



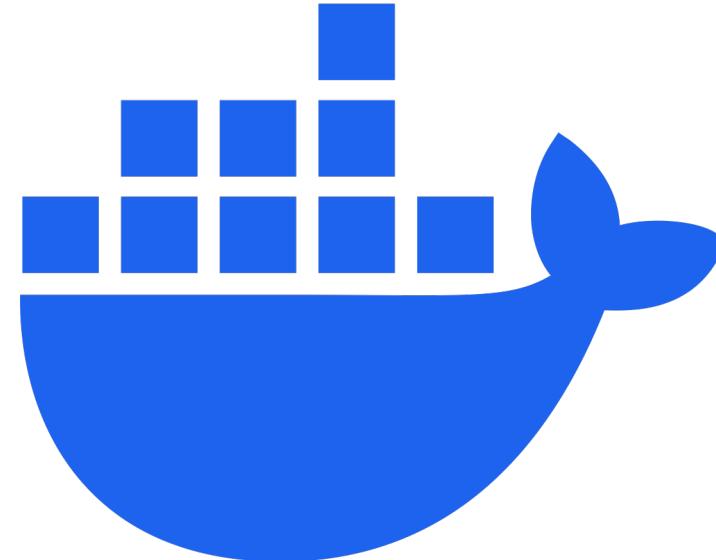
Универзитет “Св. Кирил и Методиј” во Скопје

Факултет за електротехника и информациски технологии



Основи на контенерирација

Вон. проф.
Даниел Денковски



Содржина

- | | |
|-----------|--|
| 01 | Зошто се потребни контејнери? |
| 02 | Архитектура и внатрешна структура на Докер |
| 03 | Градење и извршување на контејнери |
| 04 | Складирање, мрежа и повеќеконтејнерски системи |
| 05 | Современи предизвици и најдобри практики |

Предизвици при распоредување на софтвер (1/2)

- Софтерот често престанува да функционира правилно кога се преместува **од една во друга околина** поради:
 - различни верзии на зависности,
 - разлики во оперативните системи,
 - или промени во конфигурацијата низ времето.
- Наместо да се фокусираат на развој, програмерите често трошат време на **отстранување грешки** што:
 - не се поврзани со логиката на апликацијата,
 - туку произлегуваат од околината во која се извршува.

Предизвици при распоредување на софтвер (2/2)

- Традиционалните решенија, како **виртуелните машини**, овозможуваат изолација, но по цена на значителен трошок.
- Тие се бавни, тешки за стартивање и често користат **повеќе меморија и процесорски ресурси** отколку што е потребно.
- Ова што е потребно е метод што ќе дозволи лесно, брзо и доследно **пакување и стартивање на апликации** за различни околини на извршување.
 - Без разлика дали се работи за локална машина, сервер во компанија, платформа во облак или сл.

Контејнери како можно решение

Контејнерите се технологија што овозможува лесно **создавање, дистрибуција и извршување** на апликации во предвидливо и преносливо опкружување.



Со помош на контејнери, апликациите стануваат:

лесни за
распоредување

репродуцибилни

преносливи

конзистентни низ сите
фази на развој



Со контејнерите се овозможува „работеше кај мене“,
конечно да значи „**работи насекаде**“.

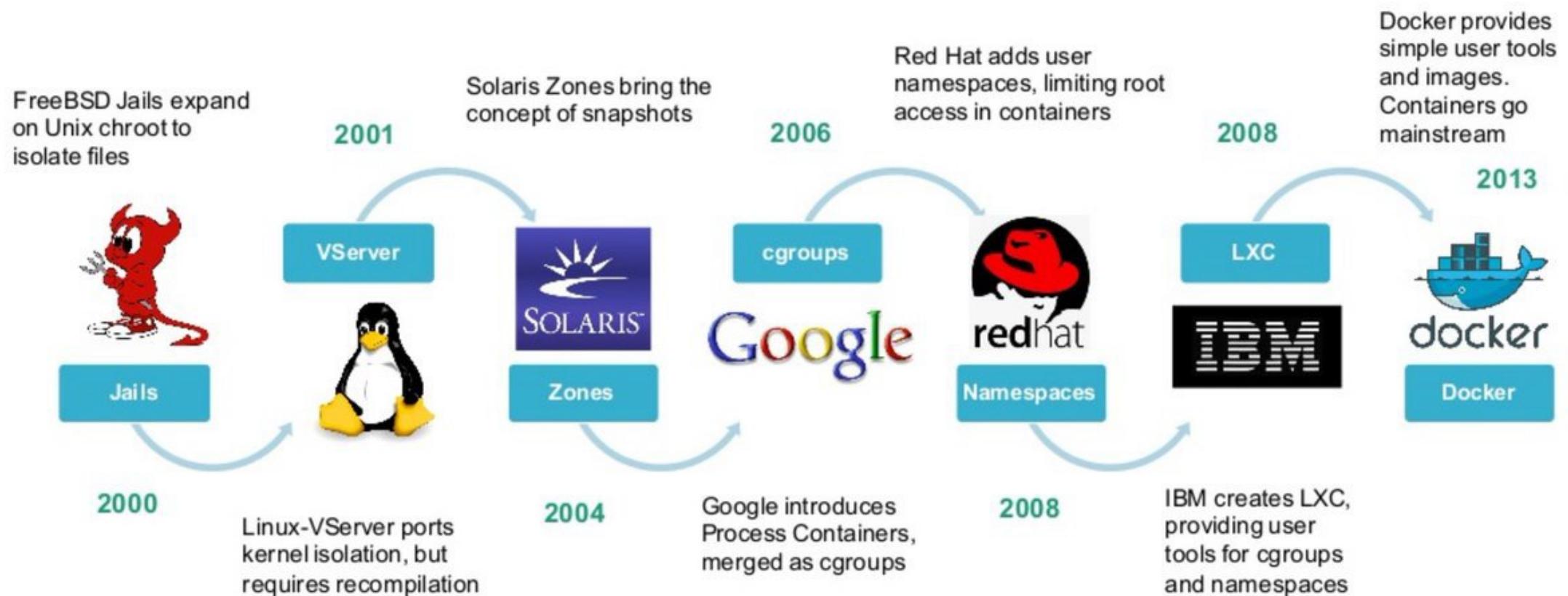
Историјат на контејнеризација (1/3)

- Потребата за изолација на апликации постои уште од почетоците на големите оперативни системи. Со текот на времето методите на изолација на процеси еволуирале до модерните контејнери.
- **Unix Chroot (1979)** е првиот чекор кој овозможил ограничување на процес во дел од податотечниот систем. Основната идеја е процесот да верува дека е единствениот во системот.
- **FreeBSD Jails (2000)** додава изолација на процеси и мрежа. Овозможува повеќе виртуелни инстанци да коегзистираат на истиот хардвер, секоја со сопствени IP адреси и безбедносни правила.
- **Solaris Zones (2005)** внесува концепти како snapshotting, контрола на ресурси и брзо креирање на нови изолирани околини. Тоа претставувал значаен чекор кон автоматизирана и скалабилна инфраструктура.

Историјат на контејнеризација (2/3)

- **Linux Containers LXC (2008)** овозможуваат прва вистинска виртуализација на ниво на оперативен систем, комбинирајќи:
 - Linux namespaces за изолација на процеси, мрежа и корисници, и
 - cgroups за контрола и ограничување на ресурси како CPU, RAM и I/O.
- **Докер** (2013, анг. Docker) не ја воведува изолацијата, туку ја стандардизира и прави подостапна, преку овозможување на:
 - Докер слики како лесни и повторливо употребливи шаблони,
 - Docker Hub како регистар за споделување на слики, и
 - Едноставен CLI и API, што го прави погоден за интеграција во CI/CD.
- Докер го отвора патот кон модерни **DevOps практики**, микросервисни архитектури и cloud-native екосистеми.

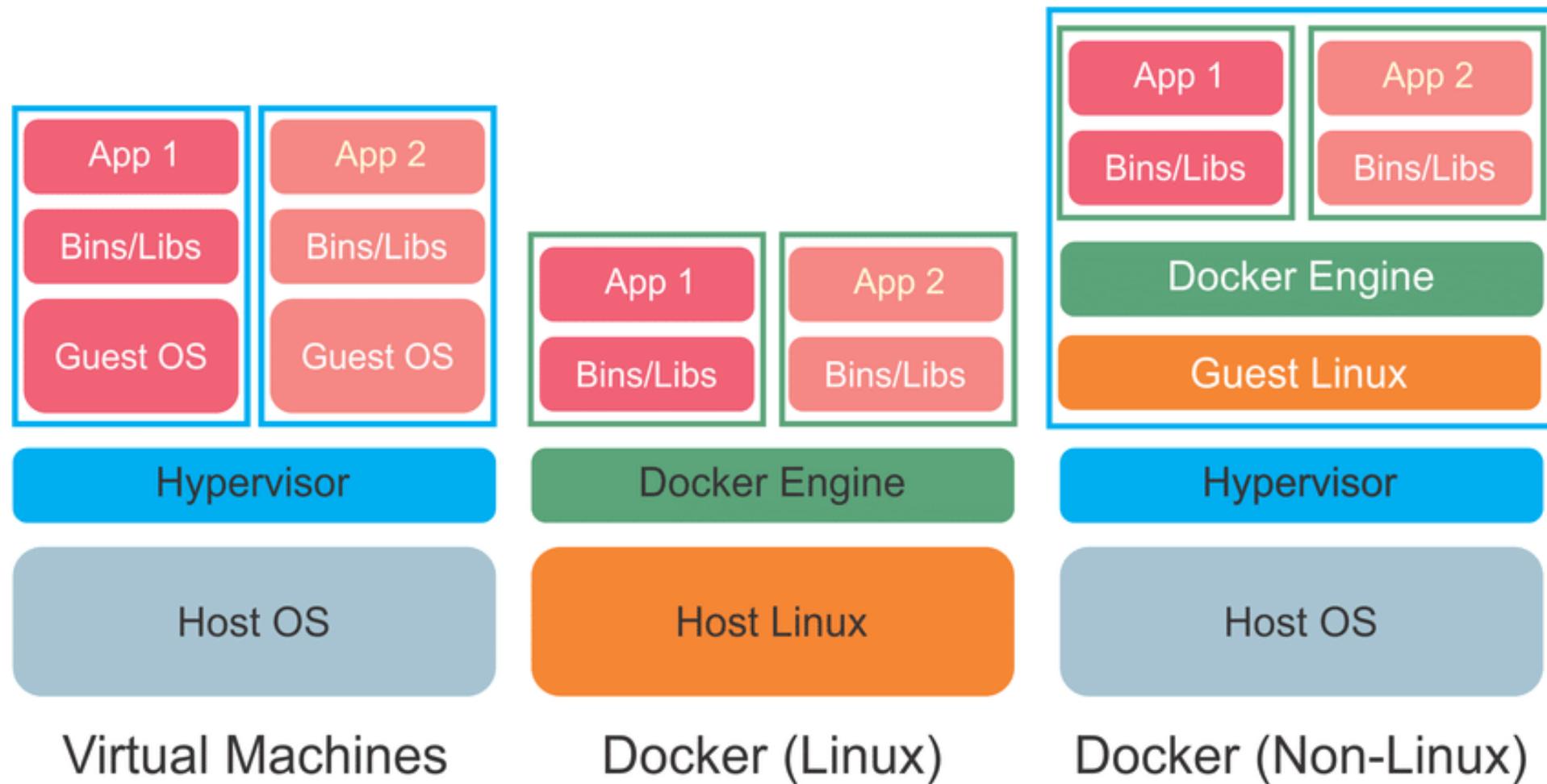
Историјат на контејнеризација (3/3)



