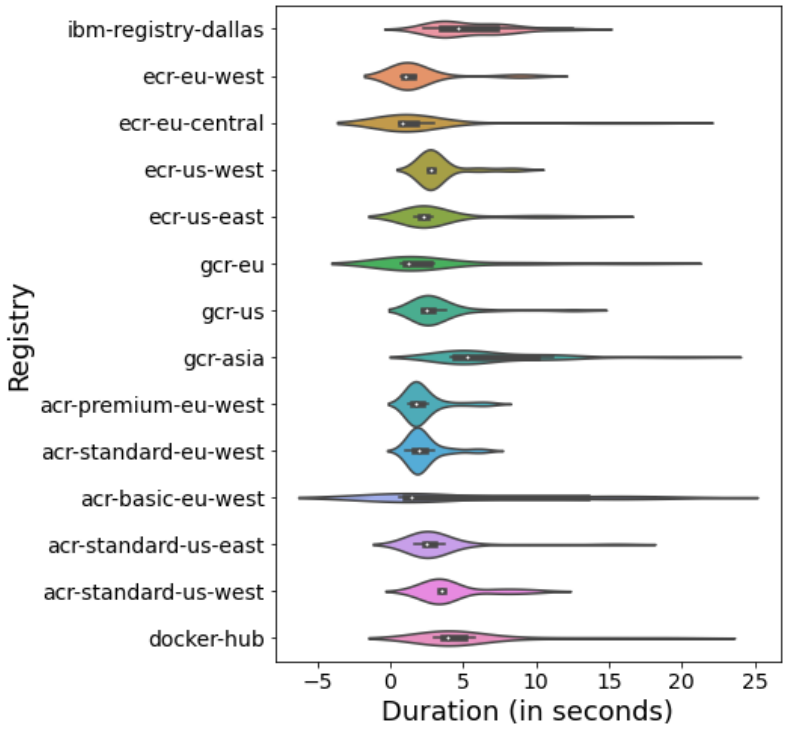

Figure 5.24: E1 Stress: Violin plot of push latency.

variance for manifest push as well.

For this experiment we can conclude that due to the size of the workload, the performance differences are very small between registries. We note that GCR EU seems to slightly outperform other registries based on performance variance and general latency. However, we cannot observe any noticeable bottlenecks or performance degradation that would be caused by a large workload.

5.2.3 Long-Running Experiment Results

For each of the registry categories, we analyze *push and pull latency*. Since we performed 10 pulls and pushes every 6 hours, we sum the latencies of 10 generated layers, presenting it as a 10 layer image. For pull and push latency over time analysis we plot the sum for every 6 hours for the 2 month period.

In the Figure 5.29 the pull latency over time for European regional registries. This graph highlights the importance of even a tenth of a second difference in the push/pull times when dealing with high number of layers. We see registries ECR `eu-west-1`, ECR `eu-central-1`, and GCR EU are the most stable and the fastest registries, consistently

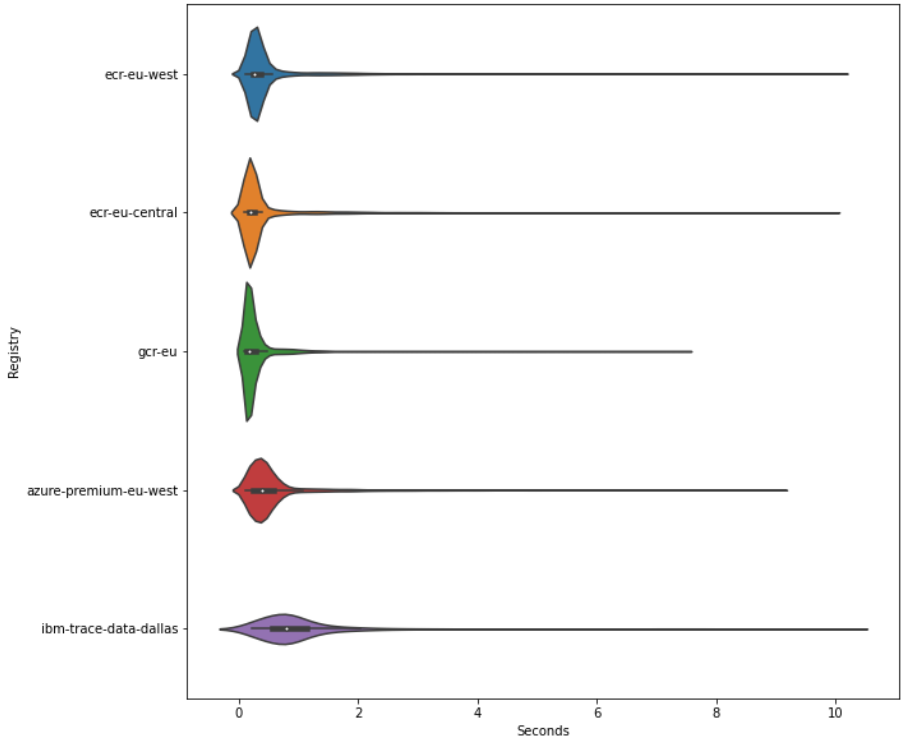

Figure 5.25: E2: Violin plot all GET requests latency.

performing around a 2 second mark for 10 layers.

The other three registries we recorded are ACR EU-West Basic, Standard, and Premium. These registries exhibited very low latencies at the start of the experiment, but around 9th of October they started exhibiting very high performance variance and generally their performance worsened. We reached out to Azure Container Registry team for an explanation regarding this performance degradation and are waiting for their answer. Another interesting note regarding these three registries is that in the period between 9th of October and 9th of November we can observe a difference in the pull latency between these three tiers, with Premium performing the best and Standard performing the worst. Since latency is not part of the tier guarantees, we do not know if this behavior is intended. After 9th of November the tier difference in performance does not have the same pattern.

In Figure 5.30 we can observe higher differences in performance between different US and Asian-Pacific regions. This can be attributed to the geographical distances, but we can also note differences between what should be geographically close regional registries from different cloud platforms.

