How does Docker ACTUALLY work? The Hard Way: A Comprehensive Technical Deep Diving

Furkan Türkal · Follow

35 min read · Jun 4, 2024

Docker. Containers. The revolution of the orchestration. An industry-leading Platform as a Service product. Build, share and run. Any app, anywhere… Just like that. Have you ever wondered *“How Docker works”*? It’s not magic. It’s talent and sweat.

Container world is a big iceberg with lots of unknownness and darkness

In this article, we will find answers the following questions:

- How has containerization changed the world?
- How the containers took over the cloud ecosystem?
- What does “container” mean exactly?
- What is “Docker” actually?

The aim of this article is to create a comprehensive **zero to hero** pathway from higher-level to lower-level for the developers who want to learn and understand more about “Docker”. The target reader audience is, who wants to:

- Learn what does Docker use under the hood
- Understand containerization in-depth
- Know the relationships between components
- Master in the overall mental model
- See the big picture

Four years ago, I knew almost nothing about containers. Nowadays, I am engulfed in the container world. I have decided to compile my 4-years knowledge in one place to help everyone interested in or wanting to learn about Docker stay aligned. I'm Furkan, have been tackling with *containers* for 4+ years at Trendyol.

How to read this article

This article may be a bit long. Here are some guidances on how to read it:

- We start with a high level overview, briefly describing the concepts
- We then explain the full concept starting, introducing the design detail with simple examples
- After the high level designs are completely described, we look up the some components, give some issues and PRs related to implementation, cross references to articles and authors
- We then deep dive into internal underrated components, specifications, behind the scenes, internal details
- We will wrap up the article by summarizing the key points covered

Notes

- Buzzwords alert: You will find lots of cross-references to external sources and important keywords, each of which is worth looking at in more depth.
- No-AI: This article does not contain any GPT or AI generated content! Grab a coffee and enjoy reading!
- Drawings: All drawings are hand-made by me on Excalidraw; and all rights

reserved.

Abstract

Container-related technologies has been used in various contexts in recent years in order to deliver software quickly, with a few commands. This article conceptualizes and examines the containerization technologies in depth. This article will explain a set of specific terms' step by step, overall architecture, motivation and the mental model. We present a qualitative examination of the Docker architecture and the components. We will not cover every single little detail in this article, instead, enlighten to what kind of things happen behind the scenes. You will find lots of mentions, external redirections, cross references in this article.

Motivation

Although it's hard to objectively quantify and is open to subjective interpretation anyway, what's more insightful than jumping the documentation like a monkey is understanding an entire code-base. If we are on the way to building the future as engineers, I think we need to **know how the technology we use are created**. Created by *whom*. *Where*. *How*. That's why it's crucial to dive deeply into something, anything; anywhere.

I have never deep-dived to container world before. The main point here is to be able to do something educational, figuring out the overall architecture and wrapping up the all different contexts in one place! Together, we will learn how to do this. Moreover, we will learn valuable information by writing these letters. Furthermore, it can be a great opportunity to make contributions, to meet new people from community, the brains behind the scenes, the architects, the decision makers!

Technology changes. All the time. Our knowledge is our value. We *have to* keep up with the latest technology changes and updates. So, let's *understand* instead of *memorizing*, let's *see* instead of *hearing*, let's *learn* instead of *ignoring*, let's *demonstrate* instead of *assuming*. These motivations are pretty much enough to continue to further from here!

1. Introduction

Containers have become the lingua franca of today's infrastructure for sure, providing a universal language that transcends the complexities of diverse environments and empowers seamless deployment and scalability. With their lightweight and modular design, containers encapsulate applications and their dependencies, ensuring consistency and portability across different platforms, from local development environments to cloud-based deployments. The adoption of containers continues to grow exponentially, redefining the way applications are developed, deployed, and managed in today's dynamic and fast-paced technological landscape.

One of the solutions that has revolutionized the way we build, ship, and run applications is Docker. With its ability to package software into self-contained units called containers, Docker has transformed the world of software development and deployment.

Let's explore the driving forces behind the rise of Docker and why it has gained immense popularity among developers, operations teams, and organizations of all sizes.

2. What is Docker?

Docker is an open platform for developing, shipping, and running of applications within containers. Docker simplifies the process of creating, distributing, and

running applications by utilizing containerization technology. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. It abstracts the underlying infrastructure and provides a standardized way to package and deploy software. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

2.1. Rise of the Containers

Almost every software application were built in as a big *monolithics* last decades years. The aim was that handle all process and every function in one single service. The whole system was managed and used in a monolithic way. That is the answer why monolithics builds were updated infraquently. All system has to be package up at the same time then, deliver to the ops team. The complete system was taken over by the ops team that developed health services and monitoring. This legacy approach is still used for small projects and teams.

Struggles in deploying and this old structure that does not respond to new features make required to developers should find new service design. In that point microservices are come out. Microservice architecture is actually a monolithic structure that broken down into smaller parts. All services are working independently. Microservices running in separate from each other, they may updated, leveraging and scaled individually. This gives opportunity to both develop and ops team to keep software recent, deploy product rapidly. The number of companies using this structure continues to increase day by day. Most common users are Netflix, Google, Amazon, etc.

Microservices have positive sides, but otherwise with bigger number of small services and business has mean more workloads, difficult to control and merge them as like a single software. It is much harder to make them running smoothly and keep hardware costs down. That point, the duties of ops team is maximizing the automation to prevent human mistakes or failure and develop a scalable software. [17]

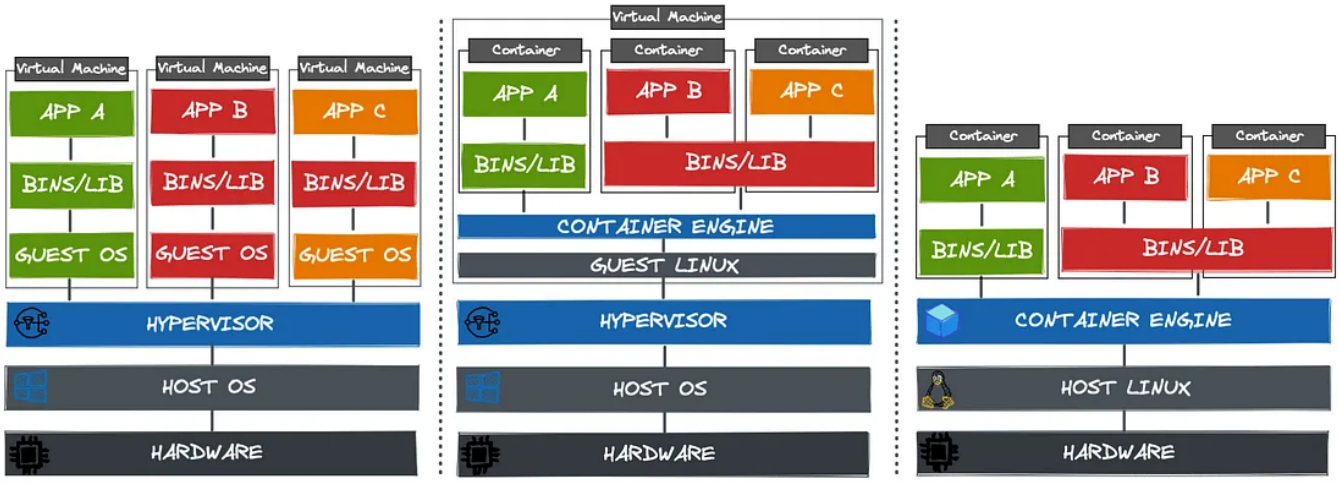
2.2. Containers vs VMs

Virtualization is the process of running a virtual instance of a computer system in a layer abstracted from the hardware system. It enables to running multiple operating systems on a computer at the same time by Virtual Machines.

Virtual Machines (VMs) are an abstraction of physical hardware that turns one server into many. VM allows multiple VMs to run on a single machine. Each VM includes a full same of an operating system, the application, and necessary binaries and libraries and all dependencies. VMs are not the best way to keep cost down and avoid waste hardware resources since each VM needs to be managed and configured. By this reason, migration from virtualization to container technologies is increasing day by day.

Linux container technologies allow developers to run multiple microservices on the same machine while not prepare a different environment to each service. Also, make them isolated them from each other by using containers.

Read the Learning Containers From The Bottom Up article for containers learning path.



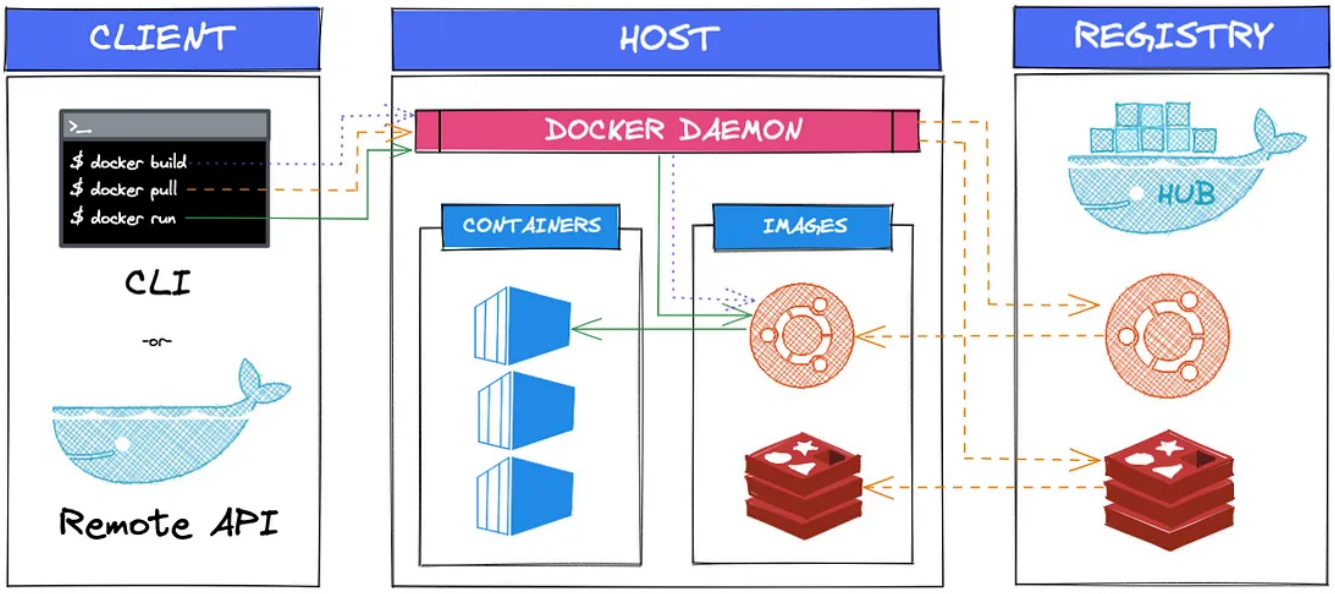
Containers vs Virtual Machines [0] (by @furkan.turkal)

3. Docker Architecture

Docker architecture is a client-server model that enables the creation, distribution, and deployment of containerized applications.