Виртуелни машини наспроти контејнери (1/3)

- Во светот на модерната инфраструктура, и виртуелните машини и контејнерите имаат цел да обезбедат **изолација и предвидливост**.
- Меѓутоа, тие го постигнуваат тоа на различен начин, што директно влијае врз нивната брзина, ефикасност и наменска употреба.
- Виртуелните машини **емулираат физички хардвер** и секоја VM работи со целосен гостински оперативен систем (Guest OS).
- Контејнерите не емулираат хардвер и не извршуваат сопствен kernel. Тие го делат kernel-от со/од домаќинот, а изолацијата се постигнува на **ниво на процеси**.
- Виртуелните машини овозможуваат стабилност и посилна изолација, додека контејнерите овозможуваат брзина, ефикасност и скалабилност.

Виртуелни машини наспроти контејнери (2/3)



Виртуелни машини наспроти контејнери (3/3)

Аспект	Виртуелни машини (VMs)	Контејнери
Ниво на виртуализација	Виртуелизација на ниво на хардвер (секоја VM има свој OS).	Изолација на ниво на процеси (споделен kernel со домаќинот).
Оперативен систем	Секоја VM стартува целосен оперативен систем.	Контејнерите го користат истиот kernel од домаќинот.
Потрошувачка на ресурси	Голема, поголема потрошувачка на хардверски ресурси: CPU, RAM, диск.	Многу лесни, минимален товар.
Време на започнување	Бавно (секунди до минути).	Многу брзо (милисекунди до секунди).
Преносливост	Ограничена, зависи од хипервизор и OS конфигурации.	Многу висока, иста слика работи насекаде.
Големина	Големи (често во GB).	Мали (обично MB или неколку стотици MB).
Перформанси	Намалени поради целосна виртуелизација.	Речиси native перформанси.
Типична употреба	Извршување на повеќе различни OS околини, legacy системи.	Распоредување на апликации, микросервиси и cloud-native системи
Скалирање	Потешко и побавно.	Многу лесно и брзо (хоризонтално скалирање).
Безбедност	Подразбира подобра изолација.	Потребна е правилна конфигурација за силна изолација.

Клучни придобивки од контејнер системи

Конзистентна околина од развој до продукција: Контејнерите гарантираат дека апликацијата се однесува исто без разлика каде физички се извршува.

Вградено пакување на зависности: Апликацијата и сите зависни библиотеки и конфигурации се спакувани заедно. Нема повеќе конфликти меѓу системски верзии или недостасувачки компоненти.

Максимална ефикасност при извршување: Контејнерите користат минимални ресурси и стартираат речиси моментално, што овозможува голема густина на апликации на ист хардвер.

Флексибилни и брзо скалабилни: Системите изградени врз контејнери лесно се прилагодуваат на зголемен број корисници и оптоварување со додавање повеќе реплики или сервиси.

Подобрена стабилност и отпорност: Изолацијата на компонентите овозможува грешките да останат локализирани, што ја штити целокупната апликација и ја прави поотпорна на дефекти.

DevOps и контејнерите

- **DevOps** претставува технолошки пристап кој ги обединува тимовите за развој (Dev) и операции (Ops). Целта е побрза, поквалитетна и континуирана испорака на софтвер.
- Клучни принципи:
 - Автоматизација на процеси,
 - Континуирана интеграција и испорака (CI/CD),
 - Брза повратна информација,
 - Соработка и заедничка одговорност.
- Контејнерите природно ги поддржуваат DevOps процесите, бидејќи овозможуваат **преносливи и стабилни извршни околини**.

Улогата на контејнерите во DevOps

Конзистентни изградба (build): исти услови за извршување на секоја машина.

CI/CD автоматизација: лесно вклучување на алатки како Jenkins, GitLab CI/CD, GitHub Actions, Azure DevOps и др.

Брзо стартивање и лесно тестирање: секој тест се извршува во чист, изолиран контејнер.

Конзистентност меѓу Dev и Ops: нема повеќе проблеми со зависности или „кај мене работи“.

Инфраструктура како код и тимска ефикасност

- Со Докер, околината на апликацијата е дефинирана како код (Dockerfile, Docker Compose или K8s манифести).
- Наместо рачно подесување и споредување, инфраструктурата станува верзионирана, автоматизирана и пренослива.
- **Клучни придобивки:**
 - Конзистентни окolini низ развој → тестирање → продукција;
 - Лесно повторно креирање и скалирање на цела околина;
 - Минимален конфигурациски дрифт и помалку нејасни грешки;
 - Поефикасна соработка Dev ↔ Ops, помалку конфликт и губење време;
 - Побрзи испораки и повисок квалитет на системите.
- **Резултат:** Потранспарентен процес, постабилни распоредувања на софтвер и тим што работи како целина.

Примена на контејнери во различни домени

- **Веб апликации:** Контејнерите овозможуваат брзо и стабилно распоредување на веб сервиси со сите зависности.
- **Машинско учење:** Контејнерите гарантираат идентични услови за тренинг и ML инференција, овозможувајќи лесна дистрибуција на модели и повторливи резултати.
- **Пресметување на раб и IoT уреди:** Овозможуваат брз старт, минимален отпечаток и локална обработка на податоци кај уреди со ограничени ресурси.
- **Миграција на застарени апликации кон облак:** Контејнерите овозможуваат стари апликации да се преместат во облак без повторно кодирање и минимален ризик од премин кон нова инфраструктура.
- **SaaS платформи и multitenancy:** Контејнерите овозможуваат изолирана извршна околина за секој клиент, ефикасна распределба на ресурси и динамично скалирање.
- и многу други...

Практична примена во индустријата

See who uses Docker

Adobe AT&T Blue Apron INTEGRATE
Lucent Health NETFLIX paloalto PathFactory PayPal
pipedrive Segment snowflake spiceworks splunk
stripe TARGET The Container Store UNIVERSITY OF CALGARY Verizon
VIRGINIA TECH Yale

Companies that use kubernetes + docker

IBM amazon Microsoft Google
HUAWEI adidas Pinterest OpenAI Spotify

Платформи во облак и поддршка за контејнери

- Сите големи провајдери на сервиси во облак, како AWS, Google Cloud и Azure, нудат **првокласна поддршка за контејнери**, вклучувајќи:
 - Управувани контејнерски платформи: Amazon Elastic Container Service, Google Cloud Run и Azure Container Instances.
 - Kubernetes услуги: Amazon Elastic Kubernetes Service, Google Kubernetes Engine и Azure Kubernetes Service.
 - Регистри за слики: Amazon Elastic Container Registry, Google Container Registry и Azure Container Registry.
- Ова ја потврдува официјалната поддршка и стандарден статус на контејнерите во инфраструктурите во облак.

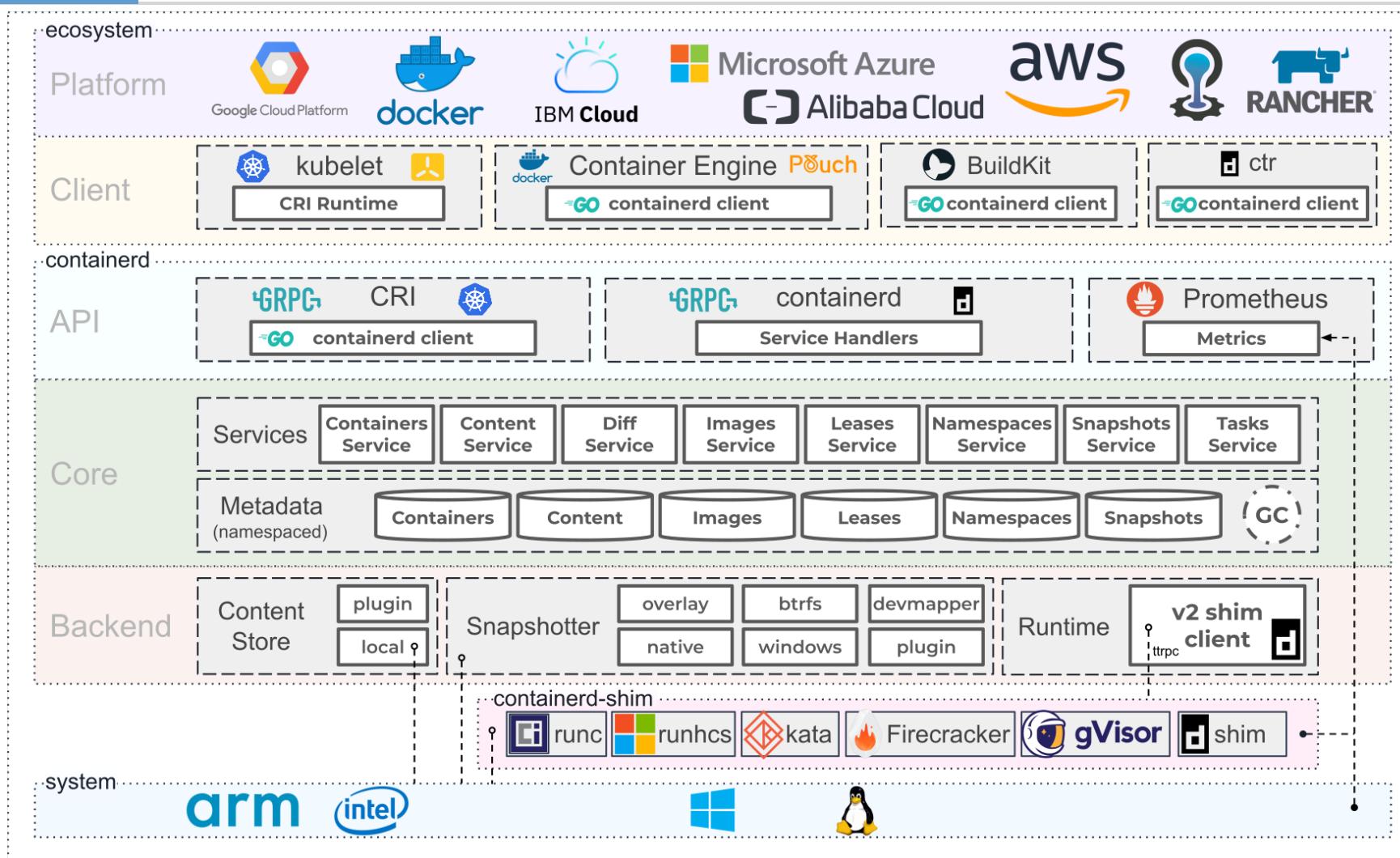
Екосистем на контејнери (1/2)

