

Effizient, produktiv & unabhängig: Schlanke DevOps-Workflows mit Docker

Sven Vinkemeier



MOTIVATION

Situation:

- Build-Server von anderem Team bereitgestellt & gewartet
- Kein Zugriff via SSH o. Ä.
- Schwierige Kommunikation mit dem Team (verschiedene Zeitzonen etc.)
- Häufige Deployments zu erwarten

Fazit: Wir brauchen...

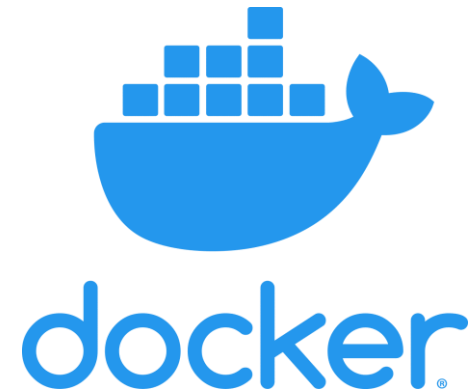
- **Unabhängigkeit** vom Build-Server
- **Effiziente**, schnelle Builds
- **Produktivität** beim Testen/Debuggen der Build-Pipelines



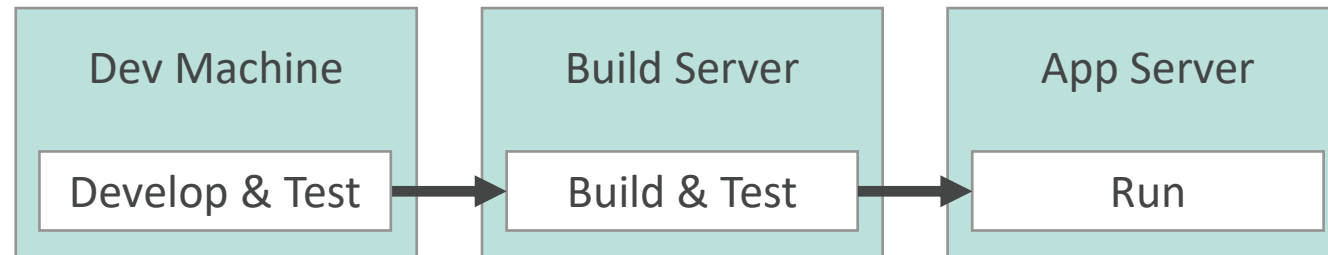
ZIEL

Schrittweise Containerisierung des DevOps-Prozesses

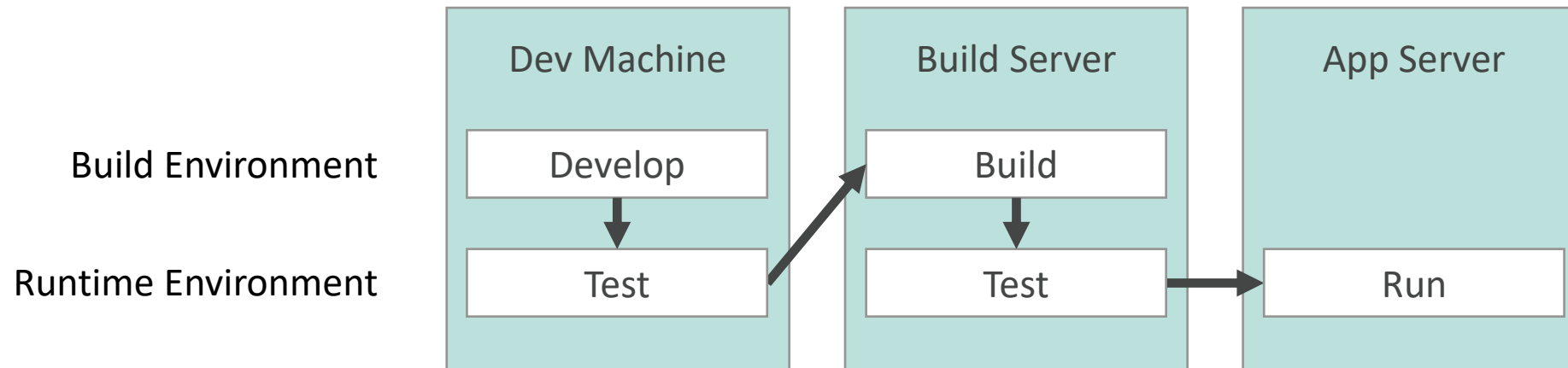
- Ausführung im Container
- Bauen im Container
- Entwickeln im Container