3.1. High-level Architecture

Docker's higher level of architecture revolves around a client-server model, where the client interacts with the Docker daemon (server) to manage containers and related resources. At its core, Docker consists of three key components: the client, the daemon, and images. Client and daemon communicate using a REST API, over UNIX sockets or a network interface.



High-level architecture of Docker (by @furkan.turkal)

- **Docker Client**: is a command-line tool, API, or graphical interface that users interact with to issue commands and manage Docker resources. The client sends requests to the Docker daemon, which orchestrates the execution of those commands.

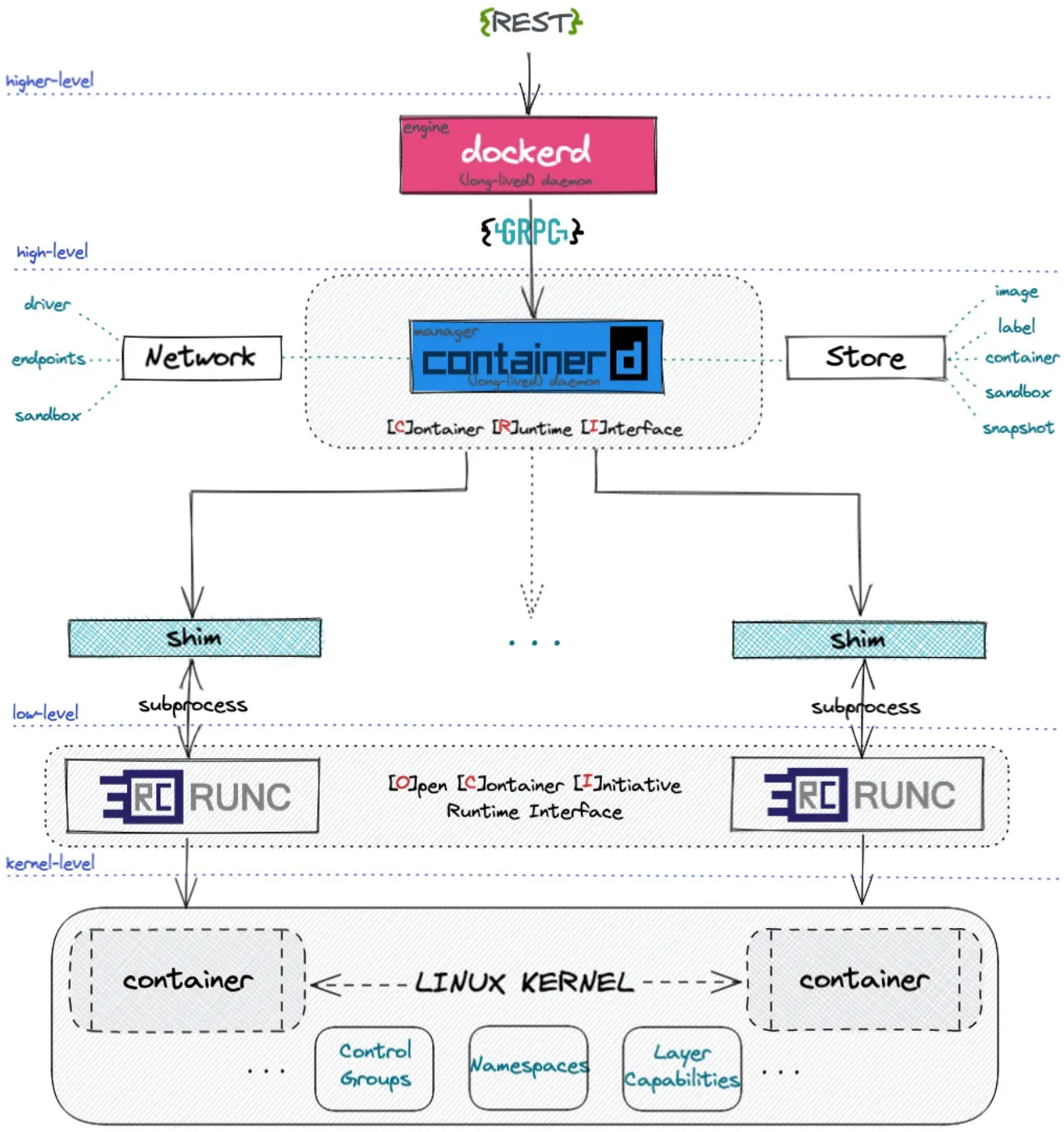
- **Docker Daemon:** also known as Docker Engine, is a background service and long-running process that runs on the host machine and actually does the work of running and managing both containers and container images. The Docker daemon is responsible for managing the lifecycle of containers and orchestrating their operations. It listens for requests from the Docker client, manages containers, and coordinates various Docker operations. The daemon

interacts with the host operating system's kernel and leverages kernel features and modules for containerization, networking, and storage.

- **Docker Desktop**: is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. With Docker Extensions, you can use third-party tools within Docker Desktop to extend its functionality.
- **Docker Registry**: is a *registry* that stores container images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default.

3.2. Low-level Architecture

Since we will deep dive into each technology in detail, but first let's look at the architecture from 30,000-foot view:



Low-level architecture of Docker (by @furkan.turkal)

1. Docker Desktop employs a *virtual machine (VM)* to provide the necessary Linux environment.

2. Within the VM, the Docker client interacts with the Docker daemon *(dockerd)* through a RESTful API.

3. It uses containerd as its container runtime under the hood. Containerd is an industry-standard runtime that manages the lifecycle of a container on a physical or virtual machine. It is a daemon process that creates, starts, stops, and destroys containers.

4. Containerd Plugins can be added to containerd to extend its functionality.

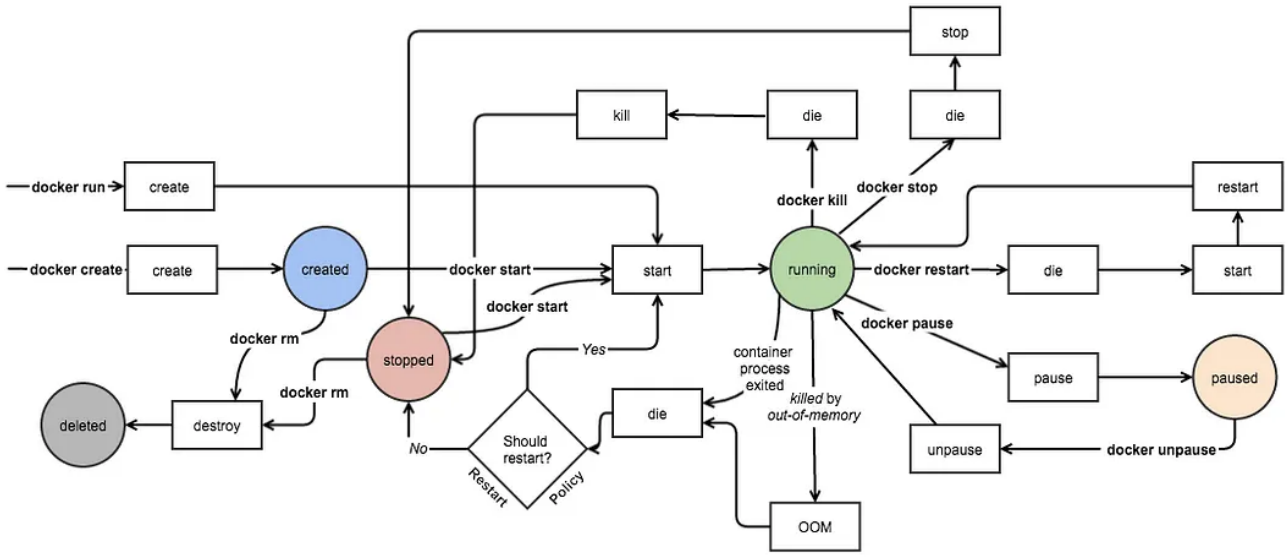
5. When a container is launched, the shim (runtime v2) API, which is part of containerd, acts as an intermediary between containerd and the *OCI runtime*.

6. The OCI runtime is responsible for setting up the container's namespaces, control groups (cgroups), capabilities, and other settings required for containerization. It leverages the Linux kernel's capabilities to isolate and control resources, enforce security, and manage the container's behavior. Cgroups, short for control groups, are a Linux kernel feature that allows fine-grained resource allocation and control for processes. Capabilities within the Linux kernel provide privileges to containers, defining their permissions and access to various system resources.

Docker CLI

Docker CLI tool is a command line application used to interact with the dockerd daemon. It includes several useful features. It handles standard UNIX-style arguments, and in many cases, it offers both short and long forms.

You can find all the commands in the repository if you are deeply interested.



A state machine provides a good summary of how the various states of the container are converted [1]

Docker Registry

Docker Registry is a service that allows users to store and distribute the container images. It serves as a centralized or decentralized locations where users can push their images, making them publicly available, accessible to other team members or systems for deployment.

> *Insterested in How to create your own private Docker registry and secure it?*

There are some decentralized registires such as OpenRegistry. A few months ago @ktokunaga.mail wrote a blog post about "P2P Container Image Distribution on IPFS With Containerd". IPFS support introduced in nerdctl v0.14.

> *IPFS is a peer-to-peer and content-addressable hypermedia data sharing protocol designed to preserve and grow humanity's knowledge by making the web upgradeable, resilient, and more open.*

There are bunch of tools to interact with the Registry API:

- **crane:** is a tool for interacting with remote images and registries

- **skopeo:** is a command line utility that performs various operations on container images and image repositories
- **regctl:** is a client interface for the registry API

BuildKit

BuildKit is a concurrent, cache-efficient, and Dockerfile-agnostic builder toolkit.

By integrating BuildKit, users should see an improvement on performance, storage management, feature functionality, and security.

To enable BuildKit builds you can either pass `DOCKER_BUILDKIT=1` environment flag when invoking `$ docker build` on the client-side or set `{ “features”: { “buildkit”: true } }` in your `/etc/docker/daemon.json` file and restart the daemon afterward. Currently only supported for building Linux containers.

BuildKit has two primary components: `buildctl` and `buildkitd` . buildctl communicates with buildkitd through a gRPC server. During the initialization it sends a message to the init daemon (systemd) through sdnotify. We are sending a `sddaemon.SdNotify(false, sddaemon.SdNotifyReady)` to tell the service manager that service startup is finished; and SdNotifyStopping eventually.

BuildKit is designed to work well for building for multiple platforms and not only for the architecture and operating system that the user invoking the build happens to run.