GCR US performs the best, which follows the pattern from experiment E1, where perfor-

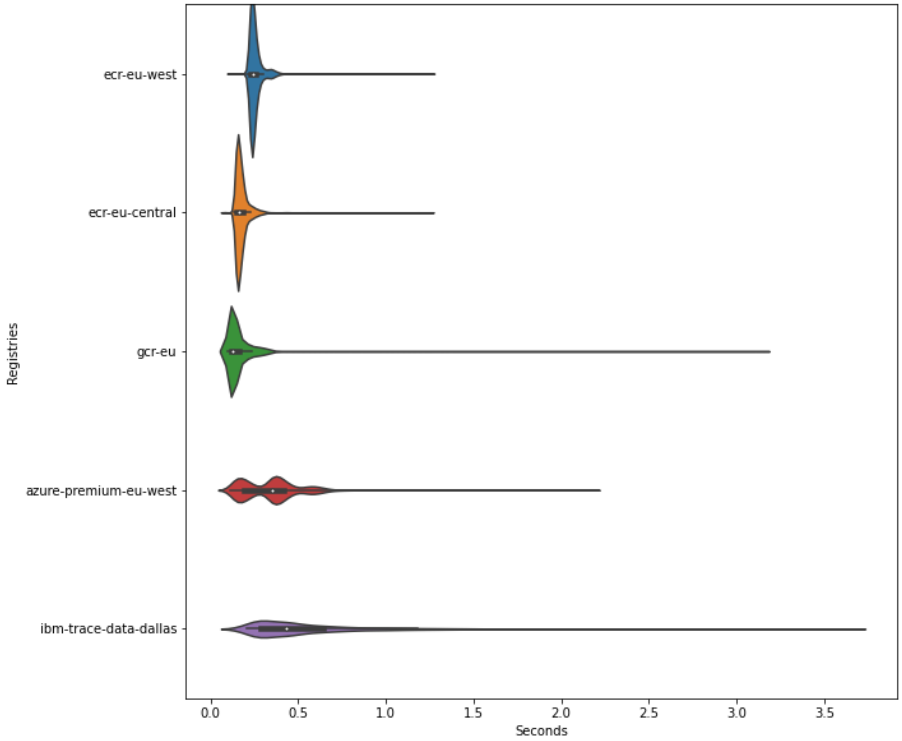

Figure 5.26: E2: Violin plot manifest GET requests latency.

mance of GCR US was comparable even to performance of some European regions. Next performance tier is ECR US East (North Virginia and Ohio). In the next tier we see the Asian and US-West regions: ECR `us-west-1` (North California), ECR `ap-southeast-1` (Singapore), and GCR Asia. Here we can observe that GCR Asia exhibits by far the largest performance variance, which we do not observe for other GCR registries.

Finally the worst performer is the ECR `ap-southeast-2` (Sidney), which is expected due to its geographical distance to the experiment source. We can note that even though the geographical distance is extremely large, the performance variance is fairly low.

In the last analysis of the pull layer measurements we look at the public registries in 5.31. Even though the three registries are virtually the same regarding the pricing model and offering, we can see there is a clear performance difference. They all observe a high performance variance, and the Gitlab registry exhibits a worsened performance since approximately 10th of November. On the other hand, Docker Hub observed a large performance variance at the start of the experiment, but despite few latency spikes the performance variance lowered.

In regards to push measurements, in the Figure 5.32 we can see the push measurements

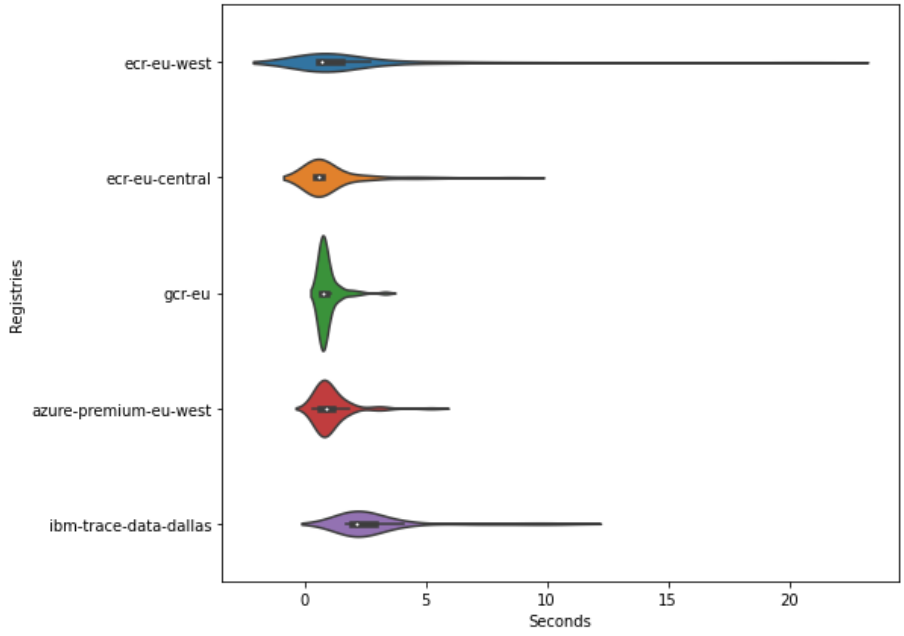

Figure 5.27: E2: Violin plot for layer push latency measurements.

for European registries. Again there is a pattern with ACR registries where around 9th of October there is a jump in latency and performance variability. Before that mark the ACR registries performed better than the rest of the registries. The other registries do not exhibit the same behavior, therefore we conclude that this pattern is due to a performance degradation in the ACR EU-West service. Interestingly, ACR Basic observes better performance than other ACR tiers in this category, with ACR Standard being the worst performer out of the three.

For the US and Asian-Pacific registries again we can see different performance tiers in the Figure 5.33 due to difference in the geographical distance to the experiment source in Amsterdam. We observe a lot of performance degradations in the services.