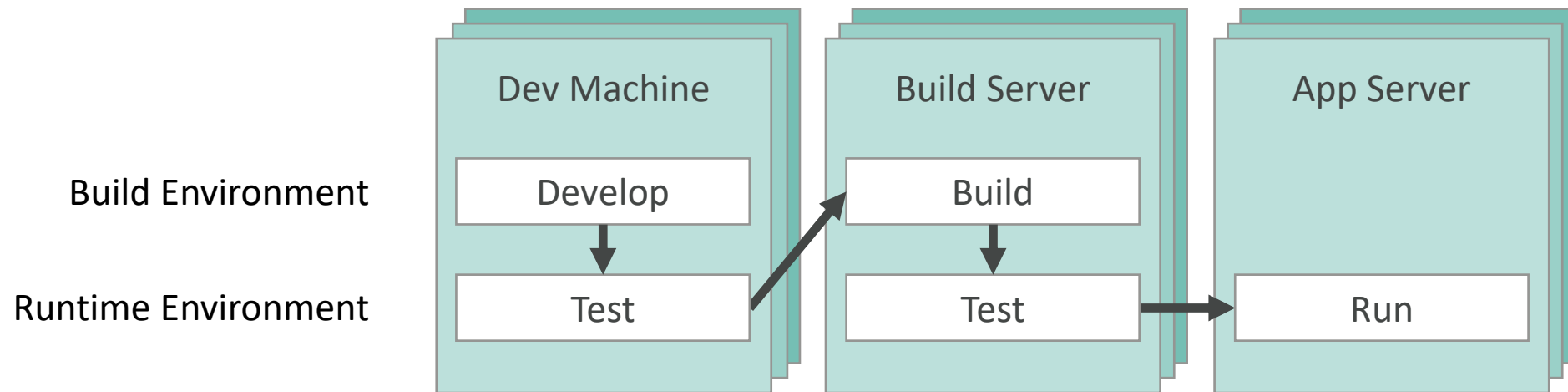
CI/CD OHNE DOCKER



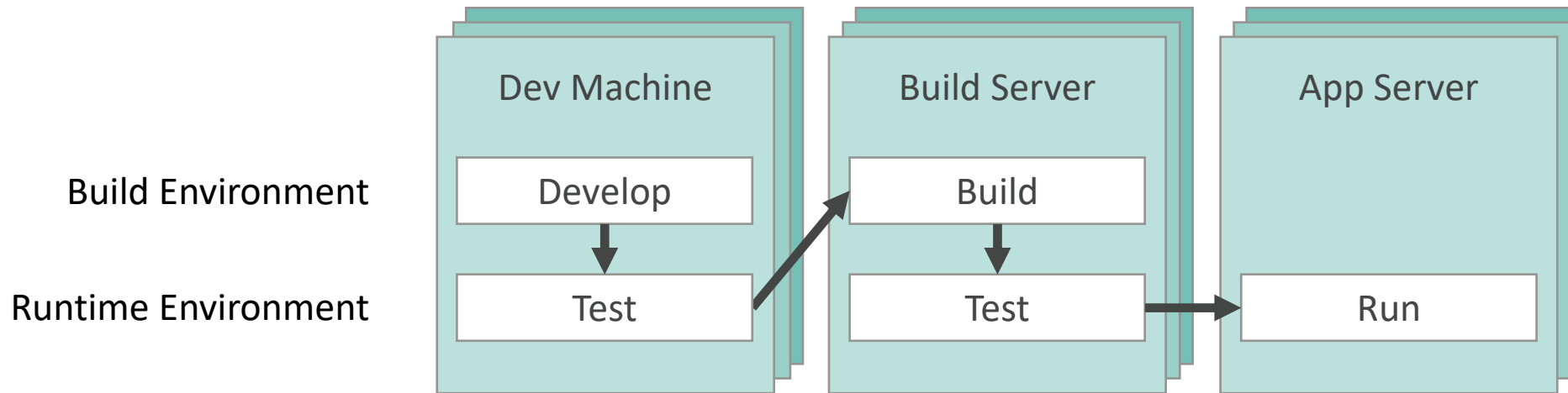
CI/CD OHNE DOCKER



CI/CD OHNE DOCKER



CI/CD OHNE DOCKER



Konsequenz unterschiedlicher Laufzeitumgebungen:

Dokumentationsaufwand

😬 Anforderungen müssen notiert und aktuell gehalten werden