BuildKit supports the creation of SLSA Provenance for builds that it runs. The provenance format generated by BuildKit is defined by the SLSA Provenance format.

> ***SLSA (Supply-chain Levels for Software Artifacts)*** *is a security framework, a checklist of standards and controls to prevent tampering, improve integrity, and secure packages and infrastructure.*

Buildx

Docker Buildx is a CLI plugin that extends the docker command with the full support of the features provided by Moby BuildKit builder toolkit. Docker Buildx always enables BuildKit. It provides the same user experience as docker build with many new features like creating scoped builder instances and building against multiple nodes concurrently.

Let's build an example multi-platform multi-stage image:

```
docker buildx build — platform <platform1>,<platform2>,… — tag <image_name> — f
```

Alternative Ways to Build Container Images

You do not need *just* "Docker" to build container images. There are many ways for building OCI compliant container images:

- google/ko

> *ko builds images by effectively executing* `go build` *on your local machine, and as such doesn't require* `docker` *to be installed. It's ideal for use cases where your image contains a single Go application without any/many dependencies on the OS base image (e.g., no cgo, no OS package dependencies).*

Want to use ko on your GitLab CI/CD pipeline? It's just that easy:

```
$ KO_DEFAULTBASEIMAGE: gcr.io/distroless/static:nonroot
$ KO_DOCKER_REPO: ${CI_REGISTRY}/${CI_PROJECT_NAMESPACE}
```

```
$ ko login $CI_REGISTRY -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
$ ko apply -B --bare -f your_deployment.yaml
```

Want to create vulnerebility-free container images? It's just that easy with apko!

Here is a list of container image builders:

- GoogleContainerTools/kaniko
- GoogleContainerTools/jib
- containers/podman (daemonless!)
- containers/buildah (daemonless!)
- moby/buildkit
- docker/buildx
- genuinetools/img
- uber/makisu (deprecated)
- pivotal/kpack
- openshift/source-to-image
- buildpacks
- nix/ocitools
- rancher/kim
- shipwright
- earthly
- bazelbuild (rules_docker)

BONUS: If you interested in building OCI container images without using Docker by

building the layers and image manifests programmatically using the go-containerregistry module, you should take a look to @ahmetb 's blog post.

> *go-containerregistry is a golang library for working with container registries seamlessly. It is used by thousands of projects.*

Container Runtime Interface (CRI)

Heard of Kubernetes (K8s)? If not, it's fine. We did not mention on this article until now. Skip this session if you haven't heard of Kubernetes before.

> *Kubernetes is responsible for orchestration, runtime and knows how to run and check the status of* containers. *Docker launched Swarm, its own Kubernetes alternative, offering orchestration as a built-in Docker "mode".*

Kubernetes have a component called "kubelet" — an agent that runs on every node (physical machine) in a Kubernetes cluster. Container runtimes are a foundational component of a modern containerized architecture.

The Container Runtime Interface (CRI) allows Kubernetes to use any CRI-compliant runtime. Every Docker image can run in every container runtime.

There are various container runtimes. Each has special features and decide on trade-offs they make between performance, security and functionality.

- containerd (CNCF Graduated)
- CRI-O (CNCF Incubating)
- gVisor
- Firecracker
- kata
- lxd

- Singularity (apptainer)
- SmartOS

> *Kubernetes is deprecating Docker as a container runtime after v1.20. Don't Panic: Docker as an underlying runtime is being deprecated in favor of runtimes that use the Container Runtime Interface (CRI) created for Kubernetes. Switching to Containerd as the container runtime eliminates the middleman.*

dockershim

Dockershim implements CRI support for Docker .In the past, Kubernetes included a bridge called dockershim, which enabled Docker to work with CRI. From v1.20 onwards, dockershim will not be maintained, meaning that Docker is now deprecated in Kubernetes. Kubernetes currently plans to remove support for Docker entirely in a future version, probably v1.22. [15]

> *dockershim: Mirantis and Docker have agreed to partner to maintain and support the shim code as a standalone open source outside Kubernetes, as a conformant CRI interface for the Docker Engine API. You can find the initial prototype from dims/cri-dockerd. More information in Mirantis blog.*

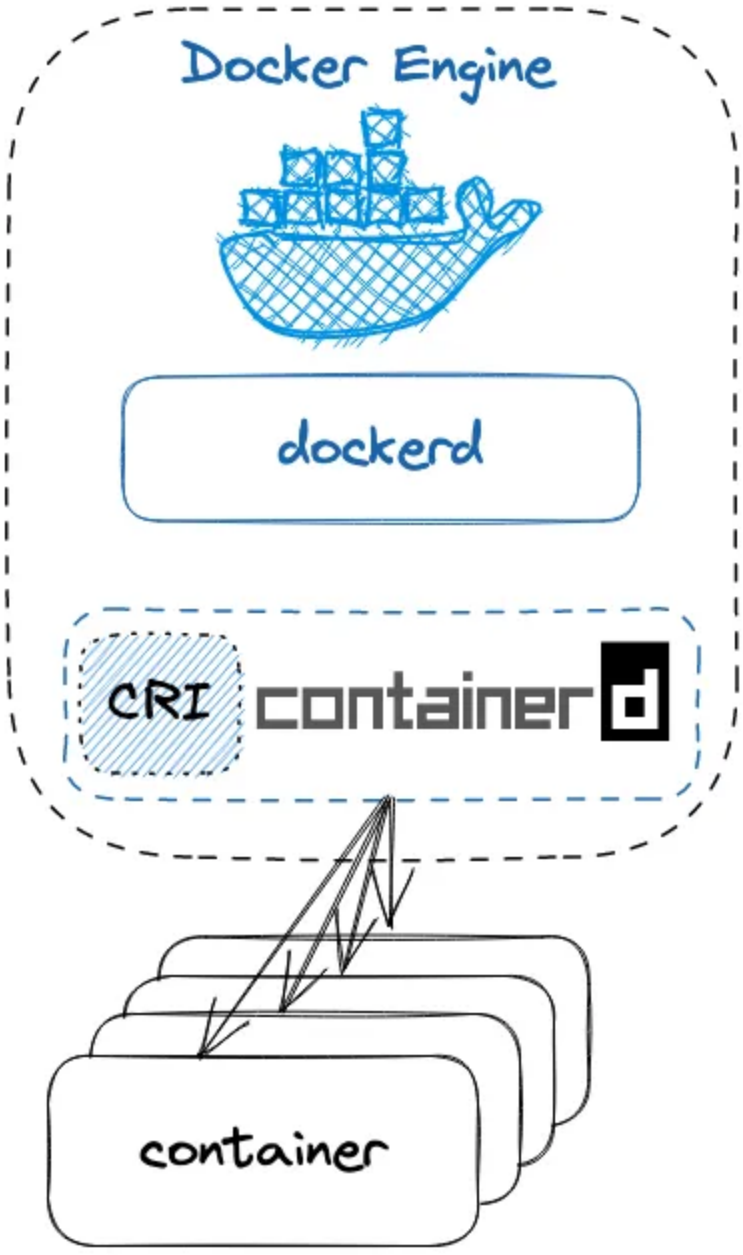
Containerd

containerd is an industry-standard container runtime with an emphasis on simplicity, robustness and portability. containerd can manage the complete container lifecycle of its host system: image transfer and storage, container execution and supervision, low-level storage and network attachments, etc.

containerd is designed to be embedded into a larger system, rather than being used directly by developers or end-users.

containerd was designed to be used by Docker Daemon; and extracted its container runtime out into a new project.

Containers will schedule directly by containerd, which means they are not visible to Docker.



containerd being used by Docker Engine * (by @furkan.turkal)

crictl

Container runtime command-line interface (CLI) is a useful tool for system and application troubleshooting. For containerd and all other CRI-compatible container runtimes, e.g. dockershim, *crictl* is recommend to use as a replacement CLI over the Docker CLI. *crictl* works consistently across all CRI-compatible container runtimes. *crictl* is designed to resemble the Docker CLI to offer a better transition experience for users, but it is not exactly the same. [16]

Many commands has direct mapping down to the name. Output format is similar. Some experimental dokcer commands have no mapping yet.

The scope of *crictl* is limited to troubleshooting, it is not a replacement to docker. Docker's CLI provides a rich set of commands, making it a very useful development tool whereas *crictl* provides just enough commands for node troubleshooting, which is arguably safer to use on production nodes.

nerdctl

nerdctl (contaiNERD CTL) is a Docker-compatible CLI for containerd, with support for compose, rootless, lazy pulling (eStargz), OCIcrypt, P2P image distribution (IPFS), image signing and verifying...

The motivation of `nerdctl` is to facilitate experimenting the cutting-edge features of containerd that are not present in Docker. Secondary goal might be potentially useful for debugging Kubernetes clusters.

ctr

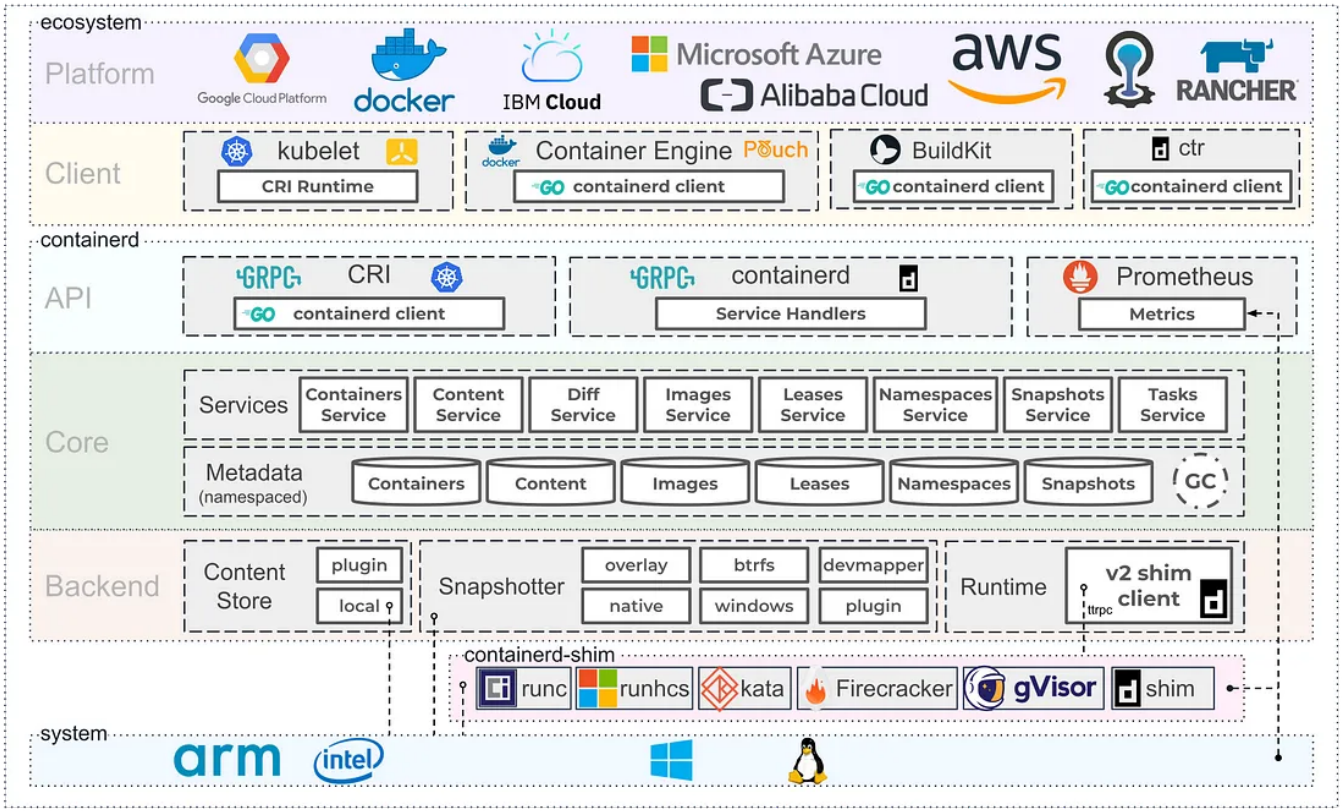
The containerd command line client is `ctr` :