A striking example is the performance of ECR Sidney (`ap-southeast-2`). Its latency measurements were surprisingly low until around 20th of October, outperforming GCR US, ECR Singapore (`ap-southeast-1`), and GCR Asia. However after this mark we observe an extreme performance degradation, of around 2 seconds per layer. This in turn resulted in a 20 second degradation for a 10 layer image. We observe other performance degradations, albeit lower, for GCR Asia, ECR US-West-1, ECR US-East-1, and ECR US-East-2.

We see GCR US does not exhibit the same performance distinction from other registries for layer pushes as for the layer pulls. However, the performance variance is still lower

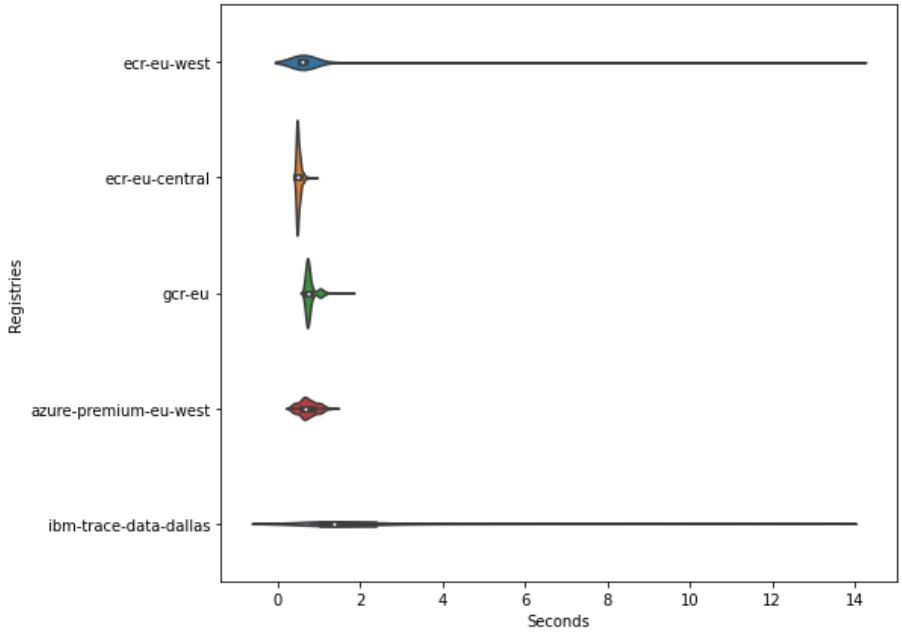

Figure 5.28: E2: Violin plot manifest push requests latency.

than what is observed for other registries (apart from a few latency spikes), and there is no performance degradation patterns observed in the graph.

Lastly, the public latency push measurements can be seen in the Figure 5.34. There are some extreme values, where pushing layers took more than a minute for certain registries. The performance variance observed for all three public registries is large. However, there is no noticeable performance degradation that we observed for public cloud regional registries.

5.2.4 Cost Analysis

In this section we present results from the general cost model for egress and storage size of up to 200TB.

Figure 5.35 shows the estimated storage costs for all platforms. As we can see the costs for Google Cloud Registry are the lowest due to it's model which consists of costs for storage and costs for operations, which differs from the other models. This also means that the higher the operations, the higher the cost. This pricing model for storage allows organizations to optimize number of operations, but also the costs could quickly explode if the organizations perform a high number of operations on registry (for example, polling the registry). However, the cost of storage operations is 0.05 $ per 10,000 operations which does allow for a high number of operations on the storage.

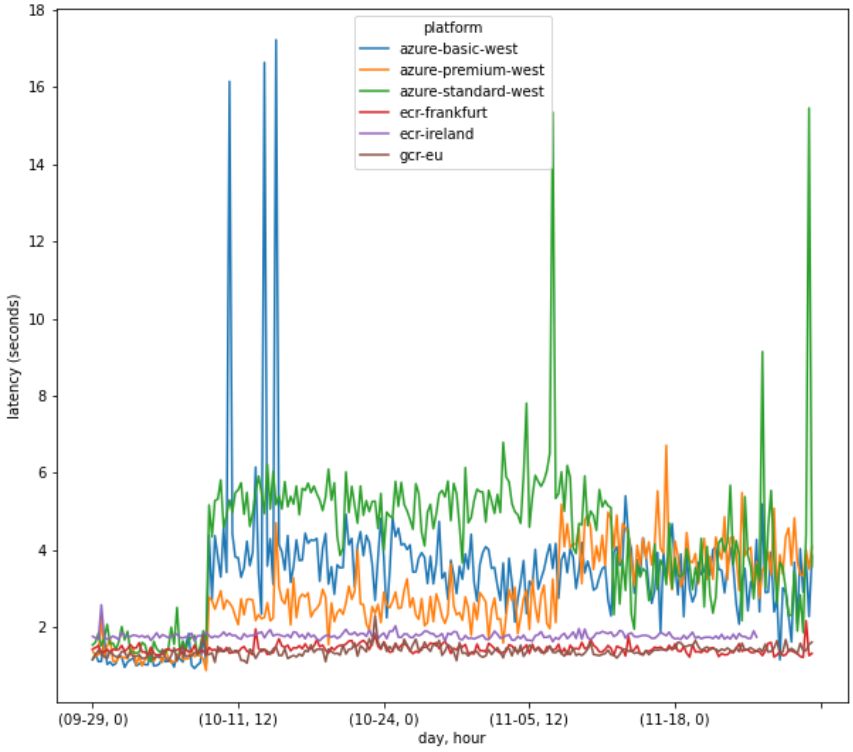

Figure 5.29: E3: Pull latency over time – Europe.

