Engineer

Container-Runtime

die Software, die Container startet und verwaltet

Was es ist

Eine Container-Runtime ist die Software zwischen einem Containerimage auf der Festplatte und einem durch den Kernel isolierten Prozess. Der Ausdruck umfasst zwei Schichten, und oft meinen verschiedene Personen damit unterschiedliche: die Low-Level-Runtime, auch OCI-Runtime genannt, und die High-Level-Runtime, die häufiger Containerengine heißt.

Die Low-Level-Runtime hat eine eng begrenzte Aufgabe. Sie liest ein Bundle, das die Runtimespezifikation der Open Container Initiative als Verzeichnis mit einer `config.json` -Datei und dem Wurzeldateisystem des Containers definiert. Dann fordert sie beim Kernel die in der Konfiguration beschriebenen Namespaces, cgroups und Mounts an und startet darin den Prozess. Die Spezifikation nennt die Operationen, die jede Runtime unterstützen muss, nämlich state, create, start, kill und delete, überlässt die Kommandozeile aber der jeweiligen Implementierung. Version 1.0 erschien im Juli 2017. runc wird von der OCI selbst gepflegt und bezeichnet sich als „Low-Level-Werkzeug, das nicht für Endanwender entworfen wurde“. crun erledigt dieselbe Aufgabe in C mit geringerem Speicherbedarf und ist die in Podmans Dokumentation genannte Voreinstellung. youki ist eine dritte, in Rust geschriebene Variante.

Die Engine liegt darüber. containerd beschreibt sich als Daemon, der den gesamten Containerlebenszyklus seines Hosts verwaltet: Übertragung und Speicherung von Images, Ausführung und Überwachung von Containern sowie die zugehörigen Speicher- und Netzwerkanbindungen. CRI-O erfüllt eine ähnliche Aufgabe, ist aber ausschließlich für Kubernetes gebaut. Eine Engine lädt das Image herunter, entpackt es, schreibt das Bundle, übergibt es der Low-Level-Runtime und überwacht den Prozess weiter, nachdem diese Runtime sich beendet hat. Docker Engine liegt noch eine Stufe höher: Laut Dokumentation nutzt sie containerd für den Containerlebenszyklus, und containerd nutzt standardmäßig runc. Podman geht einen anderen Weg und betreibt keinen

dauerhaft laufenden Daemon. Jeder Befehl bereitet selbst das Bundle vor, ruft crun oder runc auf und überlässt den laufenden Container einem kleinen Überwachungsprozess namens conmon.

Kubernetes ergänzt einen weiteren Begriff. Das kubelet auf jedem Knoten spricht mit der Engine über das Container Runtime Interface, kurz CRI, ein gRPC-Protokoll, das sowohl containerd als auch CRI-O implementieren. Docker Engine ist älter als CRI. Deshalb brachte Kubernetes jahrelang einen Adapter namens dockershim mit und entfernte ihn in v1.24.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist eine Container-Runtime?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. So verpackt, dass sie auf dem Laptop, dem Testserver und im Produktivbetrieb gleich funktioniert und bei höherer Auslastung mitwächst.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 20 Docker-Apps auf Google Cloud und AWS betrieben — 20 bereitgestellte und überwachte Containeranwendungen

20 Docker-Apps auf Google Cloud und AWS betrieben Kontakt aufnehmen →

Warum es wichtig ist

Eine Container-Runtime auszuwählen bedeutet eigentlich, eine Schicht für die Änderung auszuwählen. Die beiden Schichten bringen unterschiedliche Vorteile.

Die Engine entscheidet über den täglichen Containerbetrieb: Wer lädt und speichert Images, wer startet einen abgestürzten Container neu, wohin gehen seine Logs, welche API sprechen andere Werkzeuge und wer verwaltet den Prozessbaum? Docker Engine auf einem Server durch Podman oder auf einem Kubernetes-Knoten durch containerd zu ersetzen ändert all das. Die Container selbst bleiben dabei unverändert, weil darunter dieselben OCI-Standards für Imageformat und Bundle liegen. Die Kubernetes-Dokumentation zur Entfernung von dockershim sagt es ausdrücklich: Mit `docker build` erzeugte Images funktionieren mit allen CRI-Implementierungen.