- Pull a container image:
  ```
  $ ctr images pull nginx:latest
  ```
- List the images you have:
  ```
  $ ctr images list
  ```
- Run a container based on an image:
  ```
  $ ctr container create nginx:latest nginx
  ```
- List the running containers:
  ```
  $ ctr container list
  ```
- Stop the container:
  ```
  $ ctr container delete nginx
  ```

Learn more about *ctr* with hands-on experience!

Diving into containerd

If you want to learn more about containerd, jump to /docs folder.



Services

containerd initializes some services and provides gRPC service APIs for the client.

Plugins

containerd supports extending its functionality using most of its defined interfaces: customized runtime, snapshotter, content store, and gRPC

containerd uses plugins internally to ensure that internal implementations are decoupled, stable, and treated equally with external plugins.
Use `$ ctr plugins ls` to see all the plugins containerd has:

```
TYPE                            ID                PLATFORMS   STATUS
io.containerd.content.v1        content           -           ok
io.containerd.snapshotter.v1    btrfs             linux/amd64 ok
io.containerd.snapshotter.v1    aufs              linux/amd64 error
io.containerd.snapshotter.v1    native            linux/amd64 ok
io.containerd.snapshotter.v1    overlayfs         linux/amd64 ok
io.containerd.snapshotter.v1    zfs               linux/amd64 error
io.containerd.metadata.v1       bolt              -           ok
io.containerd.differ.v1         walking           linux/amd64 ok
...
```

From the output all the plugins can be seen as well those which did not successfully load. Use the following command to get more details about the plugin:

```
$ ctr plugins ls -d id==aufs id==zfs
```

containerd calls Introspectation Service to get all plugins from the server. All plugins are set during the initialization of the services server by the LoadPlugins() function. You can find an example how overlayfs plugin registration works.

ttrpc

GRPC for low-memory environments. It a lightweight protocol that doesn't require HTTP, HTTP2 and TLS. Go stdlib context package is used.

Want to initialize a *ttrpc* client? Just like that:

```
conn, err := dialer.ContextDialer(context.TODO(), timeout)
```

```
client := ttrpc.NewClient(conn)
```

continuity

continuity is a staging area for experiments in providing transport-agnostic, filesystem metadata manifest system storage.

Here is a straightforward example how can you call `AtomicWriteFile()` using continuity:

```
// AtomicWriteFile atomically writes data to a file by first writing to a temp
continuity.AtomicWriteFile(filename, bytes, 0666)
```

Deep Diving into Components

As you may noticed already, Docker is not only component that doing every magic under the hood.

Container Image

It's important to know what a container image is, how Docker builds and stores images, and how these images are used by containers.

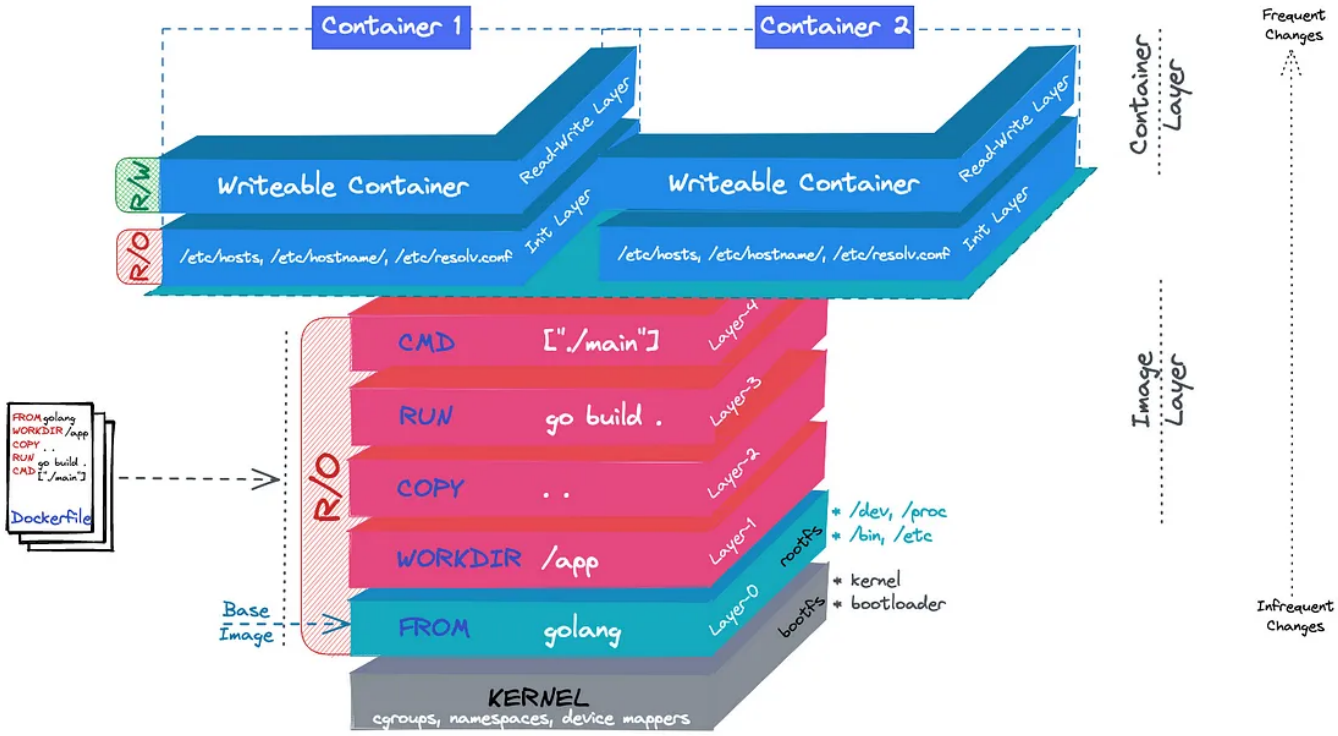
A container image is an immutable — meaning unchangeable, static file that includes executable code, so it can be deployed consistently and run an isolated process on any environment. It's notion of a parent-child relationship and image layering. It can be seen as archives with a filesystem inside.

A container image is a combination of a JSON manifest and individual file-system layers, built onto a parent or base image. These layers encourage reuse of various components and configurations, so the user does not recreate everything from scratch. Constructing layers in an optimal manner can help reduce container size

and improve performance.

Container images seems like a bit class/object concept. Image is like a class or template and then you can create any number of instances of that template and it has OCI Runtime Specification. It's a definition of the standard container.

A container image rely on open standards and can be tagged or left untagged and only searchable through a true identifier.



Container Image Layers Visualized (by @furkan.turkal)

Technically, you do not need the images to run containers! Interested in how? We will cover this in the further sections.

Storage

containers/storage is a Go library which aims to provide methods for storing and managing three types of items: layers, images, and containers.

Docker image consists of several layers. Each layer corresponds to certain instructions in your `Dockerfile` .

Dockerfile

> *Dockerfile is simply a text-based script of instructions that is used to create a container image.*

It's basically an environment in a text file. We can **RUN** any number of things that we want to configure **FROM** the is's parent image. Dockerfile ultimately ends up creating an container image that we can use to instantiate containers.

Initiating a new Dockerfile from scratch is easy with `docker init` command.

The following above overview is a visualized version of a minimal Dockerfile:

Minimal Dockerfile [5] (by @furkan.turkal)

Layer, is a copy-on-write (CoW) filesystem. Each layer is only a set of differences from the layer before it. Any layer can be stacked on top of each other. Both *adding*, and *removing* files will result in a new layer.

Image, is built up from a series of layers. Each layer represents an instruction in your Dockerfile. Multiple images can reference the same layer. Each layer except the very last one is read-only. Images are stateless.

Container, is a read-write (RW) layer. Child of an image's top wriable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. Each container has its own writable

container layer. Multiple containers can be derived from a single image. All changes (such as writing new files, modifying existing files, and deleting files) are stored in this thin writable container layer, multiple containers can share access to the same underlying image and yet have their own data state.

Container Layer, Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy.

Layers are stored similar to images internally. Each layer has its own directory in `/var/lib/docker/<driver>/layerdb/<algorithm>`. Docker stores all caches in `/var/lib/docker/<driver>`, where `<driver>` is the storage driver `overlay2` again. Read more here.

You can use the `$ docker ps -s`, to view size of a running container.

Total amount of data (on disk) that is used for the writable layer of each container is `size`, whereas `virtual size` is the amount of data used for the read-only image data used by the container plus the container's writable layer `size`.

Let's write a minimal Go file:

```
package main
func main() {
 println("Hello, World!")
}
```

Run the following command to build the image:

```
$ docker image build --tag minimal .
```

So, we can use this image in further examples.

