The Engines that run our Kubernetes Workloads

Target audiance: This article is a deep dive for people breathing the Kubernetes lifestyle and know it inside out. This small post looks into the different OCI runtime's and how they perform. It does not explain every component of containers nor Kubernetes as it is expected to be known. For a one look compassion see the table page 4.

Kubernetes has become the standard for container orchestration. It is not just a software tool, it is a framework with extensive extensibility features. There are entire businesses that are built on top of Kubernetes and offer their service. Tools like ArgoCD and Crossplane are build for Kubernetes. There is even a new category of operating systems like CoreOS, Bottlerocket and Talos that are build purposely to run containers. All that happened in just 10 years. **Happy late birthday K8s!**

Kubernetes is dominating every Ted-talk and system architecture presentation, the actual engine that drive our workloads sometimes gets forgotten. Kubernetes is just an orchestrator, and its primary (but not only) task is to manage containers. It does not run any containers itself. That gets delegated to the container runtime interface.

A Little History

Initially, Kubernetes was build to manage Docker Containers. Docker is the technology that made containers accessible to the mainstream. Containers were not an entirely new technology. LXC containers existed before and Linux namespaces are a thing since 2002, but Docker made it so easy to use containers. By handling software packaging, distribution and all low level network and device configuration, Docker allowed every developer to start any application in an isolated environment. Promising to finally overcoming the famous: *It works on my machine* meme.
Even the Kubernetes core team admits that Kubernetes would've been such a success without Docker. In an alternative universe, we are all using Mesos.

At some point, a lot of different people and companies tried to solve similar problems regarding containers. Unlike some dinosaur companies with mediocre stubborn management, they agreed to build some common interfaces for their solutions so that parts of their implementations could be swapped while relying on certain standards. This led to the creation of the Open Container Initiative (OCI), Container Network Interface (CNI), Container Storage Interface (CSI) and Container Runtime Interface (CRI).
Today, I want to talk about the Container Runtime Interface (CRI). With version 1.24, Kubernetes has overcome its dependence on Docker and now only relies on any CRI compliant runtime to start containers. And there are quite a few.

CRIs

The most common CRIs are containerd, CRI-O and Mirantis Container Runtime. Of these three, containerd is the most wildly used option. The containerd project was originally developed by Docker Inc and still is the core of Docker, but its ownership was since donated to the CNCF.
When a new Pod is scheduled to a node, the kubelet communicates with containerd over the container runtime interface to actually pull the image and start the container with the configuration specified in the pod definition. Containerd is now in charge to set up the Linux namespace, mount volumes or devices and apply seccomp profiles and much more for every new container. That's a lot of tasks, and not quite according to the UNIX philosophy to *do one thing and do it well.*