Die Low-Level-Runtime entscheidet, wie stark der Prozess isoliert ist. runc und crun starten einen gewöhnlichen Prozess unter dem Kernel des Hosts. Das ist schnell und vollständig kompatibel, bedeutet aber, dass sich alle Container auf der Maschine diesen einen Kernel teilen. Die Runtime von gVisor, runsc, schiebt

einen in Go geschriebenen Anwendungskernel zwischen Container und Host und beantwortet die Systemaufrufe des Containers selbst. Als Preis nennt die eigene Dokumentation eingeschränkte Anwendungskompatibilität und höheren Aufwand pro Systemaufruf. Kata Containers startet jede Workload in einer schlanken virtuellen Maschine mit eigenem Kernel, nutzt Hardwarevirtualisierung als zweite Schutzschicht und braucht auf jedem ausführenden Knoten einen Hypervisor.

Kubernetes erlaubt, diese zweite Entscheidung pro Pod statt pro Cluster zu treffen. Eine RuntimeClass benennt einen in der Engine konfigurierten Handler. Ein Pod, der `runtimeClassName` setzt, läuft unter diesem Handler. Eine nicht vertrauenswürdige Workload kann so in gVisor oder Kata laufen, während alles andere auf runc bleibt. Die Dokumentation beschreibt das als Abwägung zwischen Leistung und Sicherheit.

Wie es in der Praxis aussieht

Im Alltag meint der Ausdruck zunehmend die Engine-Schicht. Die Arbeiten auf dieser Website verwenden ihn in diesem Sinn.

Zwanzig mit Docker paketierte Anwendungen, die über zwei Clouds und einige wenige Container-Runtimes verteilt waren, mussten sich einheitlich bereitstellen lassen und bei Fehlern schnell diagnostizierbar sein. Docker übernahm die Paketierung, die Fehlersuche lief über Podman und das Debugging der Orchestrierung über Kubernetes. Solche Arbeit über Engine-Grenzen hinweg baut auf der beschriebenen Schichtung auf: Die Engines unterscheiden sich, die darunterliegenden OCI-Standards für Images und Runtimes sind gemeinsam.

Auf einem neueren Host fiel die Entscheidung auf der Engine-Schicht. Die Containerebene läuft mit Podman und Quadlet anstelle von Docker, weil die standardmäßig vorgesehene Engine Containerports vor den von der Härtungsrolle installierten Firewallregeln freigibt und neben systemd eine zweite Prozessaufsicht hinzufügen würde. Geändert hat die Wahl, wer jeden Container startet und danach überwacht: Aus Containern wurden Dienste, die wie andere Dienste unter der Prozessaufsicht starten, der der Host bereits vertraute.

Dieselbe Überlegung gilt für einen Host mit 512 MB für die öffentlich erreichbaren Dienste des Unternehmens, auf dem der verfügbare Arbeitsspeicher bei jeder Abwägung den Ausschlag gab. Seine Containerebene

besteht aus Quadlet-Units unter systemd statt aus einem Daemon mit eigener Prozessaufsicht. Die Entscheidung fiel auf einer Maschine mit etwa 464 MB nutzbarem Arbeitsspeicher.

Wo es schiefgeht

Runtimeprobleme zeigen sich meist an den Übergängen zwischen Schichten, an denen jede Seite voraussetzt, dass die andere wie erwartet eingerichtet ist.

Knotenwerkzeuge verlieren nach einem Clusterupgrade den Überblick. Seit Kubernetes v1.24 spricht das kubelet direkt mit containerd oder CRI-O. Vom Cluster eingeplante Container laufen deshalb nie durch Docker Engine. `docker ps` auf dem Knoten listet nur auf, was Docker selbst gestartet hat. Monitoringagenten, Startskripte und privilegierte Pods, die die Docker-CLI aufrufen oder `docker.sock` einbinden, verlieren ihre Sicht. Die Workloads laufen weiter. Die Werkzeuge um sie herum müssen auf `crictl` oder die eigene API der Engine umgestellt werden.