Docker Manifest

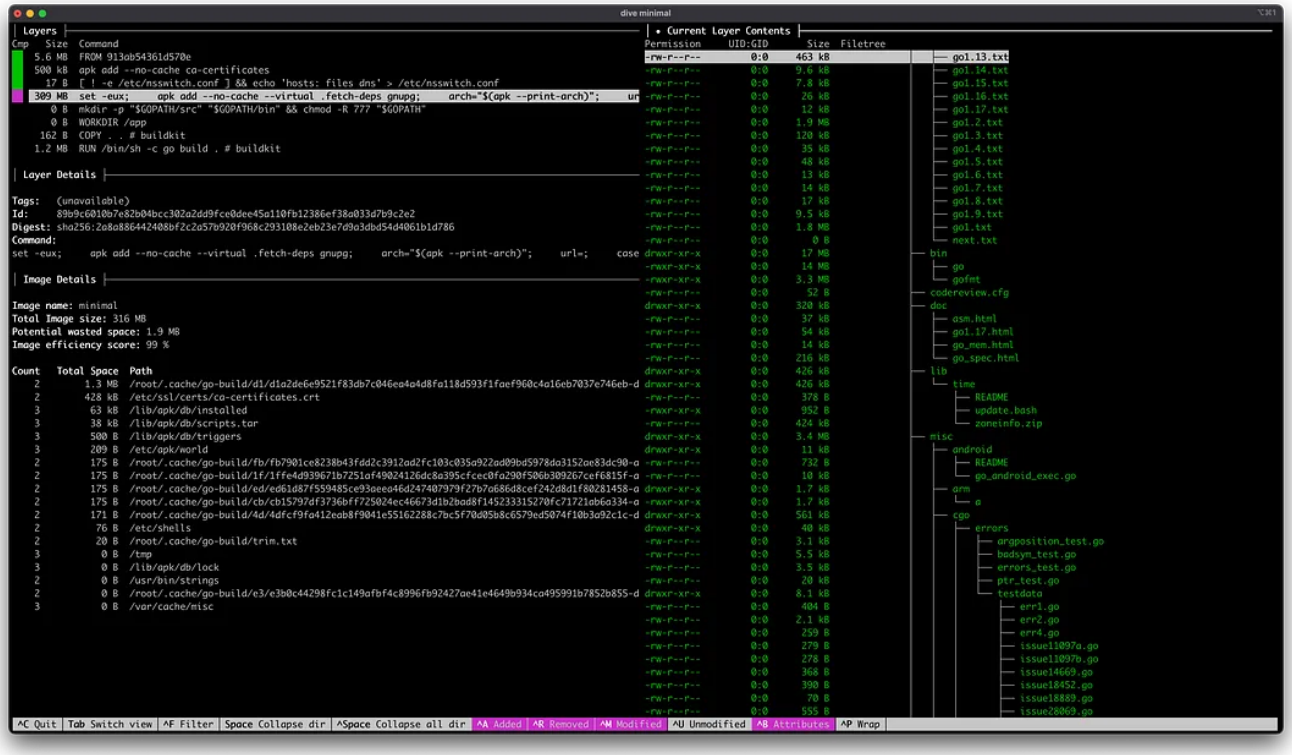
Docker Manifest, is a JSON-represented file that describes the image and provides metadata such as tags, a digital signature to verify the origin of the image, and documentation. Manifest is meant for consumption by a container runtime. [4]

Manifest Lists (aka “fat manifest”), are defined in the v2.2 image specification and exist mainly for the purpose of supporting *multi-architecture* and *multi-platform* images within an image registry. There are several IANA Media Types that Docker currently supported. If you want to view, create, and push the new manifests list object types, manifest-tool is the tool you are looking for.

> *manifest-tool, is a command line utility that implements a portion of the client side of the Docker registry v2.2 API for interacting with manifest objects in a registry conforming to that specification. (OBSOLETE)*
>
> *You can use the* `$ docker manifest …` *Thanks to Christy Perez for adding manifest command to docker/cli.*
>
> *You can use the* `$ crane manifest <IMAGE>` *to inspect the manifest.*
>
> *You can use* `$ docker commit` *command on any running container to produce a new image.*

Inspect your container image and its manifests online: https://oci.dag.dev/

If you want to dive deeper into the container image, you can use the dive tool by issuing `$ dive minimal` command to get the following information about the image:



dive Tool Output

Although *dive* says we achieved almost `%95+` efficiency score, the total size of the image is `316 MB`. This is because of unnecessary dependencies of `golang:1.17.3-alpine3.14` image. We can use multi-stage builds to optimize the image.

> *Multistage builds are useful to anyone who has struggled to optimize Dockerfiles while keeping them easy to read and maintain. With multi-stage builds, you use multiple `FROM` statements in your Dockerfile. Each `FROM` instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.*

```
FROM golang:1.17.3-alpine3.14 AS builder
WORKDIR /app
COPY . .
RUN go build  -o /usr/bin/main .
FROM scratch
COPY --from=builder /usr/bin/main /usr/local/bin/main
CMD ["main"]
```

Multi-stage Build Example

Notice that we use `scratch` here. `FROM scratch` is a no-op in the Dockerfile, and will not create an extra layer in your image; it is the smallest possible image for docker, it is empty (doesn't contain any folders or files) and is the starting point for building out images. You can not *pull* it, *run* it, or *tag* any image with the name `scratch`.

Re-run the dive tool and notice the new image size. It's `1.2MB`!

Also, it is worth pointing out the distroless images:

> `distroless` *images contain only your application and its runtime dependencies. They do not contain package managers, shells or any other programs you would expect to find in a standard Linux distribution.*
>
> `scratch` *images basically an explicitly empty image. It is just in completely empty formatter filesystem. It's just nothing. You can't pull it, run it, or tag it. Instead, you can refer to it in your Dockerfile.*

Drawbacks of Container Images

Security

Container security play a crucial role in the container world, IT organizations must monitor for fraudulent images, and train developers about applying best practices. If you want to do enterprise-hardening for your images, you can check Docker CIS Security Benchmark. To lint container image for Security, and build the best-practice images, you can use tools such dockle, hadolint, etc.

Let's say we discover a particular vulnerability in the user libraries in the image. How can we scan it? How can can we fix it? What is the affected area?

We need to find and replace the infected base image. Every single descendant child of node is going to be impacted by that vulnerability. We know the all these layers inherit from that layer. Thus, we need to understand all of the containers we are running that is going to be impacted by that vulnerability.

We should find the security vulnerabilities in container images, you must scan the image by analyzing defined packages and dependencies, and checking each of them for known security vulnerabilities. Trivy (by Aqua Security) and Clair (by Quay) are great scanners since it uses up-to-date vuln-list and actively developing.

`docker scout` analyzes image contents and generates a detailed report of packages and vulnerabilities that it detects. It can also provides you with suggestions for how you can remediate issues discovered by the image analysis.

Do not forget to check Container Security book by @lizrice! Batuhan also sharing Docker news in his substack in terms of security and cloud-native perspective.

Use secure by default container images: Chainguard Images by Chainguard is a collection of container images designed for minimalism and security. Many of these images are distroless; they contain only an application and its runtime dependencies. There is no shell or package manager.

Storage

Images are stored in container registries. Container Registry is an Open Container Initiative (OCI) compliant registry. It maked easy for you as a developer to store,

share, and manage container images. Container registries are really just "Tarballs As A Service".

Docker Registry HTTP API V2 specification adopted in the Open Container Initiative (OCI) as distribution-spec.

To optimize your storage resource, you probably want to remove unused container images, since stopped containers are not automatically removed.

To reducing the size and speeding up pulling container images, you can use lazy pulling with eStargz. (Here is my deep-dive notes.)

Garbage collection is an important aspect of managing container resources efficiently, and containerd includes a garbage collection mechanism to reclaim unused or expired resources.

Encrypting

By encrypt the layers of a container image, you can get stricter trust requirements to be able to ensure end-to-end encryption from build to runtime. @lumjjb proposed an idea to add Encrypted Layer Mediatype on the opencontainers/image-spec. If you want to go further on this, you should check the CNCF Webinar!

> *The imgcrypt library provides API exensions for containerd to support encrypted container images for use by containerd to decrypt encrypted container images, which relies on the ocicrypt library under the hood, which is the OCI image spec implementation of container image encryption.*

Eventually, we thought that why not to support layer encryption in the container registries? So, we decided to create a proposal for the Harbor registry! For the SBOM support, we filed another proposal.

Signing

Imperative to incorporate security focused steps such as scanning the image for the validating the integrity of the images to protect against tampering. [10] Digital signing of image content at build time and validation of the signed data before use

protects that image data from tampering between build and runtime, thus ensuring the integrity (between publisher and consumer) and provenance of an OCI artifact. This is where Content Trust in Docker comes in and why tools such Sigstore/Cosign and Notary are born! To get more details about cosign, you can read Signed Container Images by @dlorenc! Curious about differences between Notary and Cosign?

> *If you are a Kubernetes user, you probably do not want to miss these brilliant ideas: Ensure Content Trust on Kubernetes using Notary and Open Policy Agent by @siegert-maximilian! And Verify Container Image Signatures in Kubernetes using Notary or Cosign or both by @hamsen.christoph!*

Linux Foundation, BastionZero and Docker recently announced OpenPubkey project — read more about OpenPubkey and Sigstore.

Learn more about cosign, rekor, fulcio, and gitsign!

Software Bill of Materials

To secure your software supply chain, inevitably, it should start with knowing what software is being used. You have to produce a list of what your software is made of such as libraries, dependencies, packages, etc., We can call it **software ingredients** shortly. This list of "ingredients" is known as a Software Bill of Materials (SBOM). SBOM is a complete, formally structured list of components, libraries, and modules that are required to build (i.e. compile and link) a given piece of software and the supply chain.

> *If you want to go further here, you should consider check awesome-sbom repository for related tools, frameworks, blogs, podcasts, and articles! Maintaining by @developer-guy!*

Use syft tool by Anchore to easily generate SBOMs for your container images then scan it with grype to find vulnerabilities!

The experimental `docker sbom` command allows you to generate the SBOM of a container image.

Containers

A Linux container is nothing more than simply isolated and restricted ~~Linux processes~~ **boxes for running one or more processes** inside that runs on Linux. [6] What we mean by *box* here that the *isolated* process for start has its own process namespace. A containerized process interact with the kernel through *system calls* and needs permissions just the same way that a regular process does. A container is a self contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system. A bare minimum container is just a single executable file inside.

If we shell into a container, we would see just the processes running inside that container it has its own process namespace. Typically the way that containers are used by one process per container. Container process tied in with the lifecycle of the container itself: starting/ending a container, starts/kills the container process. Entire lifecycle is tightly coupled.

With containers, it's expected that all the dependencies. Pretty much above the kernel are packaged inside of the container. Containers share the host's kernel. Within a container, you do not see the host's entire filesystem; instead, you see a subset as the root directory is changed when the container is created. When you run the container in a OS, you actually do not install anyhing. It sits above the OS and it's own world.