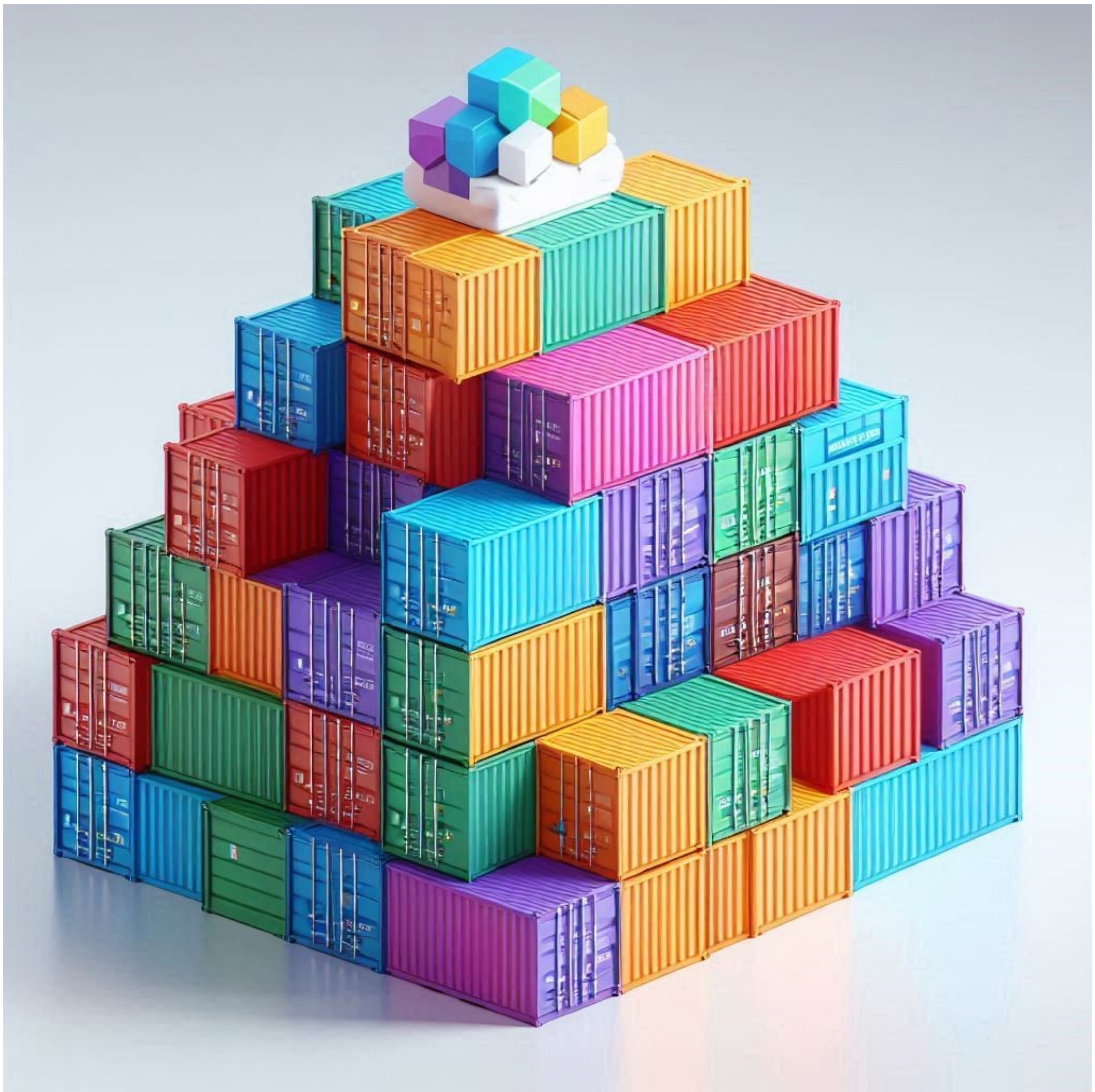


Figure 1: Kubernetes Container Stack Thumbnail

Digging deeper

So containerd doesn't do that all on its own. It uses something I call the container engine (others also call it OCI runtime, but I believe this would cause some confusion). By default, containerd calls `runc`. The runc binary actually spawns the container process with all its applied properties. It does not pull images, manages the containers file system, sets up networking or monitors the container process. Overly simplified it is just a `spawn` syscall with a bunch of extra properties for some isolation, but it still runs on the same kernel since containers are NOT VMs.

Most of the tooling for Kubernetes and containerd is written in go. Go is a great language, but some people argue that it is not the best choice for low level system functions. Discourse and different opinions are great and drive innovation. And since a bunch of smart people choose to standardize container interfaces, everyone can create their own OCI compliant implementation and use it with the existing CRIs and tools.
And that is exactly what people did, and I will take a look at some of them and compare them.

Benchmarking Container Engines

Containers are created all the time. We don't even think about them anymore when we schedule a deployment with 42 replicas. But since it happens so frequently, we might take an occasional look at what actually happens at a low level and how it impacts our cluster's efficiency. Do container engines impact start-up time and memory consumption? Since I didn't find any real up-to-date comparisons, I took a look for myself and ended up comparing these implementations:

- runc - The default OCI engine written in go that comes with containerd
- crun - Alternative OCI implantation written in C, claiming to be -49.4% faster than runc for running `/usr/bin/true`
- gvisor/runsc - Googles safe OCI implantation (also written in go) that aims to improve security by running many syscalls in userspace
- youki - Quiet young alternative OCI implantation written in rust

Testsetup

Disclaimer: This is NOT a scientific benchmark. I'm quite a noop in profiling such applications, and I cannot claim full correctness for my findings. If you know how to do it better, please show me.