- Современиот контејнерски екосистем обединува алатки и стандарди за креирање, управување, дистрибуција и оркестрација на контејнери.
- Иако Докер е најпознат и најраспространет, тој претставува само еден елемент во поширокиот контејнерски екосистем.
- **Докер** нуди алатки за работа со слики и контејнери, Dockerfile за дефинирање на извршна околина, и јавни (Docker Hub) или приватни регистри за споделување на слики.
- **Podman** е позната алтернатива која работи без централен daemon, поддржува root-less извршување, е компатибilen со Докер слики и CLI, и често се користи во enterprise и Linux-first средини.

Екосистем на контејнери (2/2)

- **containerd** е индустриски стандарден runtime кој управува со животниот циклус на контејнери, вклучително вчитување, складирање на слики и мониторинг на процеси. Тој е „engine“ под Docker и Kubernetes.
- **Kubernetes** е систем за оркестрација кој распределува контејнери низ кластер од компјутери, овозможува автоматско скалирање, мрежна комуникација, откривање на сервиси и само-опоравување.
- **Open Container Initiative (OCI)** дефинира отворени стандарди за контејнерски слики и runtime, овозможувајќи интероперабилност меѓу Докер, Podman, containerd и други, со што се поттикнува флексибилност и долгорочна одржливост.
- и многу други...

Мал дел од екосистемот на контејнери



Заклучок: зошто контејнери?

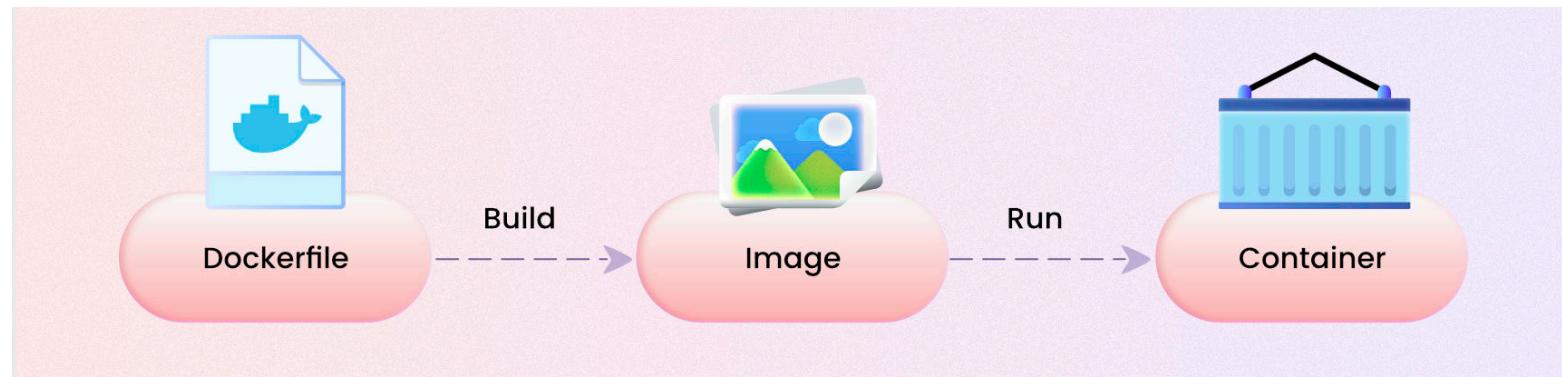
- Кonteјнерите овозможуваат апликациите да се извршуваат на ист начин при развој, тестирање и продукција, бидејќи ги носат со себе комплетно сите **конфигурации и зависности**.
- **Докер** ја приближи технологијата до поширока примена, овозможувајќи лесно креирање, споделување и извршување на контејнери.
- Наместо обемните виртуелни машини, контејнерите се лесни, стартиваат за **секунди** и ги користат ресурсите оптимално.
- Како резултат, се добива побрз развој, **автоматизирано** распоредување, лесно скалирање и стабилни системи во облак
→ основа на модерните DevOps практики.

Содржина

-
- 01** Зошто се потребни контејнери?
 - 02** Архитектура и внатрешна структура на Докер
 - 03** Градење и извршување на контејнери
 - 04** Складирање, мрежа и повеќеконтејнерски системи
 - 05** Современи предизвици и најдобри практики
-

Основни поими

- За да ја разбереме суштината на контејнеризацијата, потребно е јасно да ја разграничиме разликата меѓу **слика и контејнер**.
- Иако се користат заедно во практиката, тие претставуваат два одделни, но тесно поврзани концепти.
- Аналогија со објектно програмирање → слика е класа, контејнер е објект

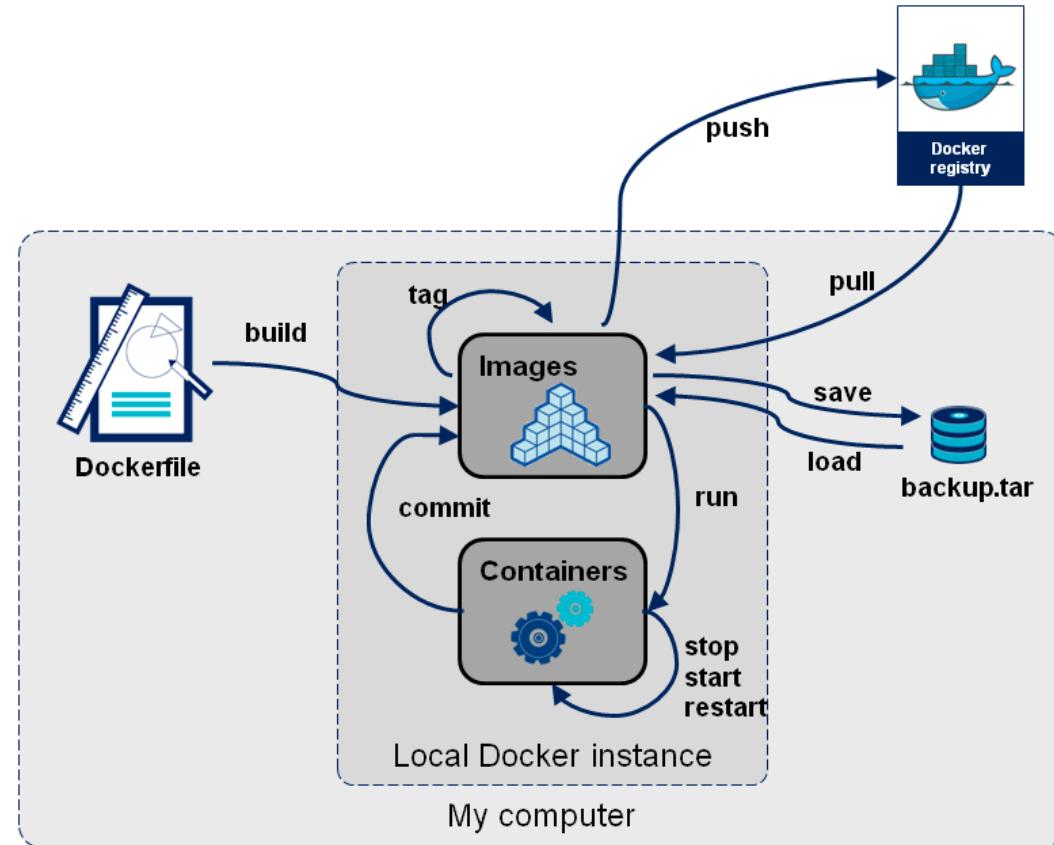


Докер слика

- Докер слика е статичен **read-only шаблон** кој ги содржи:
 - апликацијата,
 - нејзините пакети и зависности,
 - конфигурации и environment променливи,
 - основниот оперативен слој (на пример alpine, ubuntu или python).
- Сликите се **градат во слоеви**, а секој слој претставува конкретна измена (на пр. додавање на библиотека, копирање на код, конфигурација).
- Клучни карактеристики:
 - Не се извршуваат сами по себе, туку тие се опис на околината;
 - Се зачувуваат во регистри како Docker Hub, GCR, ACR итн.;
 - Се верзионираат преку тагови (пример: myapp:v1.2.0).