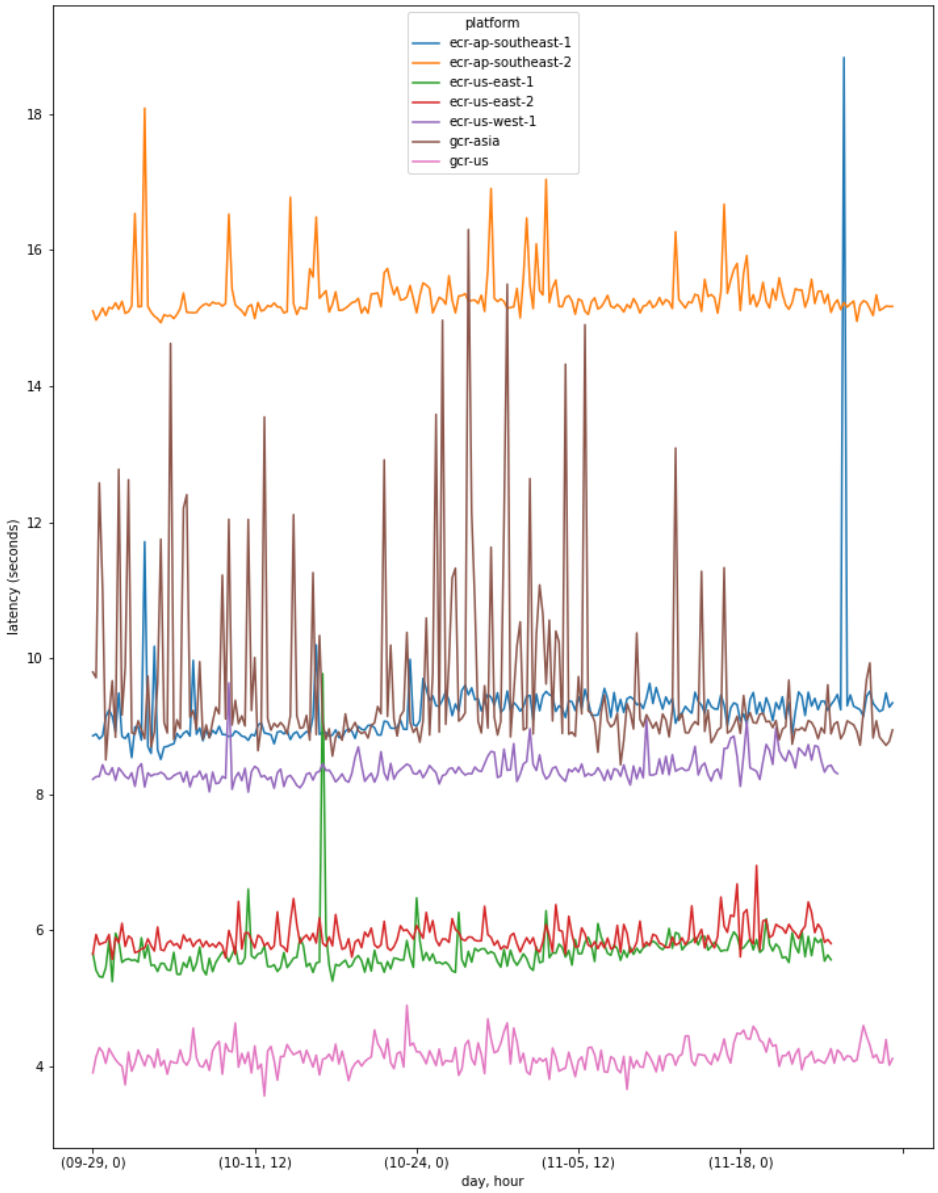

Figure 5.30: E3: Pull latency over time – US/Asia-Pacific.

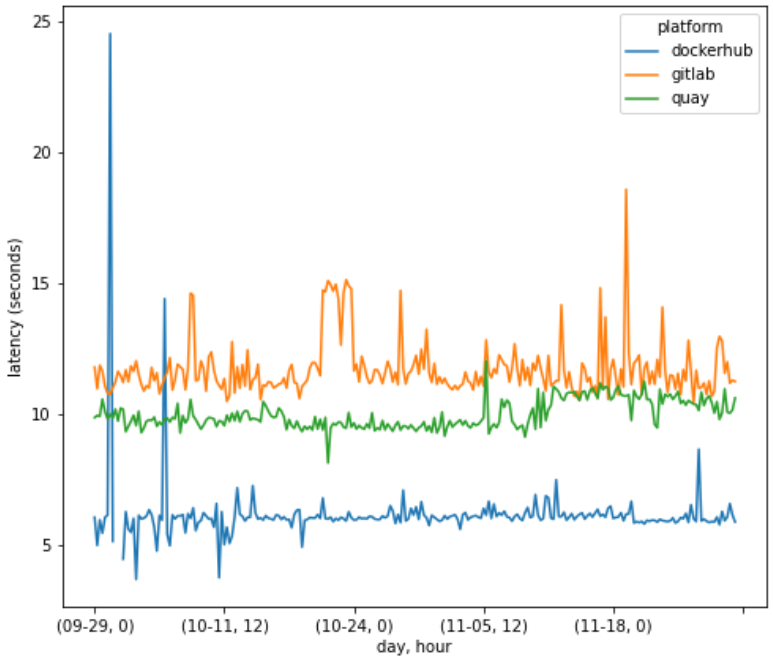

Figure 5.31: E3: Pull latency over time – Public.