I created a VM on the Hetzner-Cloud using their dedicated offerings to avoid the effect of noisy neighbors. I redid the test on another instance to ensure I didn't get a low-performing one. System Data:

```yaml
region: "nbg1-dc3"
os: "Linux 6.1.0-21-amd64 Debian 12 (2024-05-03) x86_64 GNU/Linux"
sku:
    type: CCX33
    cpu: 8
    ram: 32
```

The test environment was configured with an ansible playbook and my personal Kubernetes role I use to play around. My benchmark script uses the `time` tool to measure execution time and calculates min, max and mean for 1000 container creations for the busybox image running `/usr/bin/true`. I use `nerdctl` to create containers rather than calling the OCI implementation directly, since it mimics a more realistic use of an CRI. Before every test, I run a small warm-up to allow the CPU to boost clock speed. I found it hard to measure CPU and memory consumption for a short-lived process, so I installed node-exporter and have it scraped by Prometheus (running on another system) every 5 seconds.

```
containerd version 1.7.18
crun version 1.15
runc version 1.1.13
runsc version release-20240617.0
youki version 0.3.3
```

Results

The node's CPU and RAM were not even remotely utilized as seen in the Grafana dashboard below. In contrast to the production environment where the kubelet can create several containers simultaneously, my benchmark runs in serial. Nevertheless, some conclusions can be drawn. The test showed indeed that crun outperforms containerd's default OCI engine, yet the difference is not as big as claimed on the crun website. However, you have to bear in mind that it was not the same test as on the crun page. The performed benchmark includes the overhead of containerd, to represent a more realistic deployment compared to calling the OCI engine directly.

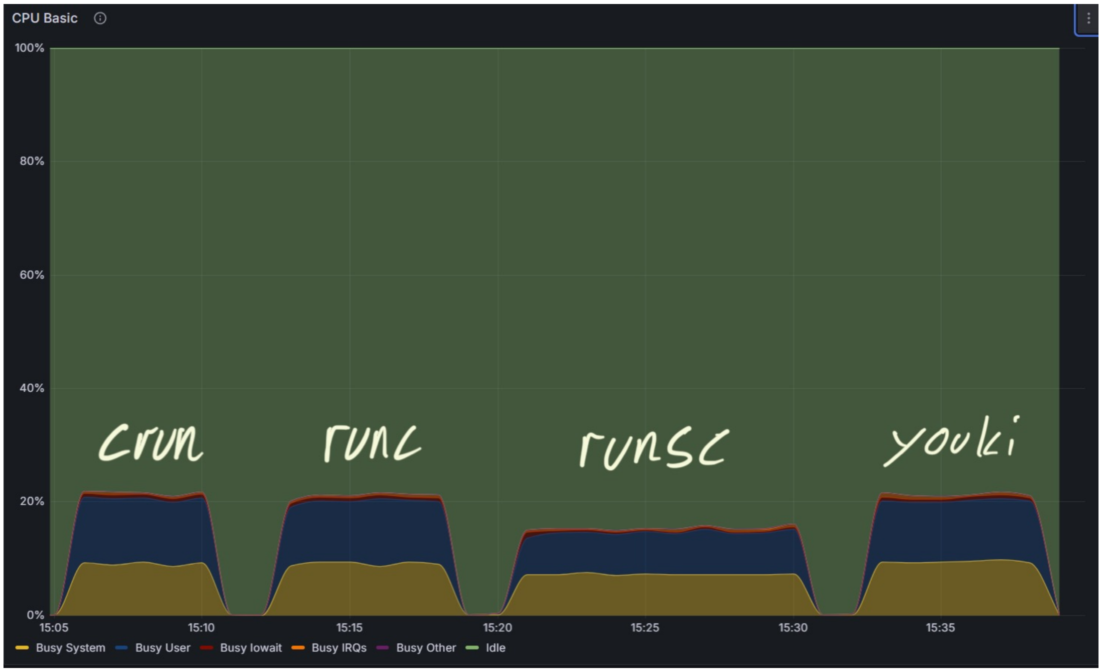


Figure 2: Grafana Dashboard showing OCI CPU usage

Given the claims from crun I was expecting a bigger difference between crun and runc. A 21% difference is still impressive. Surprisingly, the quiet new youki implementation is right up there head to head with runc when it comes to pure performance. Unfortunately, youki didn't perform as consistent and errored out quite some times. I reported the error to the authors and hope it will be fixed soon, despite its good performance an error rate of 3.6% is not acceptable in a

production environment.
As expected, runsc performs worst by far. However, performance is not its primary goal, it aims to improve security by implementing syscalls in userspace. The chosen test may also not favorable for runsc since running a short-lived container with `true` does not have many syscalls, the lower context switche are still noticeable as shown in the last dashboard below.
The main findings are summarized in the table below.

OCI	Mean	Min	Max	Errors	Total Time	Min Mem
crun	0.28s	0.13s	0.41s	0	306s	160kb
runc	0.34s	0.16s	0.44s	1	362s	8mb
runsc	0.62s	0.44s	1.64s	0	642s	32mb
youki	0.32s	0.16s	0.46s	36	354s	2mb

Memory was hard to test, and I found no significant differences here. From the data node-exporter provided it seems like crun and runc use a little less memory compared to the others, but I can not prove that no other process caused the increase, even when I did my best to keep variations low.