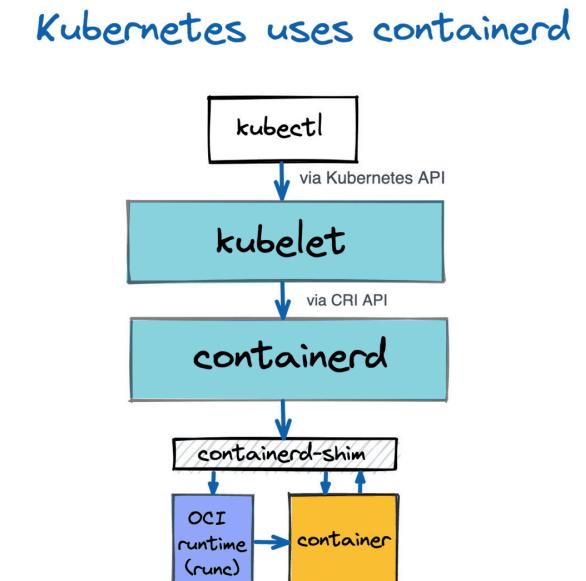
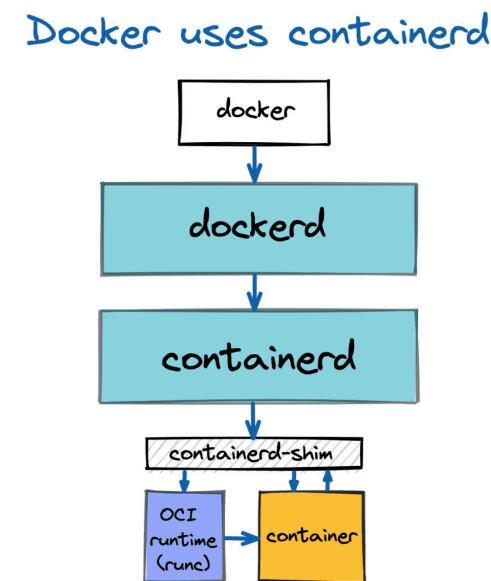
Контејнер (1/2)

- Контејнерот е активна или **живиа инстанца** создадена врз основа на некоја Докер слика, а тој содржи:
 - сопствен runtime процес,
 - запишувачки (writable) слој над сликата,
 - независна состојба, логови и мрежни правила.
- Може да стартуваме **еден или стотици контејнери** од иста слика, секој со своја работна состојба.

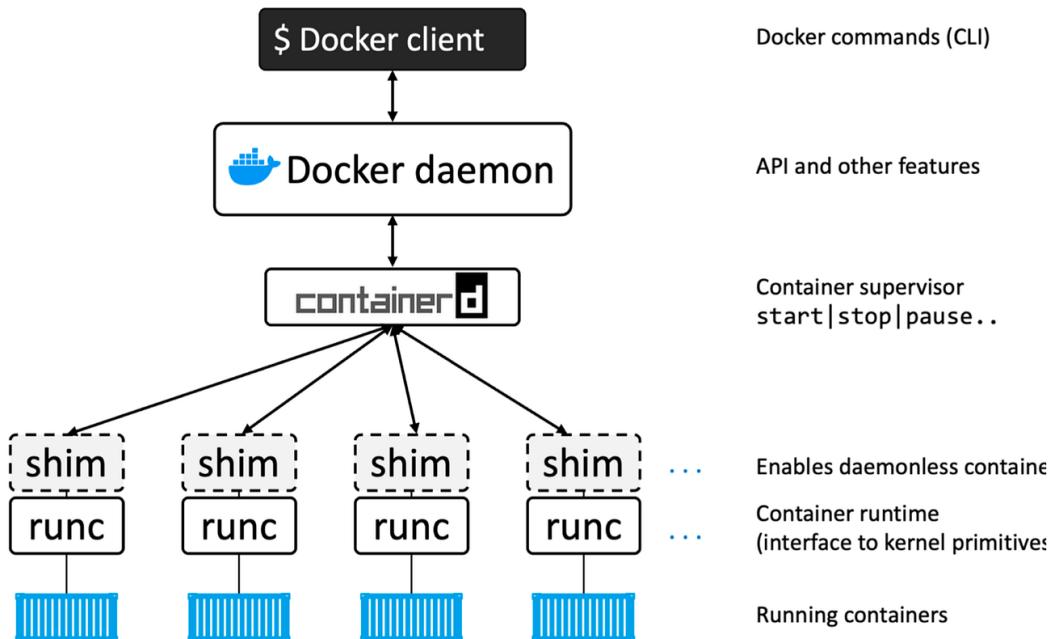


Контејнер (2/2)

- Врз основа на иста Docker слика, може да се инстанцира контејнер преку **различни системи**, како Docker, Podman, containerd или CRI-O.
- Клучни карактеристики:**
 - Лесен и брз за стартивање (секунди).
 - Изолиран од други контејнери и од домаќинот.
 - Привремен (ефемерен) по природа, што значи дека исчезнува по бришење, освен ако на него не е поврзано перзистентно складиште.



Докер архитектура (1/2)



Docker Daemon (dockerd) е заднински процес кој го контролира целиот животен циклус на контејнери. Управува со мрежи, складишта и интеракција со Докер регистри.

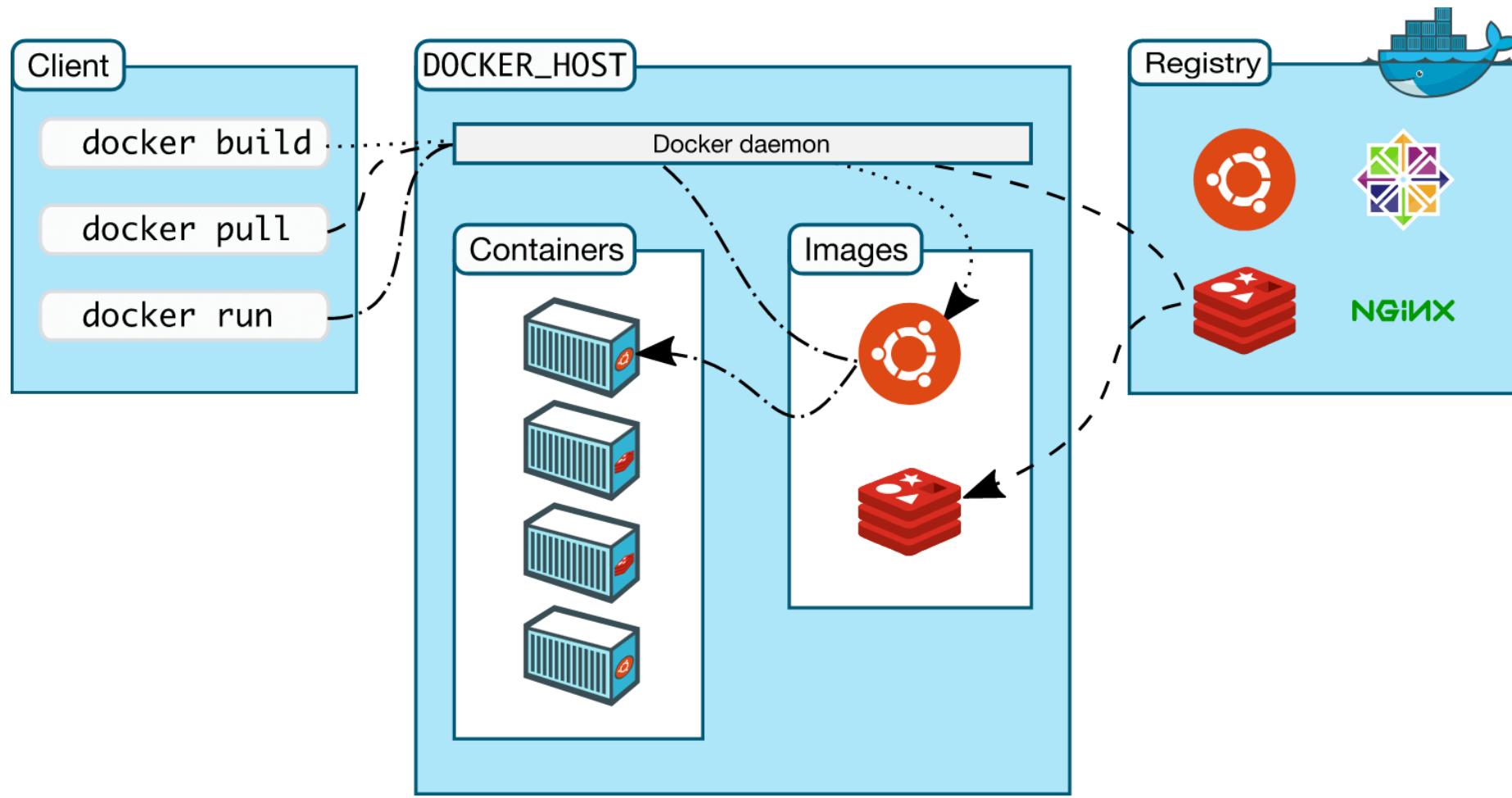
Docker CLI е командно-линиска алатка со која корисникот испраќа инструкции. CLI командите се претвораат во API повици кон Docker Daemon.

containerd и **runc** се внатрешни runtime компоненти. **containerd** управува со стартирање и стопирање на контејнери, а **runc** ја извршува изолацијата на ниско ниво користејќи Linux namespaces и cgroups.

Shim процесот го одржува континуитетот на контејнерот по неговото иницијално стартирање. Тој му овозможува на Docker daemon да може понатаму да го управува и мониторира контејнерот без прекин.