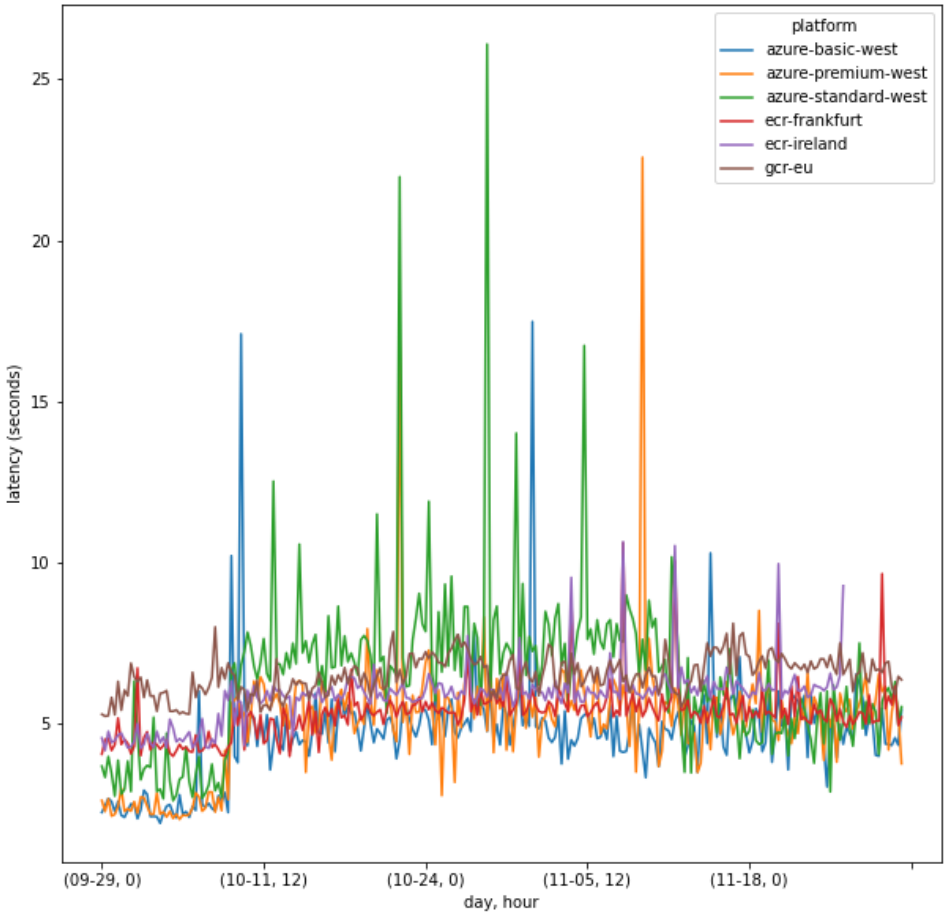

Figure 5.32: E3: Push latency over time – Europe.

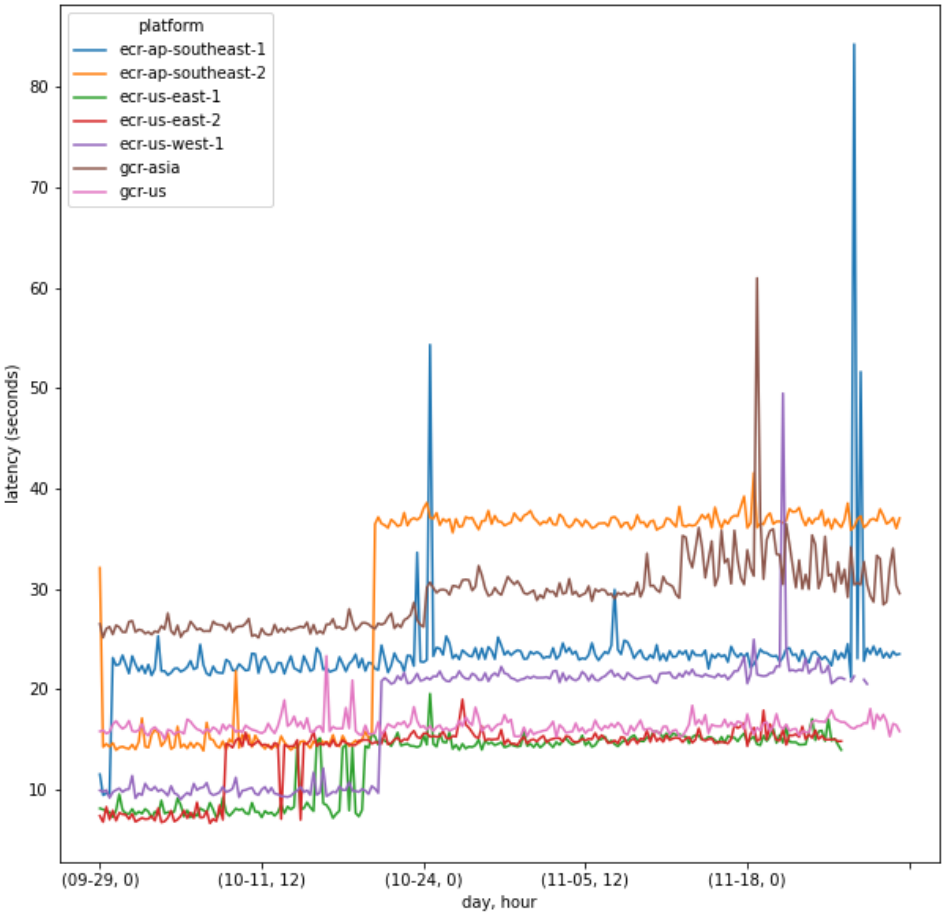

Figure 5.33: E3: Push latency over time – US/Asia-Pacific.

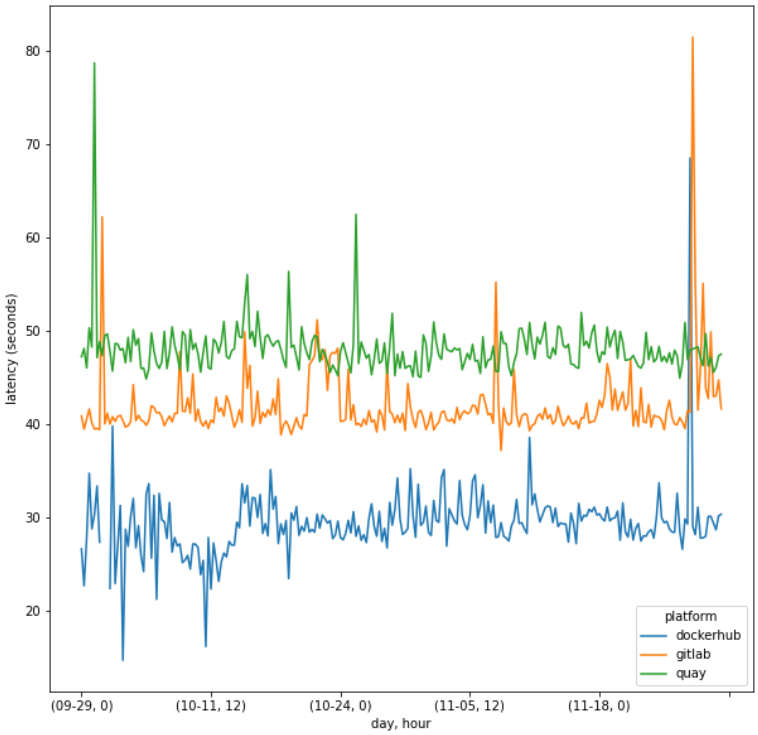

Figure 5.34: E3: Push latency over time – Public.