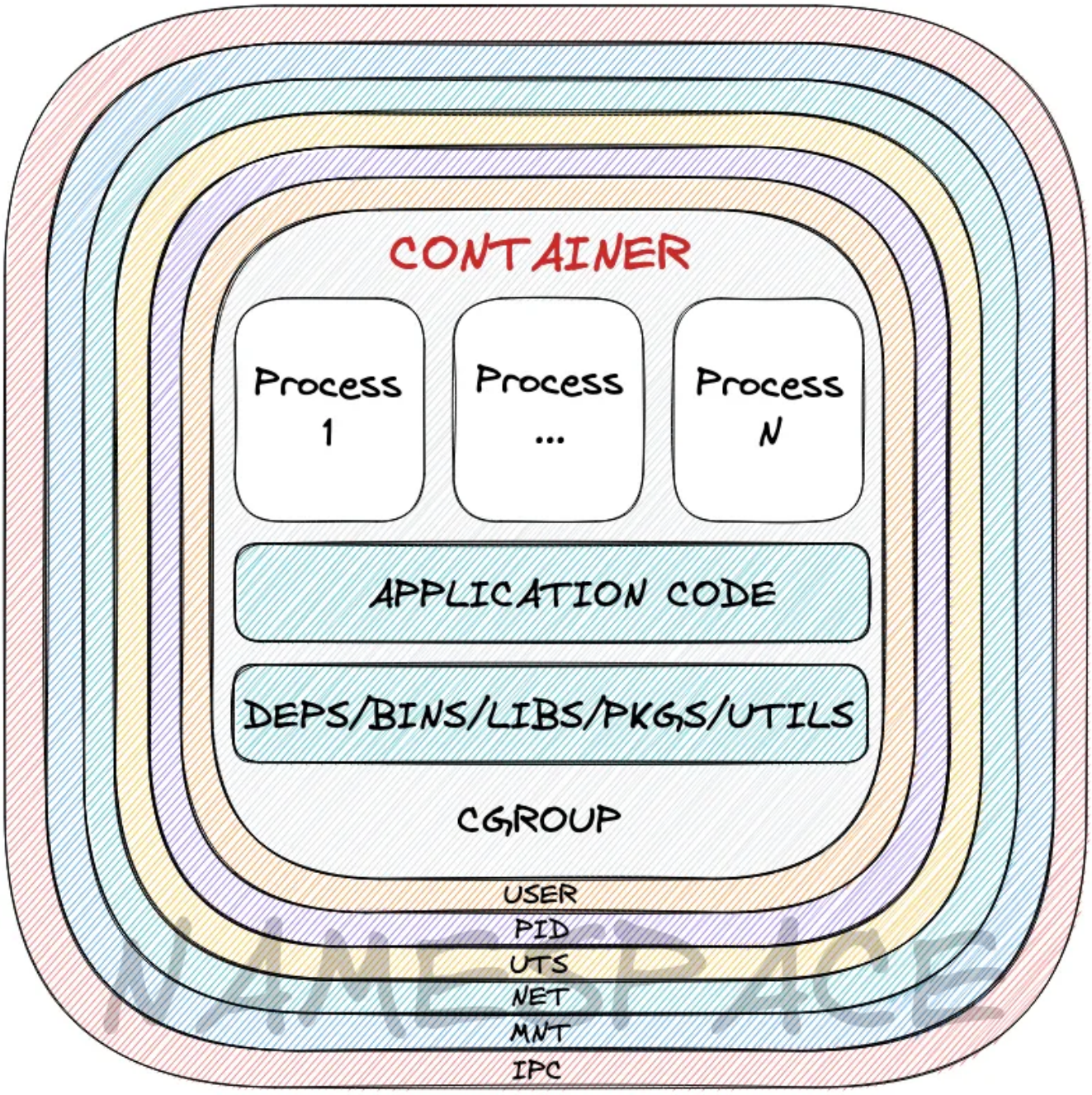
An awesome reading reference: Containers Aren't Linux Processes!

Namespaces

The Linux kernel has a concept to common functionality and makes it available to many applications running on the system. This concept is called namespaces. It's a form the foundation of container technology.

The Linux namespaces exist for a variety of abstractions including: file systems, user management, mounted devices, processes, network and several more.

Namespaces allow containers to have isolation on 6 + *1* levels. The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.



Linux Namespaces Isolation (by @furkan.turkal)

USER: (USER): [CLONE_NEWUSER]: isolates user and group IDs. A process's UserID and GroupID can be different inside and outside a user namespace

PID: (Process ID): [CLONE_NEWPID]: isolates process IDs, allow each container to have its own `init` . A process has two PIDs: inside the namespace, and outside the namespace on the host system. Better check An Init System inside the Docker Container by @BeNitinAgarwal

UTS: (UNIX Time-sharing System): [CLONE_NEWUTS]: isolates the hostname (uname() syscall) and the Network Information Service (NIS) domain name

NET: (Network): [CLONE_NEWNET]: isolates the network interface controllers (i.e., devices, IP addresses, IP routing tables, /proc/net directory, port numbers, etc.)

MNT: (Mount): [CLONE_NEWNS]: isolates filesystem mount points (mount() and umount() syscalls)

> ***findmnt*** *will list all mounted filesystems or search for a filesystem in* `/etc/fstab` , `/etc/mtab` *or* `/proc/self/mountinfo` . *If* device *or* mountpoint *is not given, all filesystems are shown.*

IPC: (Inter-process Communication): [CLONE_NEWIPC]: isolates System V IPC objects, POSIX message queues

> ***ipcmk*** *allows you to create System V inter-process communication (IPC) objects: shared memory segments, message queues, and semaphore arrays.*
>
> ***ipcs*** *shows information on System V inter-process communication facilities.*

There have been discussions about having a time namespace.

> *System Call (syscall) is a programmatic interface that the user space code uses to make these requests of the kernel. It is a way for programs to interact with the operating system. It has to ask the kernel to do it on the application's behalf. All programs needing resources must use system calls.*

Namespace isolation means that groups of processes are separated such that they cannot "see" resources in other groups: [11]

- Processes in different UTS namespaces see their dedicated hostname and can edit their hostname independently.
- Processes in different User namespaces see a their dedicates list of users and can add or remove users without affecting other processes.
- Processes get a different PID for each PID namespaces they are part of, each PID namespace has its own PID tree.
- Process is always in exactly one namespace of each type.

By putting a process in a namespace, you can restrict the resources that are visible to that process. The origin of namespaces date back to the Plan 9.

> ***lsns*** *lists information about all the currently accessible namespaces or about the given namespace. Note that lsns reads information directly from the* `/proc` *filesystem and for non-root users it may return incomplete information.*
>
> ***unshare*** *lets you run a process with some namespaces unshared from the parent. "Unshare" means that, rather than sharing namespaces of its parent, the child is going to be given its own.*

Namespace concept is expanded with the help of cgroups to apply resource management and prioritization: Linux containers make use of Control Groups (cgroups) to limit the resources (i.e., memory, CPU, I/O, network, etc.) consumed by a container. This prevents a container from consuming all host resources. This is why well-tuned cgroups are important from the security perspective.

Control Groups (cgroups)

Cgroups allowes us to apply certain degree of isolation and restrict what a process is able to do: capabilitiees, resource limits, etc. It's a fundamental notion of the runtime definition of a container. When you start a container, the runtime creates new cgroups for it.

Cgroup is not only for imposing limitation on CPU and memory usage; it also limits accesses to device files such as `/dev/sda1`.

> ***Linux Capabilities*** *are used to allow binaries (executed by non-root users) to perform privileged operations without providing them all root permissions. Capabilities can be assigned to a processes/thread to determine wheter that processes/thread can perform certain actions. @wifisecguy have a great series: "Understanding Linux Capabilities Series" I, II, III*
>
> ***lscgroup*** *(in libcgroup-tools package) provides obtaining information about controllers, control groups, and their parameters.*
>
> ***capsh*** *provides a handy wrapper for certain types of capability testing and environment creation and some debugging features useful for summarizing capability state.*

To limit total number of processes allowed within a control group, there is a control group called *pid,* which can be prevent the effectiveness of a fork bomb.

The Linux Kernel communicates information about cgroups through a set of pseudo-filesystems that typically reside at `/sys/fs/cgroup`. From inside the container, the list of its own cgroups is available from the `/proc`. Each cgroup has an interface file called `cgroup.procs` that lists the PIDs of all processes belonging to the cgroup, one per line. [12]

Rootless Containers

By default, containers run as root. But, attacker who can take control of a process inside a container still has to somehow escape the container, they will be *root* on the

host machine eventually.

Rootless containers built on Linux Kernel `user_namespaces(7)` (UserNS) for emulating fake privileges that are enough to create containers.

> *Privilege Escalation means extending beyond the privileges you were supposed to have so that you can take actions that you should NOT be permitted to take.*

Rootless containers refers to the ability for an unprivileged user to create, run and otherwise manage containers. [9] An unprivileged user does not have any administrative rights, and do not have the ability to ask for more privileges. An unprivileged user, manages a user and group range in which containers will run.

Docker 19.03 provides almost full features for Rootless mode (rootless runc, containerd, and BuildKit), including support for port forwarding and multi-container networking. Limiting resources feature implemented in Docker 20.10 using Control Group v2. Which means you can not manage rootless containers in Control Group v1. Enabling and delegating cgroup v2 controllers to non-root users requires a recent version (≥ 244) of systemd is recommended. Older systemd does not support delegation of `cpuset` controller. Kernel older than 5.2 is not recommended due to lack of freezer.

> *cgroup v2 focuses on simplicity, unified as `/sys/fs/cgroup/$GROUPNAME`. It's eBPF-oriented; the device access control is implemented by attaching an eBPF program (`BPF_PROG_TYPE_CGROUP_DEVICE`) to the file descriptor of `/sys/fs/cgroup/`$GROUPNAME directory.*
>
> *To check if you are using v2: `/sys/fs/cgroup/cgroup.controllers` should present.*
>
> *If you want to learn more about v2, you should take a look at "The current adoption status of cgroup v2 in containers" by @AkihiroSuda*

There are some libraries to limit some privileged actions on userspace.

- rootless-containers/slirp4netns: provides user-mode networking ("slirp") for unprivileged network namespaces: limit network creation/interaction

- containers/fuse-overlayfs: implementation of overlay+shiftfs in FUSE for rootless containers: interact root filesystem and User ID, Group ID handling

Docker and other container engines uses RootlessKit to protect the real root on the host from potential container-breakout attacks.

Namespaces

Namespaces are a fundamental aspect of containers on Linux!

> ***Namespaces*** *are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces. [7]*
>
> *A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes. One use of namespaces is to implement containers. [8]*

The Linux Namespaces originated in 2002 in the 2.4.19 Kernel! Additional namespaces were added beginning in 2006 and continuing into the future.

Adequate containers support functionality was finished in kernel version 3.8 with the introduction of User namespaces.

Kernel

> *The* ***kernel*** *is a computer program that is the core of a computer's operating system, with complete control over everything in the system.*

The Kernel is the “portion of the operating system code that is always resident in memory”, and facilitates interactions between hardware and software components. A full kernel controls all hardware resources (e.g. I/O, memory, Cryptography) via device drivers, arbitrates conflicts between processes concerning such resources, and optimizes the utilization of common resources e.g. CPU & cache usage, file systems, and network sockets.