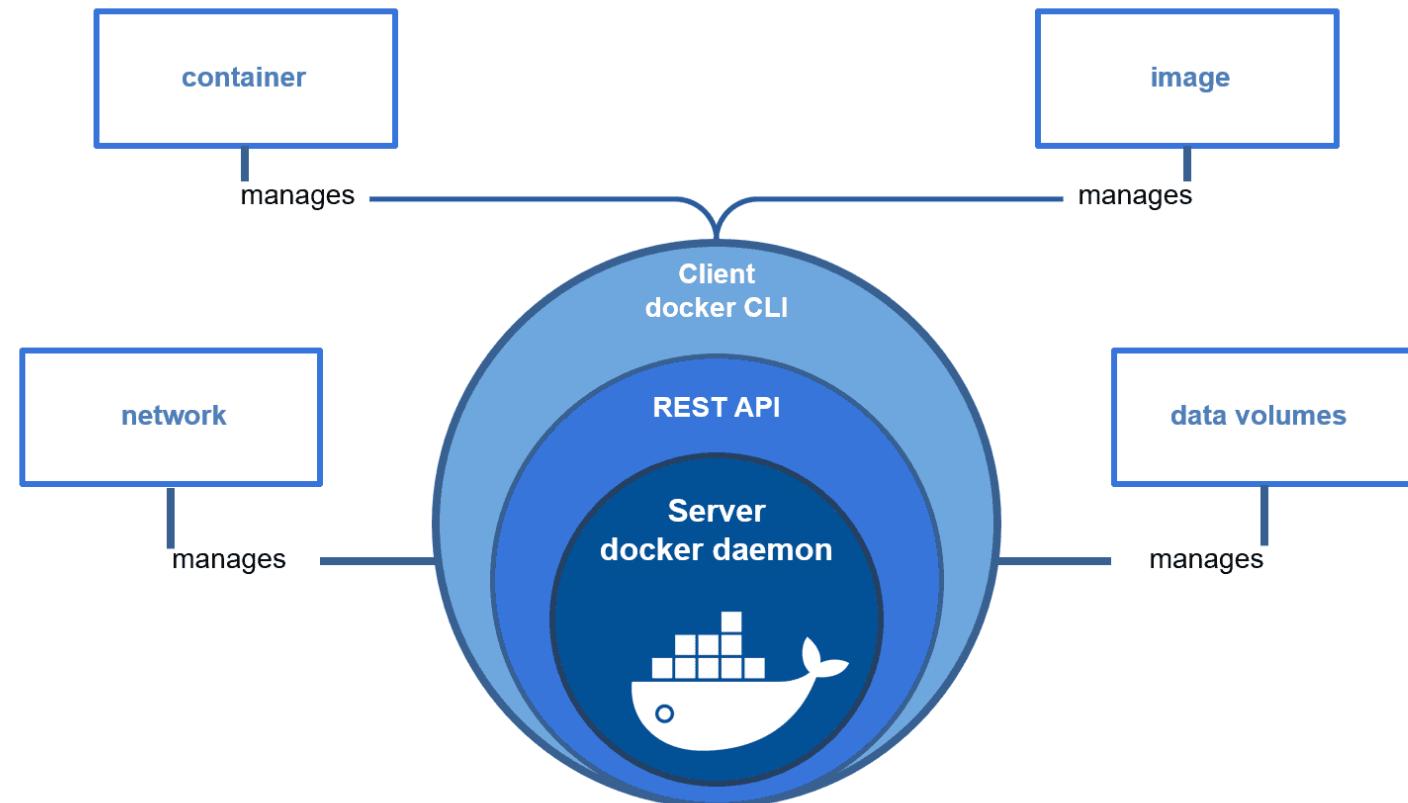
Регистри на слики (на пр. Docker Hub). Онлајн складишта за Докер слики, што овозможуваат складирање, верзионирање и размена на спакувани апликации меѓу тимови и системи.

Докер архитектура (2/2)



Docker daemon и CLI (1/2)

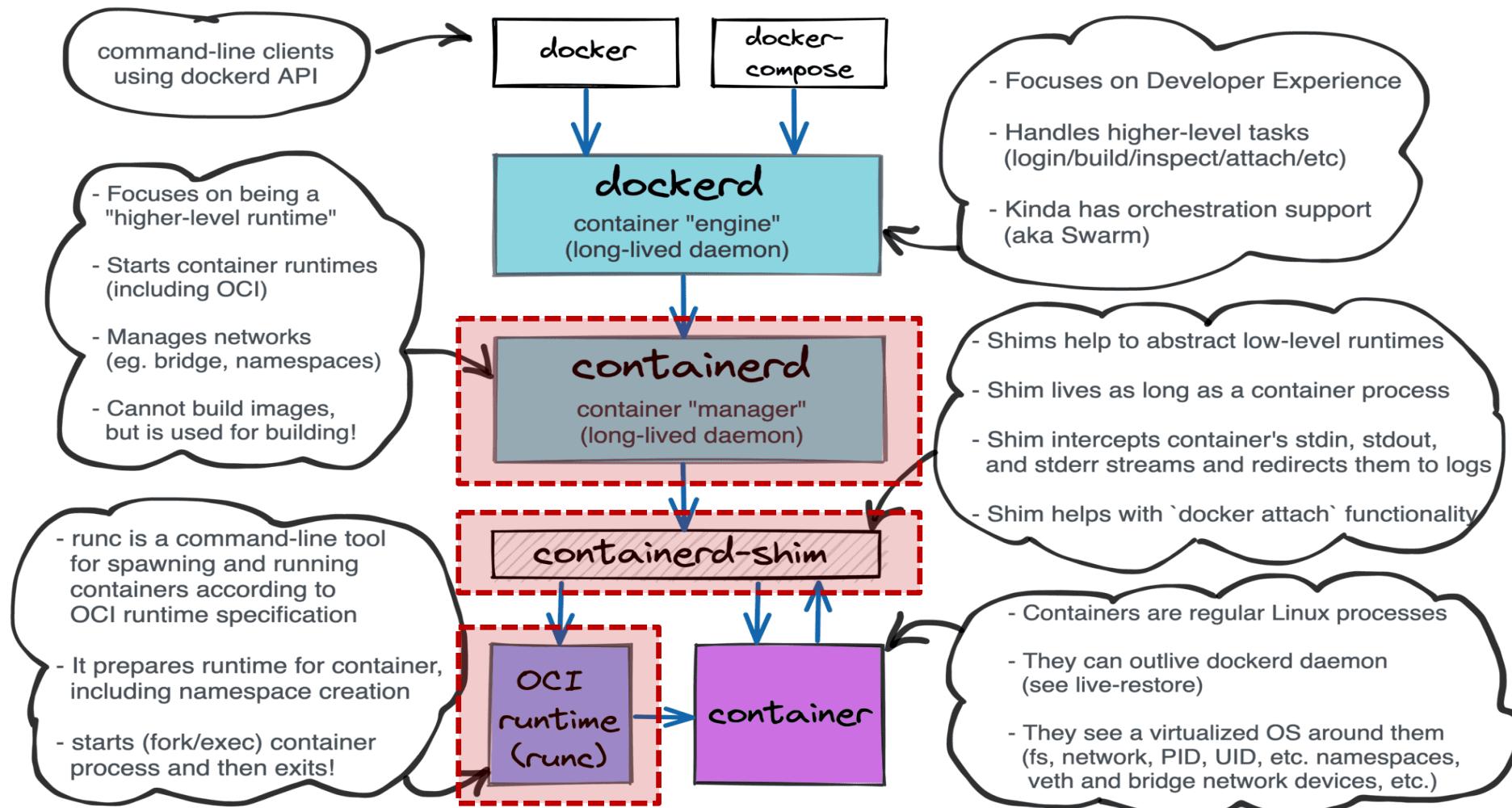
- Docker CLI и Daemon функционираат на принцип на **клиент-сервер комуникација**.
 - CLI = клиент; Daemon = сервер;
 - Можно е локално и далечно управување со Докер домаќини;
 - Флексибилност за развој, продукција и околини во облак.



Docker daemon и CLI (2/2)

- Dockerd е **централната точка за управување** во Докер, што работи како заднински процес. Тој управува со:
 - контејнери (create, start, stop, pause, remove),
 - слики (build, pull, push), и
 - регистри (push/pull).
- Docker CLI е **клиентска алатка** за командна линија преку која корисникот комуницира со Докер. CLI испраќа JSON API повици до Docker Daemon, како на пример:
 - docker build → гради Докер слика,
 - docker run → стартува контејнер,
 - docker ps → излиствува активни контејнери.

Containerd, runc и shim

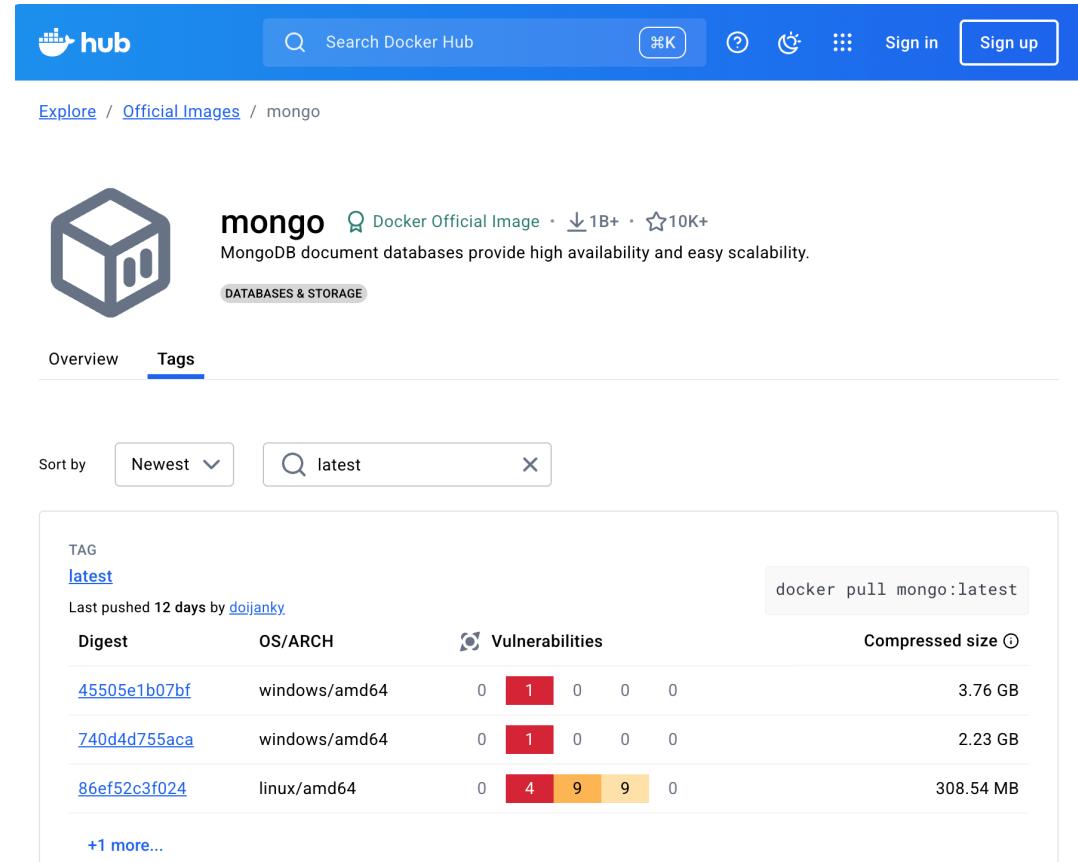


Докер регистри (1/2)

- Докер регистри се **централни складишта** каде се чуваат Докер слики, кои овозможуваат лесно споделување, повторна употреба и контролирана дистрибуција на апликации.
- Верзионирањето се прави преку **тагови**, што овозможува лесно управување со различни верзии на апликацијата.
- На пример, слика со таг myteam/myapp:v1.0.0 е конкретна верзија, а latest секогаш покажува најновата верзија.
- Најпознат јавен регистар е **Docker Hub**, кој овозможува секој да прикачува и презема слики. Сепак, за корпоративни или приватни проекти, се користат приватни регистри како **GitLab Container Registry**.