We can also see that there is no difference between different tiers of Azure Container Registry at scale. The scale at which it matters is small businesses or single users whose registry sizes are within storage included in the registry tier subscription. In Figure 5.36 the egress costs for registries. We can see a pricing model differentiation in this plot. It is important to note that for data egress inside the region there is no costs charged. However, this metric is relevant for hybrid deployments that incur these costs.

ACR egress costs are the lowest out of all public cloud registry costs. We can see a sharp price drop for ECR EU when egress exceeds 150TB. GCR and IBM CR do not reduce their price at this scale therefore they have a linear or close to linear price increase. In Figure 5.37 we can see the total cost for the equal amount of egress and storage. Due to the low storage cost, GCR has the lowest total cost out of all public cloud registries. IBM Container Registry is the second lowest, also due to its relatively low storage cost. All ACR tiers at scale have virtually equal cost, therefore at this scale organizations do not

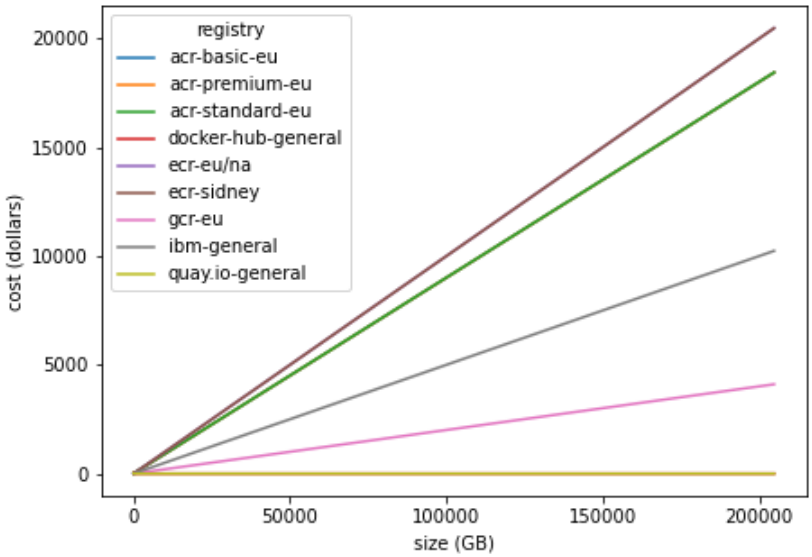

Figure 5.35: Storage cost estimation.

need to think too hard about which tier to choose, especially with all the added benefits that ACR Premium offers (ACR Tasks, Geo-replication, etc.). Most costs are incurred when using AWS ECR, no matter if the deployment region is one of the busier regions with lower costs per GB (Europe, North America) or the higher cost region (Sidney).

Also we conclude that even with the highest tier subscriptions, Docker Hub and Quay pricing model differentiates extremely from the public cloud models. This shows their focus on a very different customer base than the public cloud registries. However, the question is whether Quay and Docker Hub can handle these workload sizes (150TB+), especially considering the experiment results in this chapter. If the user or organization needs lower performance variability and low latency guarantees, they should not choose Quay or Docker Hub, especially because of their deployments that do not provide optimal performance for every region. However, for a single user or organization that wants to minimize their costs, and does not prioritize low latency, it should be considered as a valid option. Especially considering the added benefit of working with an open source technology.

5.3 Threats to Validity

First threat to validity we observe is the source of experiment. We have fixed the source of the experiment to scope the experiment configuration domain, however there are issues

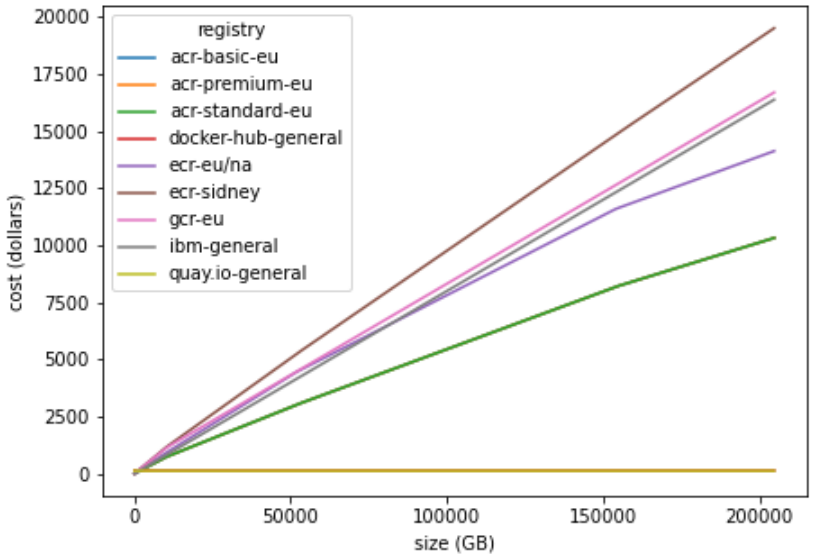

Figure 5.36: Egress cost estimation.

with this approach. First of all, the geographical distance plays a big role in latencies observed. The larger the distance, the higher is the chance that there is noise in the connection with the registries. We have tried to somewhat tackle this issue by comparing geographically close regional registries.

We have also gotten feedback from the Azure Container Registry team that we need to keep in mind that the cloud registries are optimized for delivering content within the cloud region. Performing external testing is possible, however it might not be of the most relevance to the industry.

Another potential threat to validity is the workload choices. Firstly, another point of feedback from the ACR team is that the registries are not optimized for the stress test.

We also think we could improve the trace sampling techniques used, and provide better reasoning behind selecting traces of specific size and characteristics.

In 6 we discuss possible experiment designs that could tackle the feedback provided by the ACR team.