The kernel’s interface is a low-level abstraction layer. When a process requests a service from the kernel, it must invoke a system call, usually through a wrapper function.

Docker Kernel Module

Kernel modules are pieces of code that can be dynamically loaded into the Linux kernel to extend its functionality or provide specific features. In the case of Docker, it leverages kernel features and modules to implement containerization and storage capabilities.

The overlay2 storage driver is a copy-on-write (CoW) mechanism that allows multiple layers of container images to be stacked on top of each other. This driver leverages the overlay filesystem, a feature provided by the Linux kernel, to overlay multiple directories onto a single mount point, creating a unified and layered view of the file system.

If you want to check module overlay path: `$ modprobe overlay` Your modules folder stored under such as: `/lib/modules/5.11.0-38-generic` .

> *.ko (Kernel Object) files are Loadable kernel modules that are used to extend the kernel of the Linux Distribution. They are used to provide drivers for new hardware like IoT expansion cards that have not been included in the Linux Distribution.*
>
> `$ lsmod` *is a trivial program which nicely formats the contents of the /proc/modules, showing what kernel modules are currently loaded.*
>
> `$ depmod` *creates a list of module dependencies by reading each module under* `/lib/`

> *`modules/version` and determining what symbols it exports and what symbols it needs. Run this to re-create the module dependency list.*

> ***TIP:** If you want to know container's mount device on the host, run the following command in the container:* `$ cat /proc/cmdline` *It returned:* *`BOOT_IMAGE=/boot/vmlinuz-5.11.0-34-generic root=PARTUUID=81df8e86-5e57-4b2a-96be-cd72fcc3d492`*
>
> vmlinuz *is the name of the Linux* kernel executable. *It's compressed Linux kernel. It is* bootable. *It should not be confused with* vmlinux. *And located in the /*boot *directory.*
>
> *On the host machine run the:* `$ findfs PARTUUID=81df8e86-5e57-4b2a-96be-cd72fcc3d492` *It returned:* `/dev/sda1`
>
> *Interested in differences between UUID and PARTUUID?*
>
> *The /dev directory contains the special device files for all the devices. These files are created during installation.*

Open Container Initiative

The Open Container Initiative (OCI) is a lightweight, open governance structure (project) for the express purpose of creating open industry standards around container formats and runtime, formed under the auspices of the Linux Foundation, for the express purpose of creating open industry standards around container formats and runtime. The OCI was launched on June 22nd, 2015 by Docker, CoreOS and other leaders in the container industry.

The OCI currently contains two specifications: the Runtime Specification (runtime-spec) and the Image Specification (image-spec). The Runtime Specification outlines how to run a "filesystem bundle" that is unpacked on disk. At a high-level, an OCI implementation would download an OCI Image, then unpack that image into an OCI

Runtime filesystem bundle. At this point, the OCI Runtime Bundle would be run by an OCI Runtime. [3]

Do not forget to join OCI Slack channel to stay in the loop!

OCI Runtime Spec

The Open Container Initiative develops specifications for standards on Operating System process and application containers. The Open Container Initiative Runtime Specification (opencontainers/runtime-spec) aims to specify the configuration, execution environment, and lifecycle of a container.

A container's configuration is specified as the `config.json` for the supported platforms and details the fields that enable the creation of a container. If you want to validate JSON Schema against to configuration, use the validate.go file. Here are the some schema layouts:

- config-schema.json — the primary entrypoint for the configuration schema
- state-schema.json — the primary entrypoint for the state JSON schema
- defs.json — definitions for general types

The execution environment is specified to ensure that applications running inside a container have a consistent environment between runtimes along with common actions defined for the container's lifecycle. [13]

Runtimes MUST support the following operations:

- `state <container-id>`
- `create <container-id> <path-to-bundle>`
- `start <container-id>`
- `kill <container-id> <signal>`
- `delete <container-id>`

runc

`runc` is a CLI tool for spawning and running containers on Linux according to the OCI specification. runc starts the container using the files contained in the container image, and by telling the Linux kernel to start up the process(es) in the appropriate namespace, cgroups context, etc. [2]

Docker is donating its container format and runtime, runc, to the OCI to serve as the cornerstone of this new effort.

The runtime of Go itself is multi-threaded. `setns(2)` is not available for multi-threaded processes. Therefore, runC execs the C process before the language runtime starts. However, this needs to be done before reaching the main function of Go. Therefore, the create subcommand calls the init subcommand, and the init subcommand calls the C process in the `init()` function. The `nsenter` package registers a special init constructor that is called before the Go runtime has a chance to boot. [14]

runc re-execs itself and use a module called *libcontainer* written in C for setting up the environment before the container process starts.

> *libcontainer provides a native Go implementation for creating containers with namespaces, cgroups, capabilities, and filesystem access controls. It allows you to manage the lifecycle of the container performing additional operations after the container is created.*

crun

crun is a fast, lightweight and fully featured OCI runtime and C library for running containers. It is yet another implementation of OCI Runtime Spec fully written in C,

led by Red Hat. It has been around since several years now and it's cri-o default runtime.

crun aims to be also usable as a library that can be easily included in programs without requiring an external process for managing OCI containers.

crun is `-49.4%` faster for running 100 times `/bin/true` than runc and has a much lower memory footprint.

railcar (archieved)

railcar is a rust implementation of the OCI runtime-spec. It is similar to the reference implementation `runc`, but it is implemented completely in Rust for memory safety without needing the overhead of a garbage collector or multiple threads.

You had better to not miss "Building a Container Runtime in Rust" article by @vishvananda

youki

youki is an implementation of the OCI runtime-spec in Rust! The motivation behind the project is really cool: memory safety, potential to be faster, use less memory! This project is inspired by *railcar*. Don't forget to check design and implementation details of the project.

> *oci-runtime-rs is a OCI Runtime and Image Spec in Rust. It is a library provides a convenient way to interact with the specifications defined by the OCI.*

OCI Runtime Tools

runtime-tools is a collection of tools for working with the OCI runtime-spec:

- `$ oci-runtime-tool generate` : generates configuration JSON for an OCI bundle. OCI-compatible runtimes like runc expect to read the configuration from `config.json` .
- `$ oci-runtime-tool validate` validates an OCI bundle. The error message will be printed if the OCI bundle failed the validation procedure.
- `$ sudo RUNTIME=runc validation/default/default.t` : runs the runtime validation suite. For example, jump to *youki*'s `integration_test.sh` to see how it runs the test cases. *crun* uses `oci-runtime-validation` to run tests. *runc* mostly uses bats-core to run integration test cases.

Conclusion

Exploring the inner workings of Docker has revealed a **complex and intricate** ecosystem of technologies and collaborations. Throughout this deep dive into high and low level architectures, from Docker Desktop to Linux cgroups, it has become evident that *containers world* is not as straightforward as it may have initially seemed.

The journey of understanding Docker has been filled with numerous valuable insights and learnings:

- Gained a deeper appreciation for the level of effort and expertise required to seamlessly integrate the various components that make containers such a powerful and versatile.
- Enhanced our understanding of this technology but also highlighted the collaborative efforts of countless individuals working tirelessly to make this process seamless from end to end.

- Embracing the complexity and investing in continuous learning will empower us to harness the power of containers and contribute to the thriving containerization ecosystem.

Thanks to Batuhan Apaydın and Yasin Taha Erol for your contribution!

What's Next

It is crucial to emphasize the importance of staying up-to-date with the latest industry news, projects, and new releases. The containerization landscape is constantly evolving, with advancements and innovations being made regularly. By actively following the industry trends, we can ensure that our knowledge remains current and relevant.

Keep yourself always up-to-date by following the news and participating in discussions on Slack and mail groups:

- **Docker:** Blog — GitHub — Twitter —Slack — HNRSS — Reddit
- **OCI:** GitHub — Twitter — Slack — Mail Group
- **Containerd:** GitHub — Twitter — CNCF Slack *(#containerd)*

By expanding your skill set to include learning container orchestration platform such as Kubernetes, *(which is I highly recommend)* you can further enhance your hands-on abilities and gain a deeper understanding of how it efficiently manages and orchestrates containers within a complex environment.

References

[0]: https://www.aquasec.com/cloud-native-academy/docker-container/docker-containers-vs-virtual-machines/
[1]: https://www.tiejiang.org/23394.html
[2]: https://www.pngegg.com/en/png-pbils
[3]: https://opencontainers.org/about/overview/
[4]: https://stackoverflow.com/a/47023753/5685796
[5]: https://ops.tips/blog/dockerfile-golang/
[6]: https://iximiuz.com/en/posts/oci-containers/
[7]: https://en.wikipedia.org/wiki/Linux_namespaces
[8]: https://man7.org/linux/man-pages/man7/namespaces.7.html
[9]: https://rootlesscontaine.rs/
[10]: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/managing_containers/signing_container_images
[11]: https://blog.ramlot.eu/containers/
[12]: https://facebookmicrosites.github.io/cgroup2/docs/create-cgroups.html
[13]: https://github.com/opencontainers/runtime-spec/blob/main/spec.md
[14]: https://www.reddit.com/r/rust/comments/pweqkb/comment/hejdot0
[15]: https://www.aquasec.com/cloud-native-academy/container-security/container-runtime-interface/
[16]: https://kubernetes.io/blog/2018/05/24/kubernetes-containerd-integration-goes-ga/#crictl
[17]: https://www.oreilly.com/library/view/kubernetes-in-action/9781617293726/
[18]: https://developpaper.com/practice-of-docker-file-system/
[19]: https://www.alibabacloud.com/blog/cri-and-shimv2-a-new-idea-for-kubernetes-integrating-container-runtime_594783
[20]: https://vitalflux.com/docker-images-containers-internals-for-beginners/
[21]: https://www.oreilly.com/library/view/container-security/9781492056690/

I hope you found this article insightful and enjoyed exploring the inner workings of Docker in a comprehensive manner. Feel free to ping me on Twitter or GitHub

anytime.

> “Thank you, and have a very safe and productive day!”