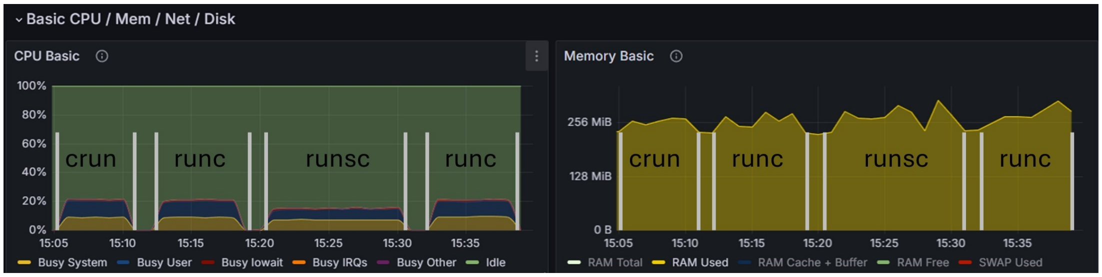


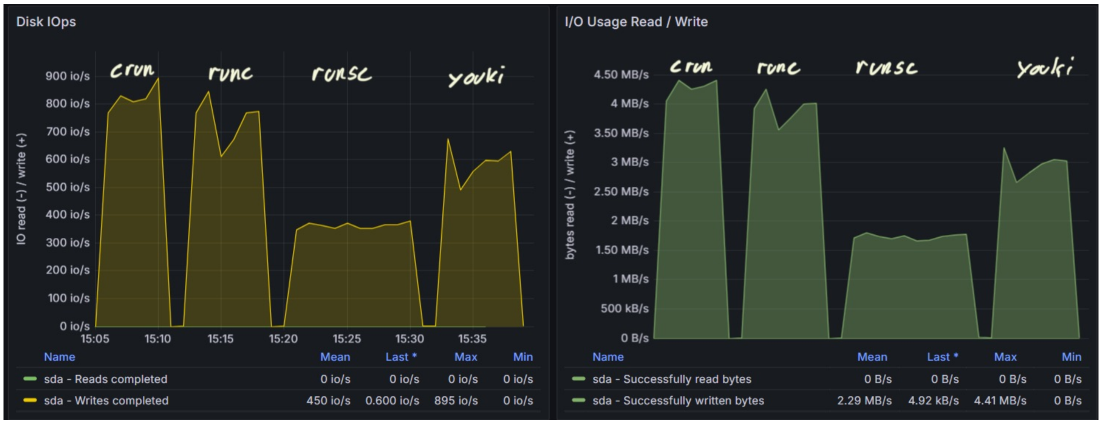
Figure 3: Grafana Dashboard showing OCI CPU & RAM usage

A claim that I was able to verify is that cruns authors say it requires far less memory to start a container than runc. I tested how low I could set the memory limit via the CRI for each engine via `nerdctl run --memory 4mb ....` The authors claim it requires less than 4mb, which is the lowest limit podman allows. I was able to start busybox with a memory limit of just 160kb. This is indeed impressive, even though it is not directly relevant in a production environment, since most applications within a container will need and have much higher limits.

Honorable findings

I wanted to compare the runtime and memory performance of these OCI implementations, but when monitoring, I also noticed some spikes in IO.