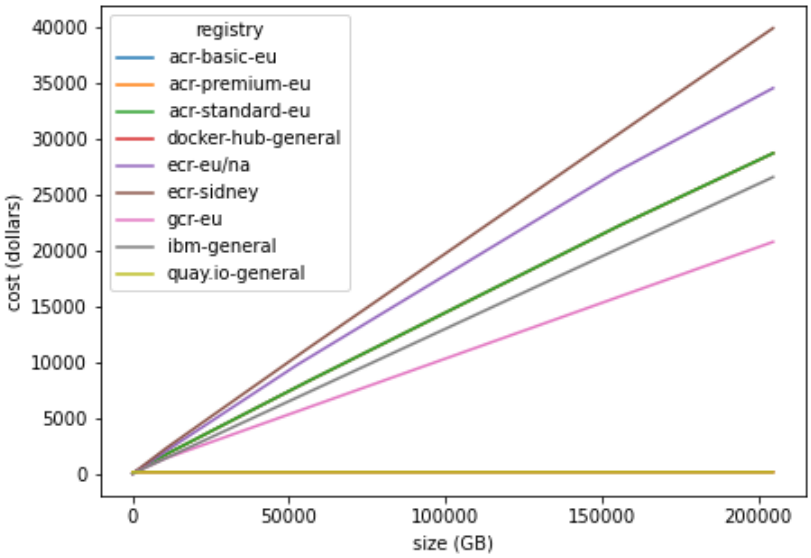

Figure 5.37: Total cost estimation.

6

Conclusion and Future Work

Container registry performance benchmark is important for gaining insight into performance of this crucial component in the deployment pipeline. Container registry set of functionalities is extending continuously and providing a basic evaluation enables further evaluations of new features and different use-cases that become more and more important (for example, serverless). In the public cloud environment, providing seamless experience where the user cannot distinguish whether the service is run on cloud or in a private datacenter is key, and improving deployment speed is the first step.

We identified shortcomings in the current registry performance evaluation, and presented a tool that bridges the gap between evaluation of performance optimizations in academic research and evaluation of production-grade real-world registries.

In this section we will summarize contributions and look at the future opportunities in the field of container registry research.

6.1 Conclusion

We posed three research questions in Chapter 1, which were aimed to guide us in solving the problem that we identified with performance evaluation of container registries. These questions are answered in Chapters 3, 4, and 5.

RQ1. What are the key aspects of a container registry benchmark relevant to users?

We presented a systematic literature and systems survey in Chapter 3 aimed at identifying current advances and concerns in the field of container registries. We surveyed scientific literature and state-of-the-art production registries to identify metrics

that are used to describe registry performance, workload types and traces which will encompass typical workloads that production registries need to handle, and set of registries that are widely recognized by the community and used by organizations.

RQ2. How to design a registry benchmark that supports key registry benchmark aspects?

In Chapter 4, based on the metrics, workloads, and registries identified in the literature and system survey conducted to answer RQ1, we created requirements for the benchmark instrument. We presented the benchmark that satisfies the user requirements and the general benchmark requirements. This benchmark integrates another state-of-the-art registry evaluation instruments and allows fine tuning of all component parameters from a central point. We implemented the benchmark instrument design and open-sourced it for community use. With this benchmark we provide a method to evaluate container registry performance to support stakeholder use-cases identified in Chapter 4, and likely more use-cases outside of the identified scope.

RQ3. What are the cost and performance differences between production-grade, hosted registry platforms?

We designed experiments by varying benchmark instrument parameters (workloads, registry set). We deployed the benchmark instrument in the DAS cluster and on a machine in the public cloud. We executed the experiments and report results in Chapter 5. We found patterns of performance degradation over time in cloud registries, and we also expierenced rate throttling which gave insight into size of datacenters where the registries are located. We also gained insight into performance difference between public registries, and private cloud registries, and specifically difference in performance between cloud registries located in the similar regions.

We further explain how we answered the research questions posed in Chapter 1. We applied a structured approach to this topic, identifying metrics, types of workloads and registries in industry and academia. Then we translated this into the requirements for an instrument that performs the benchmark. Finally we execute an extensive registry evaluation using the implementation of the instrument.

6.2 Future Work

With our work we scraped the surface of performance evaluation of container registry evaluation. We provide an instrument that can be used in different environments to test different optimizations that are implemented in the state-of-the-art-registries.

However, we also know that there are other factors that contribute to the container startup latency. There is an opportunity to extend or change the component of interest in SUT, focusing on different parts of registry interaction to provide a complete registry interaction evaluation. We present the following future work ideas:

1. **Region Isolated Experiment:** We want to isolate the noisy network channel and perform a large scale evaluation inside one cloud region. We aim to evaluate on-premise registries that can be deployed inside the cloud alongside the cloud provider registry. This would provide insight into whether the cloud provider registry is the best solution inside the region and if possibly other registry solutions actually perform better. This experiment would not be cost intensive, as the egress costs would not be incurred due to all data transfer happening inside the cloud region.

2. **Geo-Replication Experimental Evaluation:** Many cloud providers offer managed geo-replication feature. We want to evaluate the performance of this feature compared to the performance of pushing an image directly to multiple regions. The extra hops taken inside the cloud datacenter before pushing the image to another region could introduce latency that is unwanted in latency sensitive multi-region deployments.

3. **Compression Technique Evaluation:** We want to investigate compression techniques for delivering images as fast as possible to the client. In current registry design the network is the bottleneck, therefore image layers are compressed to reduce the network data transfer. In the current design the major contributor to container startup time is the time required for layer decompression. Azure Container Registry is inovating with Project Teleport, where image layers are delivered uncompressed. Project Teleport is aimed at cloud deployments where network is not the bottleneck. We want to evaluate different compression rates and possibly translate this approach to outside of cloud, possibly via adaptive compression rates.

4. **Benchmark Instrument Extension:** We want to extend the benchmark instrument to support finer-grained experiment configurations. We want to extend the instrument with a sophisticated workload generation tool with the IBM trace format.