Докер регистри (2/2)

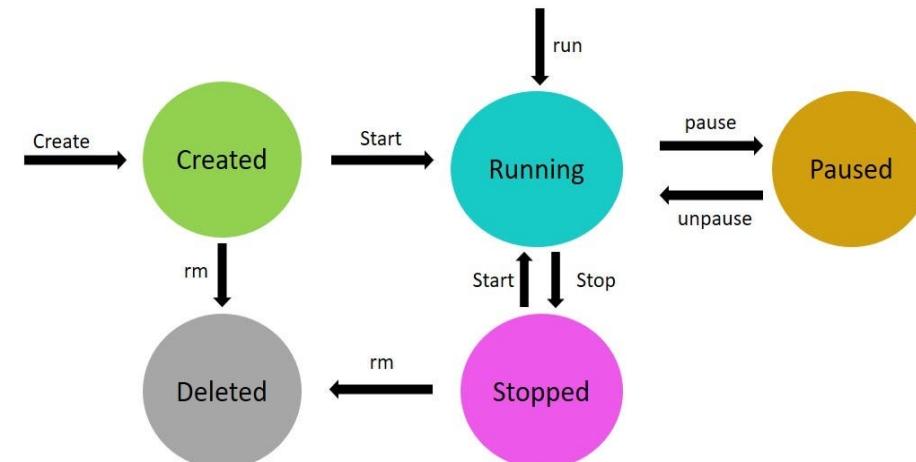
- **Работа со регистри:** Докер нуди едноставни команди за интеракција со регистри:
 - docker push → испраќа локална слика до регистар;
 - docker pull → презема слика од регистар за локална употреба.
- **Придобивки:**
 - повторливост и конзистентност: секој тим или машина користи иста верзија на слика;
 - колаборација помеѓу тимови: лесно споделување и повторна употреба.



Животен циклус на контејнер

- Контејнер поминува низ **неколку состојби** во својот животен циклус.
- Промени** во контејнерот се зачувуваат локално во него, а основната слика останува непроменета.
- По бришење, запишувачкиот слој се губи. Сепак, постои начин за перзистентност на податоци.
- Инспекција и интеракција:**
 - docker exec → команди внатре во контејнер;
 - docker logs → логови од контејнерот;
 - docker inspect → детална конфигурација;
 - docker stats → мониторинг на ресурси.

Настан	Резултат
Create	Креирање на контејнер од слика.
Start	Стартување на процесите.
Pause	Привремено паузирање без губење состојба.
Stop	Завршување на процесите, не состојбата.
Remove	Бришење на контејнерот и запишувачкиот (writable) слој.



Заклучок: поими и архитектура

- Кonteјнеризацијата ни овозможува да извршуваме апликации во лесни, изолирани и повторливи околини.
- **Докер слика** дефинира што ќе се извршува, тоа е статичен шаблон кој ги содржи апликацијата, зависностите и конфигурациите.
- **Контејнер** е активна инстанца на тој шаблон или жив процес со сопствена состојба и запишувачки слој.
- Во позадина, Докер користи **модуларна архитектура**:
 - Docker CLI за комуникација со корисникот,
 - Docker Daemon за управување со контејнери, слики, мрежи и складиште,
 - containerd и runc за изолација и извршување на пониско ниво,
 - Shim за стабилност и континуитет на процесите.
- Слики се складираат и дистрибуираат преку **регистри**, што овозможува верзионирање, колаборација и CI/CD автоматизација.

Содржина

- 01** Зошто се потребни контејнери?
- 02** Архитектура и внатрешна структура на Докер
- 03** Градење и извршување на контејнери
- 04** Складирање, мрежа и повеќеконтејнерски системи
- 05** Современи предизвици и најдобри практики

Dockerfile: план за контејнер

- Dockerfile претставува декларативна спецификација која опишува како се гради **Докер слика / контејнер**.
- Тоа е како „рецепт“ или „план“, кој дефинира од што е составена апликацијата и во каква околина ќе се извршува.
- **Клучни компоненти** на Dockerfile:
 - основна слика → од која средина почнуваме (пример python, ubuntu, alpine);
 - зависности → библиотеки и пакети потребни за извршување;
 - изворен код и конфигурации → што ја дефинира логиката на апликацијата;
 - почетна команда → што се случува кога контејнерот ќе се стартува.
- Секоја инструкција создава нов **слој (layer)** → систем кој овозможува кеширање, брзо изградба и ефикасно складирање.

Основни Dockerfile инструкции

- Dockerfile користи низа од инструкции за да ја изгради и конфигурира финалната слика.

Инструкција	Опис	Пример
FROM	Ја дефинира почетната околина.	FROM python:3.9-slim
RUN	Извршува команди за време на градење на сликата.	RUN pip install -r requirements.txt
COPY / ADD	Добава датотеки од домаќинот во Докер сликата.	COPY . /app
CMD	Default команда при старт.	CMD ["main.py"]
ENTRYPOINT	Програма која секогаш ќе се извршува.	ENTRYPOINT ["python"]

```
# CMD може да се промени при старт: docker run image new_cmd  
# ENTRYPOINT задава фиксно однесување на контејнерот. ENTRYPOINT е фиксно,  
а CMD само дава default аргументи и може да се препокрие.
```

Пример (Python апликација)

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

- **Што се случува:**

- Започнуваме од мала Linux + Python околина;
- Ја дефинираме работната папка /app;
- Прво ги внесуваме само зависностите → ова овозможува кеширање;
- Го копираме остатокот од кодот;
- Поставуваме how-to-run на апликацијата.

- **Краен резултат:** Докер слика доволна сама за себе.

Градење на Докер слика (build процес)

- Откако ќе се дефинира Dockerfile, следен чекор е **да се изгради сликата**.
- Оваа слика ја содржи целата околина и потребните компоненти за апликацијата да работи, така што подоцна можеме лесно да стартуваме контејнери од неа.
- Една од најмоќните карактеристики на Докер е **кеширањето на слоевите**. Доколку некоја инструкција не е изменета, се користи веќе изграденото ниво.

```
docker build -t myapp:latest .
```

-t myapp:latest, со -t и доделуваме име и таг (т.е. верзија) на сликата, што овозможува јасно разликување на верзии.

. (точката) ја означува локацијата на build контекстот, односно директориумот каде се наоѓа Dockerfile и изворниот код.

Стартување на контејнер (run процес)

- Откако е изградена Докер сликата, можеме да се **покрене контејнер**.
- Контејнерот ја вклучува апликацијата, околината и сите зависности, така што може да работи независно и изолирано од оперативниот систем на домаќинот.

```
docker run -it --name myapp-container myapp:latest  
# -i дозволува интеракција (input), а -t вклучува терминал (tty).  
Овозможува да влеземе „внатре“ во контејнерот и директно да комуницираме со апликацијата, што е особено корисно за тестирање.  
# --name myapp-container. Наместо Докер автоматски да генерира случајно име, ние му доделуваме читливо име за понатамошно управување (на пр.  
docker stop myapp-container)
```

Стартување на контејнер (продолжува)

- Контејнерот по дефиниција е **изолиран**. Ако сакаме да му пристапиме од надвор (на пр., веб сервер на порта 80), треба експлицитно да **мапираме**:

```
-p 8080:80
```

8080 е порта на домаќинот, а 80 е порта внатре во контејнерот.

Ова обезбедува пристап до апликации преку browser или API clients.

- Контејнерите сами по себе се **ефемерни** (привремени). За дачуваме трајни податоци (бази, логови, конфигурации), може да користиме **волуamenti**:

```
-v /host/data:/app/data
```

Ова овозможува: i) податоците да се зачувани и по бришење на контејнер, ii) споделени ресурси меѓу контејнери iii) раздвојување на апликација и податоци.

Detached mode (извршување во позадина)

- За апликации кои треба да работат континуирано, како веб сервери или бази на податоци, интерактивниот режим (-it) не е идеален, бидејќи терминалот останува зафатен.
- **Detached mode** (-d) овозможува контејнерот да работи во позадина, додека CLI пристапот веднаш се враќа за други команди.

```
docker run -d --name webserver nginx
# -d (detached). Стартува контејнер во позадина. Можеме веднаш да го користиме терминалот за други задачи. Контејнерот продолжува да работи, дури и ако го затвориме прозорецот на терминалот.
# --name webserver. Читливо и едноставно име за полесно управување.
# nginx. Името на сликата што се користи. Во овој пример, креираме Nginx веб сервер.
```

```
docker attach webserver # Можност за attach подоцна
docker logs webserver # Увид во stdout/stderr на апликацијата
docker stop webserver # Стопирање на контејнерот
docker start webserver # Повторно стартирање
```

Управување со животниот циклус (1/3)

- Докер овозможува целосно управување со животниот циклус на контејнерите, од развој до продукција и одржување на апликации.
- Познавањето на овие команди е клучно за прецизна контрола, како и ефикасно управување со ресурси.

```
# СТОПИРАЊЕ, СТАРТУВАЊЕ, РЕСТАРТИРАЊЕ
docker stop <container>
# Го застанува контејнерот на безбеден начин. Сите процеси во него се
завршуваат, но writable слојот останува сочуван за повторно стартирање.
docker start <container>
# Стартува претходно стопиран контејнер, враќајќи ја неговата состојба и
поврзаните ресурси.
docker restart <container>
# Комбинација од stop и start. Корисно за брзо обновување на контејнерот
по промени или конфигурациски ажурирања.
```

Управување со животниот циклус (2/3)

```
# ПАУЗА И ПРОДОЛЖУВАЊЕ
docker pause <container>
# Сите процеси се замрзнуваат, но writable слојот и ресурсите остануваат.
docker unpause <container>
# Продолжува извршувањето на паузираниот контејнер без губење на
состојбата.

# ИЗВРШУВАЊЕ НА КОМАНДИ ВО ЖИВ КОНТЕЈНЕР
docker exec -it <container> <command>
# Овозможува интерактивно извршување на команди внатре во контејнерот.

# Пример:
docker exec -it myapp-container bash
# Пристап до shell терминал, каде може да се направи тестирање на команди,
проверка на filesystem и процеси
```

Управување со животниот циклус (3/3)

```
# ИНСПЕКЦИЈА И МЕТАПОДАТОЦИ
docker inspect <container>
# Дава детални информации за контејнерот: мрежни конфигурации, волуеми и
слоеви, environment variables, статус, лимити на ресурси и сл.
```

```
# COMMIT НА КОНТЕЈНЕР
docker commit <container> <new_image_name>
# Создава нова Докер слика од тековната состојба на контејнерот.
# Снима промени направени во writable слојот. Овозможува лесно повторно
дистрибуирање и локален backup.
```

- Со овие команди, Докер дава **целосна контрола на животниот циклус**, од стартивање, пауза, стопирање, инспекција, до commit на нови верзии.
- Ова ја прави платформата флексибилна и сигурна за развој и продукција на модерни апликации.