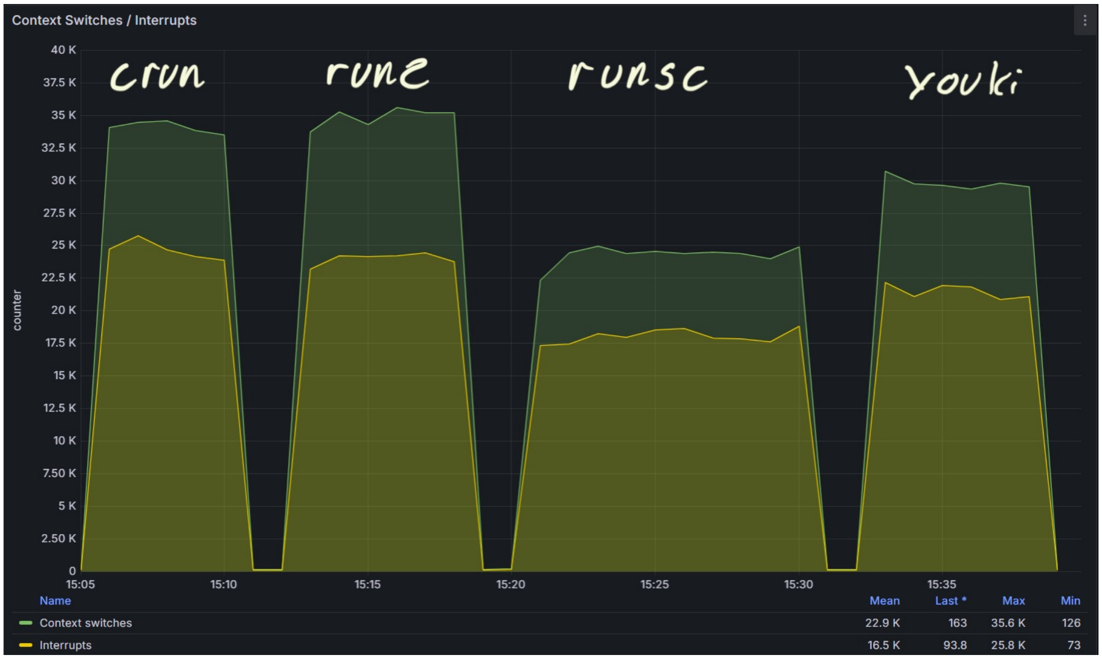
I did not expect that starting a small container like busybox would cause some significant IO writes, and I don't really know why write activity is "so high" for just starting a container. Both crun and runc caused around 700-800 IOPS and 4-4.5 MB/s in writes. Slightly less was needed for youki and runsc was using just half of IOPS and MB/s in comparison with crun and runc. According to its promises, runsc also caused fewer context switches for syscalls. I'm quite interested in learning more about the "high" writes, though. Especially where the differences come from when the same snaphotter is used across all OCI engines.



Name	Mean	Last *	Max	Min
sda - Reads completed	0 io/s	0 io/s	0 io/s	0 io/s
sda - Writes completed	450 io/s	0.600 io/s	895 io/s	0 io/s

Name	Mean	Last *	Max	Min
sda - Successfully read bytes	0 B/s	0 B/s	0 B/s	0 B/s
sda - Successfully written bytes	2.29 MB/s	4.92 kB/s	4.41 MB/s	0 B/s

Figure 4: Grafana Dashboard showing OCI IO usage



Name	Mean	Last *	Max	Min
Context switches	22.9 K	163	35.6 K	126
Interrupts	16.5 K	93.8	25.8 K	73

Figure 5: Grafana Dashboard showing OCI system context switches

The complete raw output can be found on my GitHub Gist.

Conclusion

The performance differences between the tested OCI implementations are not as big as I expected, and even though crun performs better, I doubt that switching to it justifies the operational overhead for most companies - at least for managed Kubernetes offerings. As far as I know most managed Kubernetes offerings use containerd with runc. Being able to just use AWS managed AMIs (or Azures) is a huge time and cost saver. Unless the last percent of performance is highly important, I would just stick with runc. It would still be nice to see hyperscaler's include crun in their AMIs (which costs 2MB of space) and give users the choice via runtimeclasses.
For self-hosted and embedded Kubernetes offerings, crun might be worth a look. Even more if companies are running cutting edge technology like WASM within their clusters. Both crun and youki can be compiled with WASM support. Meaning users can run WASM applications in their clusters with the same OCI as all their other containers. Which is SICK! I've been using this for quite some time and it works great. The only downside is that WASM support is not enabled by default yet and you have to compile it yourself. (I've set up an GH Action automation just for that.) The first look of youki is promising, but without having these errors solved, I wouldn't use it in production. When it comes to gvisor/runsc I'm a little unsure. It offers additional security, but most people run their own or trusted workloads in their clusters. If one would run random untrusted code, like someone's leetcode or lambda code I might want to use things like kata containers or firecracker VMs which offer a higher level of isolation. If someone wants to include these in the tests, I'm very interested - I couldn't run these since they require nested virtualization, which is not supported on Hetzners CCX VMs. This leaves gvisor in some weird middle ground, especially with Kubernetes upcoming ability to run containers in user namespaces further increasing security.

Innovation in the container world has settled down a bit since a lot of the technology as reached mainstream adoption and has standardized. The existing solutions have become more accessible, and the defaults cover the needs of the majority of use cases. Yet, it is still interesting how new solutions hit the market, resulting in a drop in replacements for solving specific problems.