Furkan Türkal

Docker Container Cloud Native Linux Cncf

Follow

Written by Furkan Türkal

213 Followers

Nothing is true, everything is permitted — https://github.com/Dentrax/

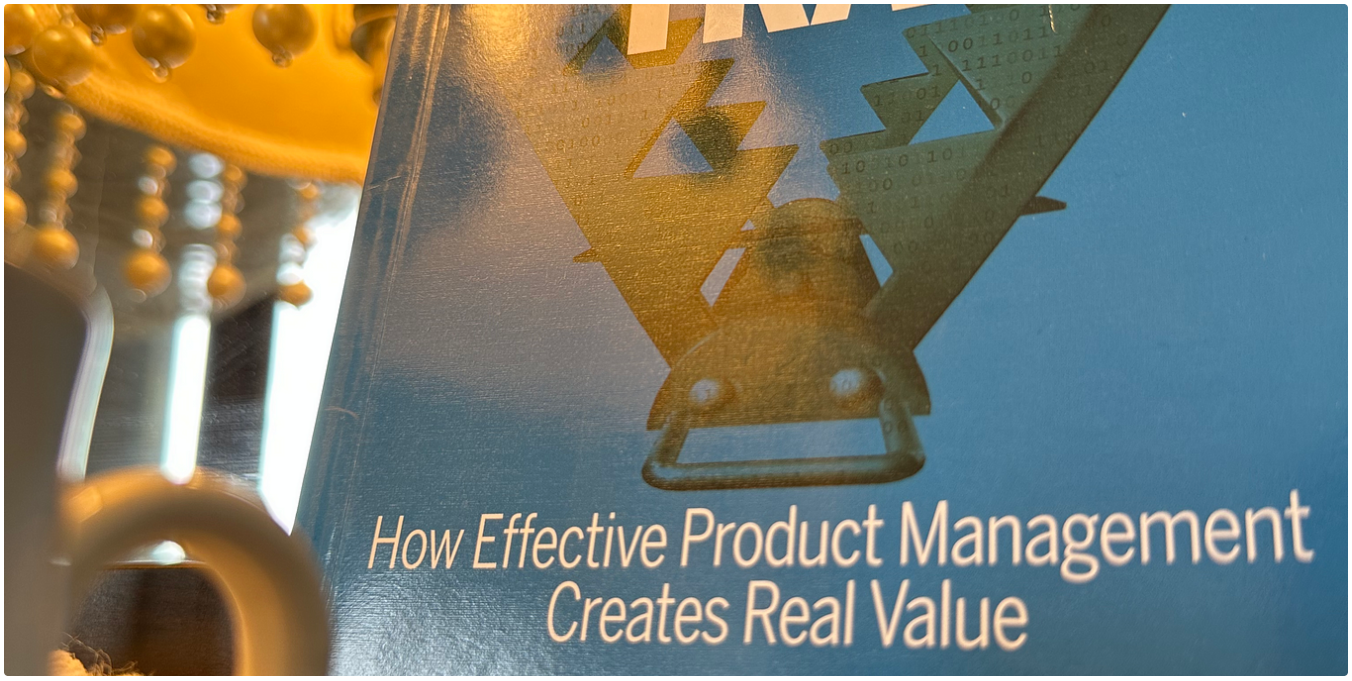
More from Furkan Türkal

Furkan Türkal in Trendyol Tech

Secure Types | Memory Safety with Go

Have you ever considered about how anti-cheating systems work? How do hackers analyze and alter memory? We always store sensitive...

Aug 24, 2020 206

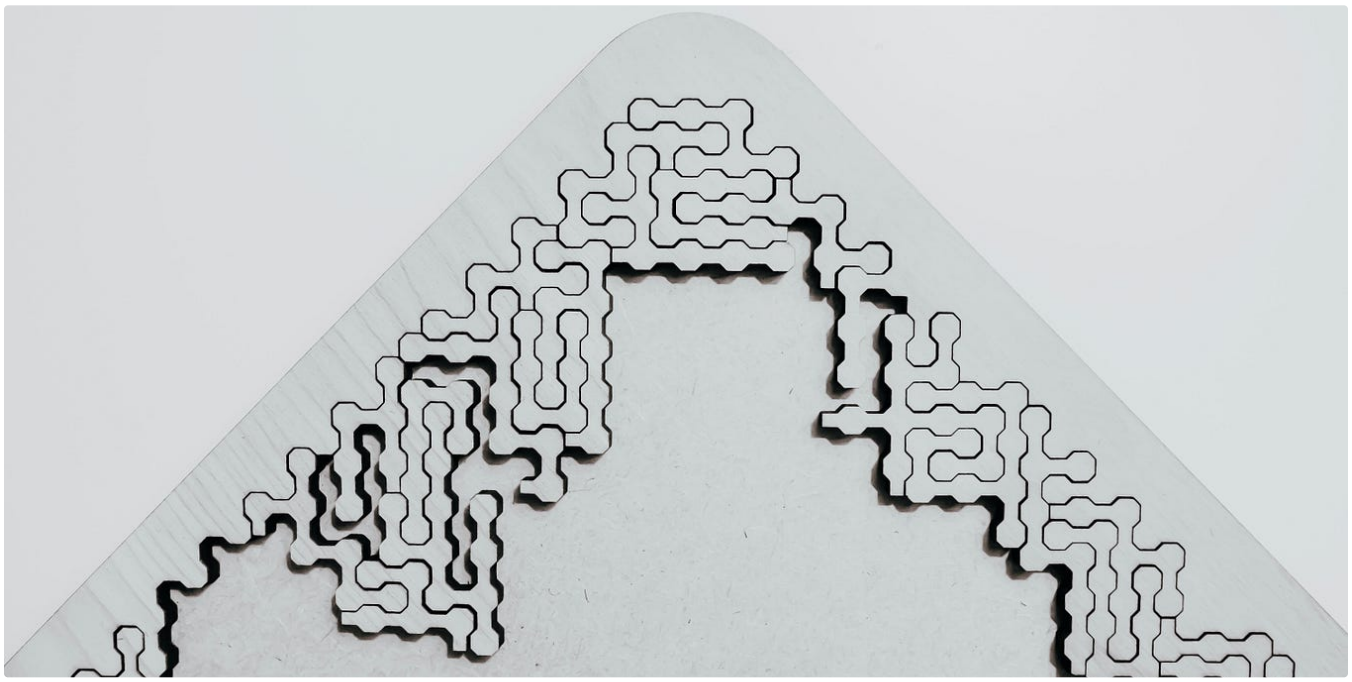



Furkan Türkal

Escaping the Build Trap: summary, highlights, and my reviews

Escaping The Build Trap book provides a good approach to effective product management!

May 17, 2023 86



Furkan Türkal in Trendyol Tech

Contributing the Go Compiler: Adding New Tilde (~) Operator

Compilers have always had challenges to be solved, even today. Go is not so different too. There are so many things to do, so many problems...

Furkan Türkal

How to survive your first year on a Software Engineer path just after graduation?

This day is some kind of special. This is my 365th day at the company where I first started just after under-graduation. I have decided to...

Jan 2, 2021 377

See all from Furkan Türkal

Recommended from Medium

Harendra

Top 10 Most Used Open Source SaaS Products

Start using top open-source products for your daily tasks and save money

Sep 24 715 3

Rahul Sharma in AWS in Plain English

I have Asked This SSH Question in Every AWS Interview—And Here's the Catch

When I interview people, I always ask questions about problems that people face in the real world.

Sep 16 812 36

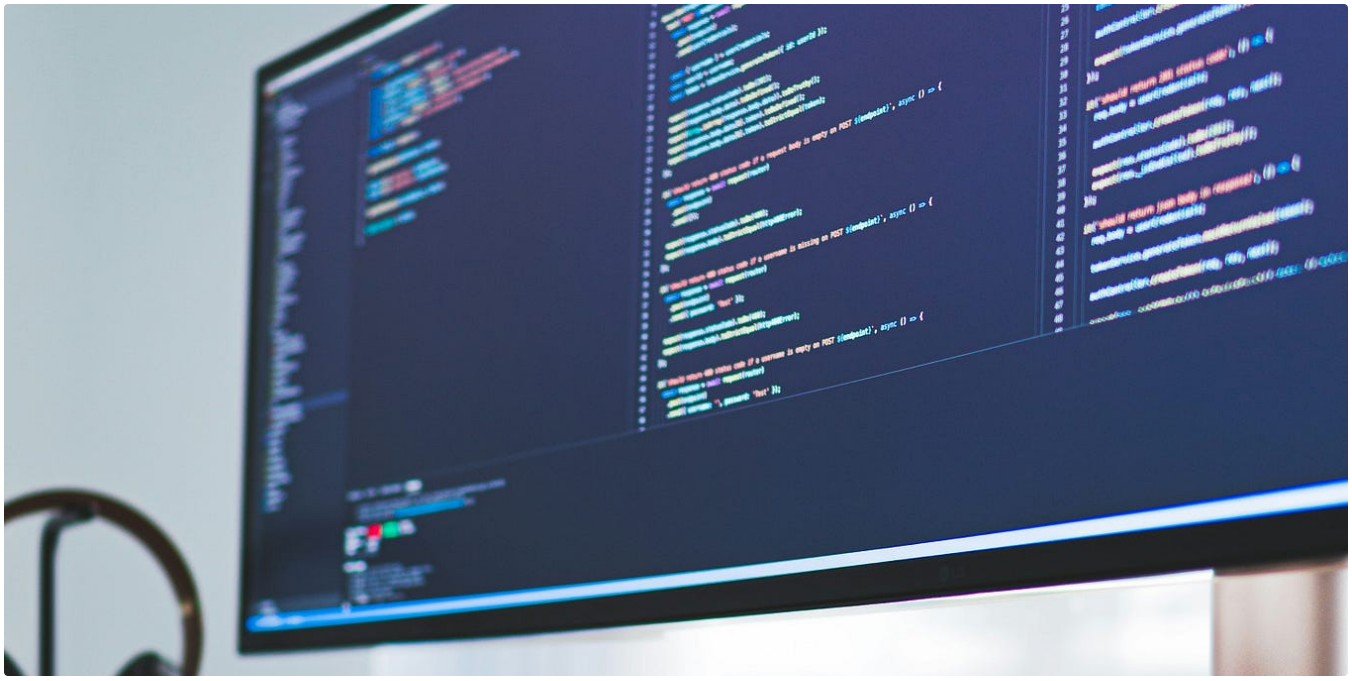
Lists

Coding & Development
11 stories · 825 saves

General Coding Knowledge
20 stories · 1604 saves

Staff Picks
742 stories · 1334 saves

Natural Language Processing
1733 stories · 1305 saves



Rahul Beniwal in Level Up Coding

18 Programming Concepts You've Never Heard of (But Should!)

Unlock Hidden Programming Gems to Boost Your Coding Superpowers

Mouri Roy

10 Best Kubernetes Tools to Use in 2024

Kubernetes has a large rapidly growing ecosystem. It is a type of open-source platform for automating tasks, deployments, and management of...

20

RED TEAM

Shadows in Rust: Crafting Advanced Windows Malware

“How Rust is Revolutionizing Malware Development and Evasion Techniques”

Sep 22 29


Markus Buchholz

Choose Simplicity. Install ArchLinux

The article below provides simple steps to install ArchLinux on your host machine. If you are already familiar with the Linux OS, these...

Apr 17 8