Дебагирање на контејнерот

- Докер нуди моќни алатки за **логирање, интерактивен пристап и мониторинг на промени**. Преку овие алатки имаме можност да добиеме целосна слика за состојбата на контејнерот во реално време.

```
docker logs <container>
# Прикажува stdout и stderr излез на контејнерот.

docker exec -it <container> bash
# Отвора интерактивна shell сесија во тековен контејнер.

docker diff <container>
# Прикажува промени во filesystem на контејнерот споредено со оригиналната
# Докер слика.

docker top <container>
# Прикажува тековни процеси кои се извршуваат во контејнерот и физичките
# ресурси што ги користат (CPU, RAM).
```

Експортирање и споделување на слики

- Со експортирање, снимање и споделување, Докер нуди **флексибилност и контрола** за дистрибуција на апликации, без разлика дали се работи онлајн преку регистри или локално преку tar архиви.

```
docker export <container> > container.tar
# Ова ја зачува целата тековна состојба на filesystem во tar архив.
# Експортирањето е snapshot на filesystem, без метаподатоци или слоеви на
оригиналната слика.
```

```
docker save -o myimage.tar myapp:latest
# Снима цела Докер слика во tar архив.
# save зачува метадата, слоеви и конфигурација и од основната слика.
```

```
docker push <registry>/<image>
docker pull <registry>/<image>
# Споделување преку онлајн регистри.
```

Заклучок: градење и извршување на контејнери

- Dockerfile е **план за контејнер**, описувајќи ги основната слика, зависностите и иницијалните инструкции.
- Секоја наредба во Dockerfile создава **кеширан** слој, што овозможува ефикасно и повторливо градење.
- При стартирање со docker run, се инициира **активна инстанца** со сопствен процесен простор и слој за запишување
- Докер обезбедува алатки за управување, следење и дијагностика, овозможувајќи стабилност и предвидливост низ различни средини.
- Овој workflow обезбедува брз развој и лесна дистрибуција на апликации, правејќи го незаменлива алатка за **модерни DevOps** и cloud-native средини.

Содржина

01 Зошто се потребни контејнери?

02 Архитектура и внатрешна структура на Докер

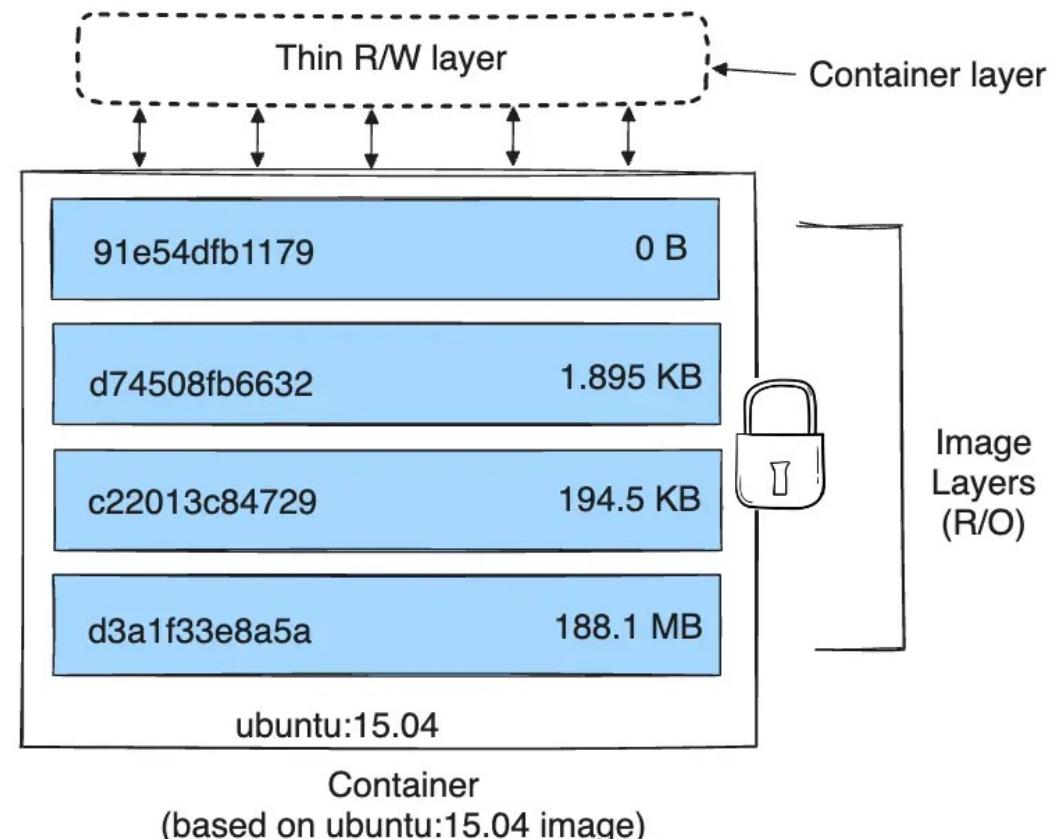
03 Градење и извршување на контејнери

04 Складирање, мрежа и повеќеконтејнерски системи

05 Современи предизвици и најдобри практики

Docker Storage (1/3)

- Секој Докер контејнер креира **запишувачки слој** на врвот на read-only слоевите од Докер сликата.
- Докер користи **Union податотечен систем** за логички да ги спои read-only слоевите од сликата со запишувачкиот слој на контејнерот, прикажувајќи ги како еден единствен датотечен систем.
- Физичка локација:**
 - Read-only слоеви:
`/var/lib/docker/overlay2/<image-id>/diff`
 - Запишувачки слој:
`/var/lib/docker/overlay2/<container-id>/diff`



Docker Storage (2/3)

- Кога контејнер работи, тој има свој **запишувачки слој** над основната Докер слика.
- Во него се чуваат сите промени за време на извршување, привремени фајлови и логови, но овој слој е **привремен** (ефемерен).
- Штом контејнерот се избрише, овие податоци **исчезнуваат**.
- Затоа, кога сакаме податоците да се **зачуваат** подолго или да се споделуваат меѓу повеќе контејнери, користиме:
 - Волуими (volumes): управувани од Докер, перзистентни, идеални за продукциски околини.
 - Врзани монтирања (bind mounts): директен пристап до директориуми од домаќинот, флексибилни, но помалку изолирани.

Docker Storage (3/3)

Тип на складиште	Опис	Перзистентност	Примена	Изолација / безбедност
Запишувачки слој (Writable layer)	Привремен слој во контејнерот каде се чуваат промени при извршување.	Не	Краткотрајни модификации, логови, кеширање.	Средна, изолирана во контејнер, но нестабилна.
Волуеми (Volumes)	Управувани од Докер, се чуваат во /var/lib/docker/volumes	Да	Продукција, stateful апликации како бази на податоци.	Висока, изолирани од домаќинот.
Врзани монтирања (Bind mounts)	Мапира директни директориуми од домаќин во контејнер.	Да	Развој, тестирање, директен пристап до локални фајлови.	Ниска, директен пристап до filesystem-от на домаќинот.
Tmpfs монтирања (tmpfs mounts)	Складирање во RAM меморија, без запис на диск.	Не	Привремени или чувствителни runtime податоци.	Висока, податоците се губат при рестарт.

Користење на волуеми во практика

- Волумените се клучен механизам во Докер за **перзистентност на податоци**, овозможувајќи им на апликациите да зачуваат состојба и податоци независно од животниот циклус на контејнерите.

```
docker volume create mydata
```

Креира перзистентен волумен со име mydata.

```
docker run -v mydata:/app/data myapp
```

Го врзува волуменот mydata во контејнерот на патека /app/data.

Податоците зачувани во /app/data ќе преживеат и рестарт и бришење на контејнерот.

```
docker volume inspect mydata
```

За увид во деталите на волуменот: локација во домаќинот, време на креирање, контејнери што го користат.

Врзани монтирања во практика

- Врзаните монтирања (bind mounts) овозможуваат **директно поврзување** на датотеки или директориуми од домаќин машината со контејнерот, што е корисно за развој и конфигурациски податоци.

```
docker run -v /path/on/host:/app/data myapp
```

Врзува директориум од домаќинот (/path/on/host) во контејнерот на патеката /app/data.

Податоците во /app/data се директно синхронизирани со домаќинот, па промени на едната страна веднаш се рефлектираат на другата.

Споредба на методите за перзистентни податоци

Придобивки од **волумени**:

- Одделување на податоците од контејнерот: контејнерите се привремени, податоците се перзистентни.
- Поддршка за апликации со состојба: бази, логови и прикачени датотеки се зачувуваат сигурно.
- Споделување помеѓу контејнери: ист волумен може да се поврзе во повеќе контејнери.

Придобивки од **врзани монтирања**:

- Споделување на датотеки: директориумот на домаќин може да се врзе во повеќе контејнери за споделување на податоци со домаќинот и меѓу контејнери.
- Лесен пристап до логови и конфигурации: може да се чита и пишува без rebuild на контејнерот.

Заклучок: врзани монтирања во фаза на развој, волумени во продукција...

Docker networking (1/2)

- Кога работиме со Докер, контејнерите по default се **изолирани** едни од други и од надворешната мрежа.
- За да овозможиме комуникација меѓу нив или да им дадеме пристап до и од надворешни сервиси, Докер користи сопствен **мрежен модел**.
- Овој модел овозможува различни нивоа на изолација, флексибилност и контролираност, во зависност од тоа како сакаме контејнерите да се поврзат.
- Docker networking е **критична компонента** при:
 - откривање на сервиси;
 - распределба на оптоварување;
 - апликации со повеќе контејнери.