Das kubelet und die Runtime verwalten cgroups unterschiedlich. Beide müssen denselben cgroup-Treiber verwenden, `cgroupfs` oder `systemd`. Das kubelet verwendet standardmäßig `cgroupfs`. Auf einem Host mit systemd als Init-System entstehen so zwei cgroup-Verwalter mit unterschiedlichen Ansichten derselben Ressourcen. Die Kubernetes-Dokumentation warnt, dass solche Knoten unter Ressourcenknappheit instabil werden können. Die Lösung ist der systemd-Treiber auf beiden Seiten, der auch für cgroup v2 empfohlen wird. Für containerd 2.x mit runc sieht die Runtimeseite so aus:

```
# /etc/containerd/config.toml, containerd 2.x
[plugins.'io.containerd.cri.v1.runtime'.containerd.runtimes.runc.options]
  SystemdCgroup = true
```

Auf der kubelet-Seite steht `cgroupDriver: systemd` in der Konfiguration. Anschließend muss containerd neu gestartet werden. containerd 1.x schreibt den Abschnitt anders; die Kubernetes-Seite zeigt beide Formen. Den Treiber auf einem bereits dem Cluster beigetretenen Knoten zu ändern ist heikel: Unter einem Treiber erstellte Pods können bei der Neuerstellung unter dem anderen auf Fehler stoßen. Die dokumentierte Empfehlung lautet deshalb, den Knoten mit der neuen Konfiguration zu ersetzen oder neu zu installieren.

Ein auf eine Sandbox-Runtime umgestellter Pod scheitert beim Start. Fehlt der Engine des Knotens der Handler einer RuntimeClass, geht der Pod sofort in die Phase Failed. Der Grund steht in einem Ereignis. Darüber hinaus muss die Workload zur Sandbox passen. gVisor beantwortet Systemaufrufe in seinem eigenen Kernel. Software, die weniger verbreitete Kernelfunktionen nutzt, kann sich deshalb anders verhalten. Die Dokumentation bezeichnet Workloads mit vielen Systemaufrufen als schlecht geeignet. Kata braucht Hardwarevirtualisierung auf dem Knoten, bei einer virtuellen Maschine in der Cloud also verschachtelte Virtualisierung. Ein `scheduling` -Knotenselektor an der RuntimeClass beschränkt solche Pods auf Knoten, die sie ausführen können.

Ein von Hand importiertes Image bleibt für das kubelet unsichtbar. containerd verwaltet Imagenamen und Metadaten pro Namespace. Sein Client `ctr` arbeitet ohne andere Vorgabe im Namespace `default` , das vom kubelet angesprochene CRI-Plugin dagegen in `k8s.io` . Ein mit einem schlichten `ctr images import` geladenes Image landet deshalb dort, wo das kubelet nie nachsieht. Der Pod greift erneut auf die Registry zu oder bleibt mit einem Fehler beim Imageladen stehen. Ein Import mit `ctr -n k8s.io images import` und die Kontrolle mit `crictl images` , das dieselbe Sicht wie das kubelet hat, vermeiden diese Lücke.

Weiterführende Lektüre

- OCI-Runtimespezifikation — der Standard, den jede Low-Level-Runtime implementiert: Bundle, `config.json` und Lebenszyklusoperationen.
- runc — die eigene Runtime der OCI mit Hinweisen dazu, wie übergeordnete Werkzeuge sie ansteuern.
- Container-Runtimes — cgroup-Treiber, CRI-Versionen und die von Kubernetes erwarteten Einstellungen für containerd und CRI-O.
- Container Runtime Interface (CRI) — das gRPC-Protokoll zwischen kubelet und Engine.
- RuntimeClass — die Wahl eines anderen Runtimehandlers pro Pod und die Beschränkung dieser Pods auf geeignete Knoten.
- Alternative Container-Runtimes — wie Docker Engine Container an gVisor, Kata Containers, youki oder Wasmtime übergibt.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Bereitstellungsweg, die Betriebsrisiken und die Bereitschaftsverantwortlichen. Wir können eine Änderung mit Prüfungen, einem Wiederherstellungsweg und klaren Zuständigkeiten abgrenzen.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist eine Container-Runtime?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/container-runtime/