6. CONCLUSION AND FUTURE WORK

We want this instrument to become a comprehensive tool for registry evaluation. Other plans are to extend the tool with the registry provisioning and cleanup to minimize costs and streamline the experiment process.

References

[1] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. **Slacker: Fast Distribution with Lazy Docker Containers**. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, February 2016. USENIX Association. 1, 3, 4, 27

[2] Haining Zhang. **What is a Container Registry?**, Jun 2017. 1

[3] Cody Gibb, Evelyn Liu, and Yiran Wang. **Introducing Kraken, an Open Source Peer-to-Peer Docker Registry**, Mar 2019. 2, 4

[4] Erwin Van Eyk, Alexandru Iosup, Johannes Grohmann, Simon Eismann, André Bauer, Laurens Versluis, Lucian Toader, Norbert Schmitt, Nikolas Herbst, and Cristina L. Abad. **The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms**. *IEEE Internet Computing*, **23**(6):7–18, November 2019. 2

[5] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. **Benchmarking in the Cloud: What It Should, Can, and Cannot Be**. In Raghunath Nambiar and Meikel Poess, editors, *Selected Topics in Performance Evaluation and Benchmarking*, pages 173–188, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 2, 5, 31, 32, 42

[6] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. **How is the Weather Tomorrow? Towards a Benchmark for the Cloud**. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, New York, NY, USA, 2009. Association for Computing Machinery. 2

[7] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. **Benchmarking Personal Cloud Storage**. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, page 205–212, New York, NY, USA, 2013. Association for Computing Machinery. 2

[8] Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, and Zhiteng Huang. **COSBench: A Benchmark Tool for Cloud Object Storage Services**. pages 998–999, 06 2012. 2

[9] D. Agarwal and S. K. Prasad. **AzureBench: Benchmarking the Storage Services of the Azure Cloud Platform**. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 1048–1057, 2012. 2

[10] E. Bocchi, M. Mellia, and S. Sarni. **Cloud storage service benchmarking: Methodologies and experimentations**. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 395–400, 2014. 2

[11] Rachel Stephens. **IaaS Pricing Patterns and Trends 2020**, Jul 2020. 2

[12] Ben Saunders. **Who's Using Amazon Web Services?**, Jan 2020. 3

[13] Jorge Ortiz. **AWS Case Study: Stripe**, 2015. 3

[14] Ari Levy. **Slack to spend at least $250 million on Amazon Web Services over five years - less than Pinterest or Lyft**, Apr 2019. 3

[15] S. Sultan, I. Ahmad, and T. Dimitriou. **Container Security: Issues, Challenges, and the Road Ahead**. *IEEE Access*, **7**:52976–52996, 2019. 3

[16] John Kriesa. **Introducing the Docker Index: Insight from the World's Most Popular Container Registry**, Feb 2020. 3

[17] Datadog. **11 Facts About Real-World Container Use**, Nov 2020. 3

[18] Michael Littley, Ali Anwar, Hannan Fayyaz, Zeshan Fayyaz, Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Mohamed Mohamed, Heiko Ludwig, Yue Cheng, and Ali Butt. **Bolt: Towards a Scalable Docker Registry via Hyperconvergence**. pages 358–366, 07 2019. 4, 28, 45

[19] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. **Improving Docker Registry Design Based on Production Workload Analysis**. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278, Oakland, CA, February 2018. USENIX Association. 4, 26, 28, 29, 31, 44, 45, 54, 56, 57

[20] Joab Jackson. **Dragonfly Brings Peer-to-Peer Image Sharing to Kubernetes**, Apr 2020. 4

[21] Ali Haider Zaveri. **Location-Aware Distribution: Configuring servers at scale**, Mar 2020. 4

[22] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R. Butt. **DupHunter: Flexible High-Performance Deduplication for Docker Registries**. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 769–783. USENIX Association, July 2020. 4, 28, 45

[23] C. Pahl. **Containerization and the PaaS Cloud**. *IEEE Cloud Computing*, **2**(3):24–31, 2015. 7

[24] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder. **Performance Evaluation of Microservices Architectures Using Containers**. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34, 2015. 7

[25] David Bernstein. **Containers and cloud: From lxc to docker to kubernetes**. *IEEE Cloud Computing*, **1**(3):81–84, 2014. 8

[26] Daniel Price and Andrew Tucker. **Solaris Zones: Operating System Support for Consolidating Commercial Workloads.** In *LISA*, **4**, pages 241–254, 2004. 8

[27] Poul-Henning Kamp and Robert NM Watson. **Jails: Confining the omnipotent root**. In *Proceedings of the 2nd International SANE Conference*, **43**, page 116, 2000. 8

[28] Eric Carter. **Sysdig 2019 Container Usage Report: Kubernetes, Security**, Nov 2020. 8, 11

[29] Ian Lewis. **Container Runtimes Part 1: An Introduction to Container Runtimes**, Dec 2017. 10

[30] Alena Varkockova. **What's the difference between runc, containerd, docker?**, Jun 2018. 10

[31] Barbara Kitchenham and Stuart Charters. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007. 18

[32] Joachim Schöpfel and Dominic J Farace. **Grey literature**. In *Encyclopedia of library and information sciences*, pages 2029–2039, 2010. 19

[33] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. **Guidelines for including grey literature and conducting multivocal literature reviews in software engineering**. *Information and Software Technology*, **106**:101–121, 2019. 19

[34] Laurens Versluis and Alexandru Iosup. **A Survey and Annotated Bibliography of Workflow Scheduling in Computing Infrastructures: Community, Keyword, and Article Reviews – Extended Technical Report**, 2020. 21

[35] A. B. Ayed, J. Subercaze, F. Laforest, T. Chaari, W. Louati, and A. H. Kacem. **Docker2RDF: Lifting the Docker Registry Hub into RDF**. In *2017 IEEE World Congress on Services (SERVICES)*, pages 36–39, 2017. 22

[36] A. Brogi, D. Neri, and J. Soldani. **DockerFinder: Multi-attribute Search of Docker Images**. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 273–278, 2017. 22

[37] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. **A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term**. *Computer*, **49**(5):54–63, 2016. 59