Docker networking (2/2)

- Докер нуди **повеќе видови на мрежи**, секоја прилагодена за различни сценарија на работа и ниво на изолација.
 - bridge (default): Приватна мрежа на машината домаќин; овозможува сигурна комуникација помеѓу контејнери.
 - host: Контејнерот ја дели истата мрежа со домаќинот, што овозможува подобри перформанси, но помала изолација.
 - none: Контејнерот е целосно изолиран и нема мрежен интерфејс.
 - overlay: Гради виртуелна мрежа што поврзува контејнери на различни домаќини како да се наоѓаат на ист систем; се користи во Docker Swarm и Kubernetes.
 - macvlan / ipvlan: Им доделува реални МАС/IP адреси; корисно за интеграција со физички мрежи и напредни конфигурации.

Bridge наспроти host мрежа

Карактеристики	Bridge Networking	Host Networking
Изолација	Контејнерите се изолирани во приватна мрежа на машината домаќин.	Минимална изолација, бидејќи дели мрежа со домаќинот.
Комуникација	Потребно е мапирање на порти за пристап од надвор.	Контејнерите се достапни директно преку мрежата на домаќинот.
Мрежни перформанси	Умерени (поради NAT).	Високи.
Безбедност	Поголема контрола и ограничување на пристап.	Намалена безбедност, директен пристап до мрежата на домаќинот.
Типична употреба	Стандардни и продукциски околини.	Сценарија кои бараат високи перформанси и мала латентност.
Пример	<code>docker run -p 8080:80 myapp</code>	<code>docker run --network host myapp</code>

Overlay наспроти macvlan мрежи

Карактеристики	Overlay Networking	Macvlan Networking
Опсег	Поврзува контејнери на повеќе домаќини.	Работи на истиот домаќин, но во интеграција со физичката мрежа.
Мрежна структура	Креира виртуелна мрежа над постоечката (енкапсулација).	Контејнерите добиваат реални MAC/IP адреси како и другите уреди во LAN.
Употреба	Multi-host системи, Docker Swarm, Kubernetes.	Enterprise и data center средини со контролирана физичка мрежа.
Комуникација	Автоматски DNS service discovery помеѓу контејнери.	Контејнерите комуницираат директно со LAN уредите.
Мрежни перформанси	Мал overhead поради енкапсулација.	Многу високи, директен пристап до мрежата.
Безбедност	Добра логичка изолација меѓу јазли.	Потенцијални мрежни ограничувања од страна на switch / VLAN правила

Инспекција и управување со мрежи

- Докер нуди команди за **следење и управување** на мрежите, овозможувајќи преглед на конфигурации, поврзани контејнери и IP опсези.

```
docker network ls
```

Ги прикажува сите мрежи на системот, нивниот тип и именување.

```
docker network inspect <network_name>
```

Дава детални информации: IP опсези и subnet-и, податоци за поврзаните контејнери, драјвери и конфигурација.

```
docker network create --driver bridge mynetwork
```

Овозможува избирање на driver (bridge, overlay, macvlan...), контрола на subnet и gateway, service discovery и DNS.

```
docker network connect mynetwork mycontainer
```

```
docker network disconnect mynetwork mycontainer
```

Дозволува динамично додавање или отстранување на контејнери од мрежи.

```
docker network rm <network_name>
```

Брише мрежи без да влијае на контејнери поврзани со други мрежи.

Docker Compose

- Docker Compose е моќна алатка која овозможува креирање и управување со апликации составени од **повеќе контејнери**:
 - Дефинира сервиси, мрежи и волумени во еден фајл (пр. docker-compose.yml).
 - Поставува зависности помеѓу контејнери.
 - Подига и спушта цели апликации со една команда.
- **Зошто се користи?**
 - Локален развој: стартување на сите сервиси одеднаш.
 - Тестирање/CI/CD: повторливи и конзистентни средини.
 - Оркестрација без сложен кластер: прост YAML формат.
- **Клучни можности:**
 - Environment variables и build context за секој сервис.
 - Health checks за контејнери.
 - Лесно управување со животниот циклус на апликациите.

Основни команди во Docker Compose

- Docker Compose нуди **едноставни команди** за стартивање, управување и чистење на multi-container апликации.

```
docker-compose up
# Ги подига сите дефинирани сервиси во docker-compose.yml.
# Со -d се стартираат во заднински режим (detached mode).
docker-compose down
# Стопира и ги брише сите контејнери, мрежи и волуумени создадени од Compose.
docker-compose ps
# ps покажува кои сервиси се активни.
docker-compose logs -f
# logs -f го следи тековниот излез на сите контејнери.
docker-compose restart
# restart ги рестартира активните контејнери.
docker-compose up --build
# up --build ги rebuild-а сликите пред старт.
docker-compose exec <service_name> bash
# Дава интерактивен пристап до конкретен контејнер за debugging или конфигурација.
```

Пример за docker-compose.yaml

```
version: '3.9'
services:
  web:
    image: python:3.11-slim
    working_dir: /app
    volumes:
      - ./app:/app
    command: >
      sh -c "pip install flask psycopg2-binary &&
            python app.py"
    ports:
      - "5000:5000"
    environment:
      DATABASE_URL: postgres://user:password@db:5432/mydb
    depends_on:
      - db
  db:
    image: postgres:15
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb
```

Објаснување:

- **web**: Flask апликација што се поврзува со база.
- **db**: PostgreSQL база со default credentials.
- depends_on гарантира дека базата ќе стартира пред веб сервисот.

Заклучок: складирање, мрежи, повеќеконтејнерски апликации

- Докер нуди **ефикасни решенија** за складирање, мрежна комуникација и управување со повеќеконтејнерски апликации.
- Со волумени и врзани монтирања, контејнерите може да овозможат **перзистентност** на податоци и лесно читање/пишување на логови и конфигурации помеѓу контејнери и домаќинот.
- Во делот на **мрежите**, Докер обезбедува различни конфигурации, кои нудат различни нивоа на изолација, комуникација и интеграција со физички мрежи.
- Со Docker Compose, цели апликации со повеќеконтенери се дефинираат во **еден YAML фајл**, овозможувајќи лесно стартивање, спуштање и управување со зависности и променливи.

Содржина

-
- 01** Зошто се потребни контејнери?
 - 02** Архитектура и внатрешна структура на Докер
 - 03** Градење и извршување на контејнери
 - 04** Складирање, мрежа и повеќеконтејнерски системи
 - 05** Современи предизвици и најдобри практики
-

Оптимизација на Докер слики

Оптимизацијата на Докер слики е важна за побрзо градење, ефикасна дистрибуција и поголема безбедност.

- Користењето на минимални основни слики, како Alpine, ја намалува обемот на сликата и го скратува времето за пренос.
- Комбинирањето на повеќе RUN команди во една намалува број на слоеви и спречува непотребно кеширање, а чистењето на привремени фајлови дополнително ја намалува големината.
- Градењето во повеќе фази овозможува одделување на алатките и зависностите за градење од самата извршна средина, оставајќи ги само потребните бинарни фајлови во финалната слика и намалувајќи го ризикот од безбедносни пропусти.

```
RUN apt-get update && \
apt-get install -y \
curl \
git && \
rm -rf
/var/lib/apt/lists/*
```

```
# Build stage
FROM golang:1.20 AS builder
WORKDIR /app
COPY .
RUN go build -o myapp
```

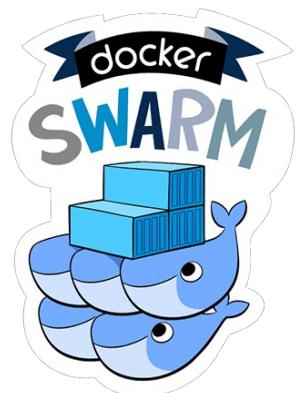
```
# Runtime stage
FROM alpine
COPY --from=builder
/app/myapp /myapp
CMD ["/myapp"]
```

Безбедност во Docker

- Безбедноста е клучна, бидејќи контејнерите делат **kernel** со домаќинот.
- Rootless Docker овозможува извршување на контејнери **без root** привилегии, што ја намалува можноста за напади и злоупотреба на системот.
- Seccomp профили ја ограничуваат листата на дозволени **системски повици**, додека AppArmor и SELinux го контролираат **пристапот** до датотеки, процеси и мрежни ресурси.
- **Скенирањето** на Докер слики за познати ранливости со алатки како Trivy или Clair помага да се откријат потенцијални закани пред пуштањето во продукција.
- Покрај тоа, користењето на **non-root корисници** и read-only файл системи ја штити инфраструктурата и ресурсите на домаќинот.

Оркестрација со Swarm и Kubernetes

- **Overlay** мрежата им овозможува на контејнерите на различни домаќини да комуницираат како да се на ист систем.
- Docker Swarm овозможува формирање на **кластер од контејнери** на повеќе домаќини, со автоматско распоредување на услуги, постепени ажурирања и безбедно управување со тајни податоци.
- За поголеми апликации, **Kubernetes** нуди автоматско скалирање, само-опоравување на неуспешни јазли, декларативна конфигурација и интеграција со алатки за мониторинг и управување.
- Докер сликите се **компатибилни со Kubernetes**, што овозможува лесна миграција од локална средина кон големи производсвки системи.



Најдобри практики во CI/CD

- За продукциски околини, најдобри практики се користење на **фиксирани** верзии на слики, **health checks** за контејнери, ограничување на ресурси и **централизирано** следење и логирање.
- Docker овозможува изолирани и **конзистентни средини** за автоматско градење и тестирање, елиминирајќи проблеми како „работи на мојата машина“.
- CI/CD pipelines овозможуваат автоматско градење, тестирање и пуштање на сликите, овозможувајќи сигурно, предвидливо и **модуларно работење** на апликациите.
- Комбинацијата на оптимизација, безбедност и организација создава системи кои се скалабилни, отпорни и лесни за одржување во cloud-native средини.

Дополнителна литература

- Docker Official Documentation
<https://docs.docker.com/>
- Play with Docker (Official Interactive Labs)
<https://labs.play-with-docker.com/>
- Docker Educational Resources
<https://docs.docker.com/get-started/resources/>
- DataCamp: How to Learn Docker from Scratch
<https://www.datacamp.com/blog/learn-docker>