Hoher Wartungsaufwand

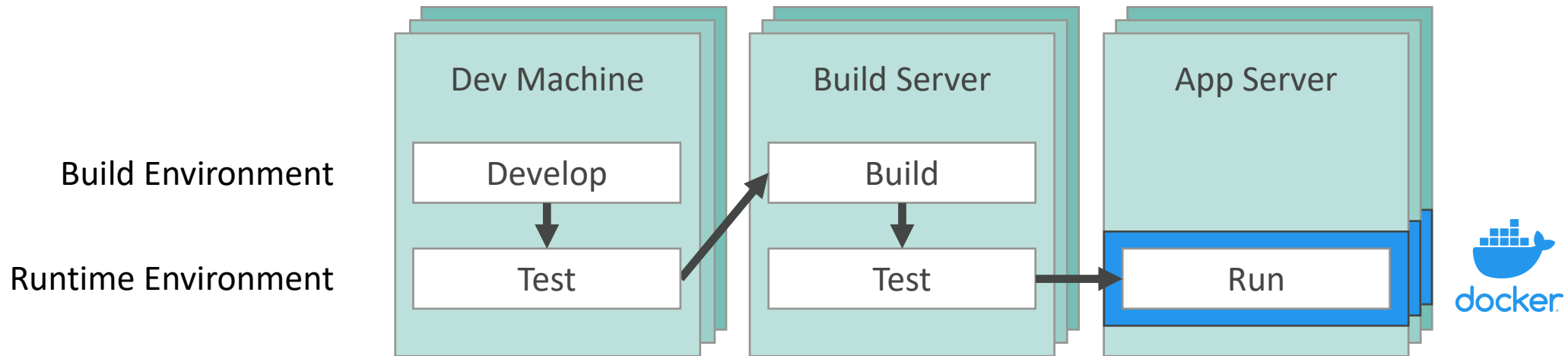
😬 Software muss überall synchron und aktuell gehalten werden

Hohe Wahrscheinlichkeit für abweichendes Verhalten

😬 Unterschiede bzgl. OS, installierter Software, Konfiguration usw.



CI/CD MIT DOCKER



Vorteile durch Ausführung im Container:

~~Dokumentationsaufwand~~

- ✓ Dockerfile dokumentiert Anforderungen an Umgebung

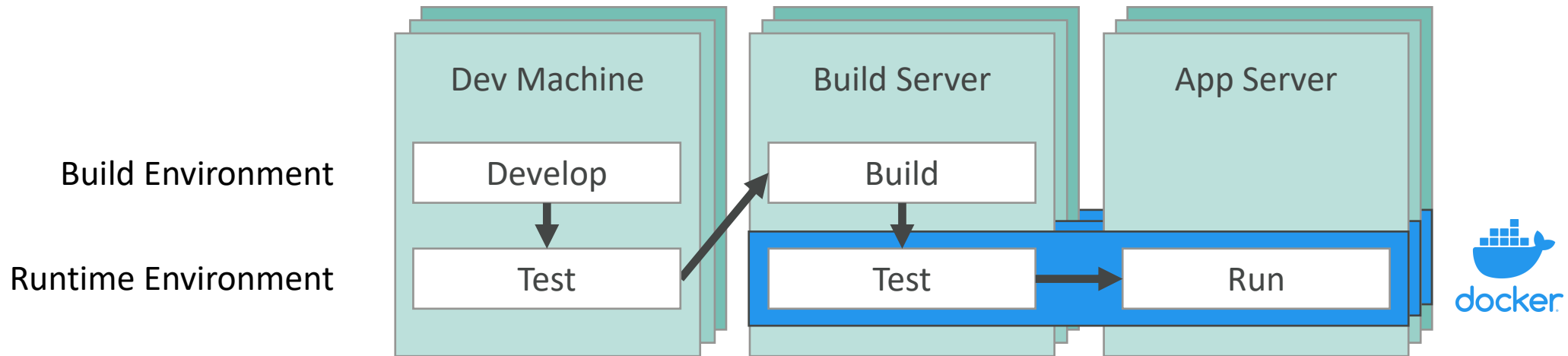
~~Hoher Wartungsaufwand~~

- ✓ Wiederverwendbare Containerdefinitionen

~~Hohe Wahrscheinlichkeit für abweichendes Verhalten~~

- ✓ Weitgehende Isolation vom Host

CI/CD MIT DOCKER



Vorteile durch Ausführung im Container:

~~Dokumentationsaufwand~~

- ✓ Dockerfile dokumentiert Anforderungen an Umgebung

~~Hoher Wartungsaufwand~~

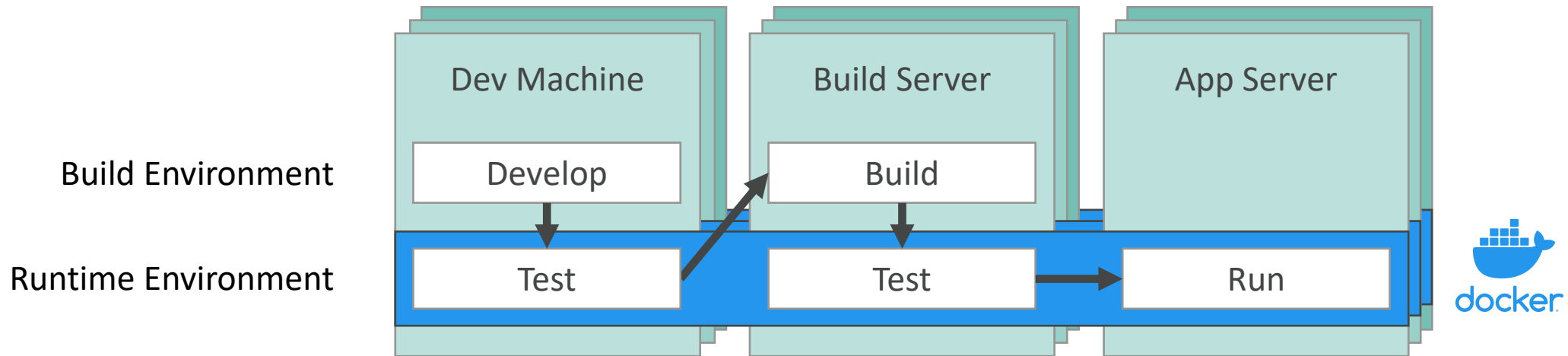
- ✓ Wiederverwendbare Containerdefinitionen

~~Hohe Wahrscheinlichkeit für abweichendes Verhalten~~

- ✓ Weitgehende Isolation vom Host



CI/CD MIT DOCKER



Vorteile durch Ausführung im Container:

~~Dokumentationsaufwand~~

- ✓ Dockerfile dokumentiert Anforderungen an Umgebung

~~Hoher Wartungsaufwand~~

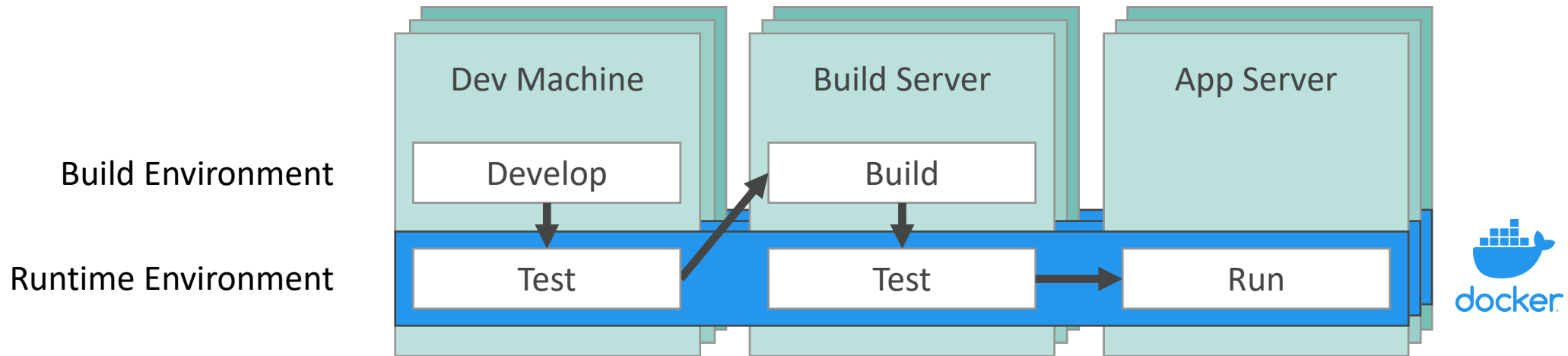
- ✓ Wiederverwendbare Containerdefinitionen

~~Hohe Wahrscheinlichkeit für abweichendes Verhalten~~

- ✓ Weitgehende Isolation vom Host



CI/CD MIT DOCKER



Was ist mit den Build-Umgebungen?

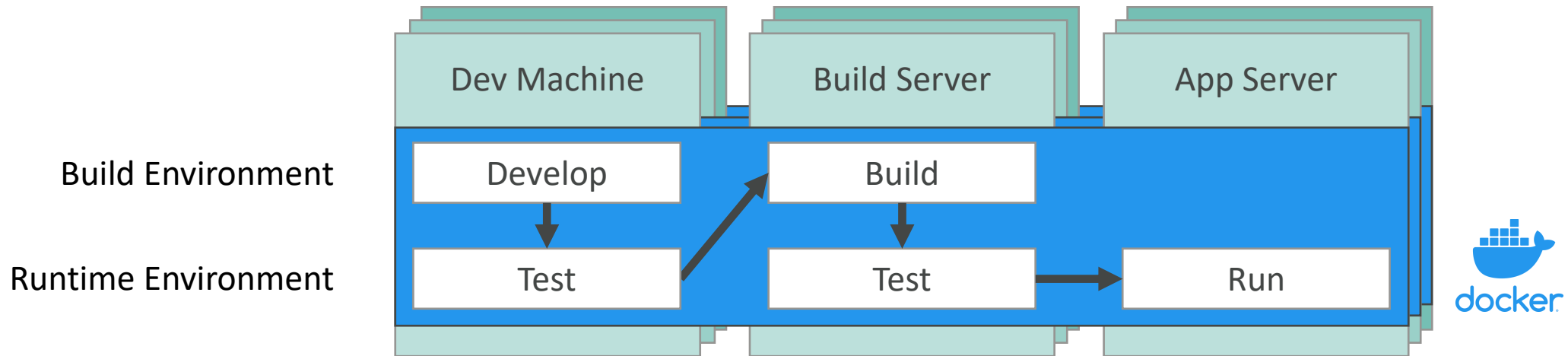
- 😞 Dokumentationsaufwand
- 😞 Hoher Wartungsaufwand
- 😞 Hohe Wahrscheinlichkeit für abweichendes Verhalten

Außerdem:

- 😞 Vendor Lock-in durch Spezifika der CI-Plattform
- 😞 Aufwendiges Onboarding neuer Entwickler*innen



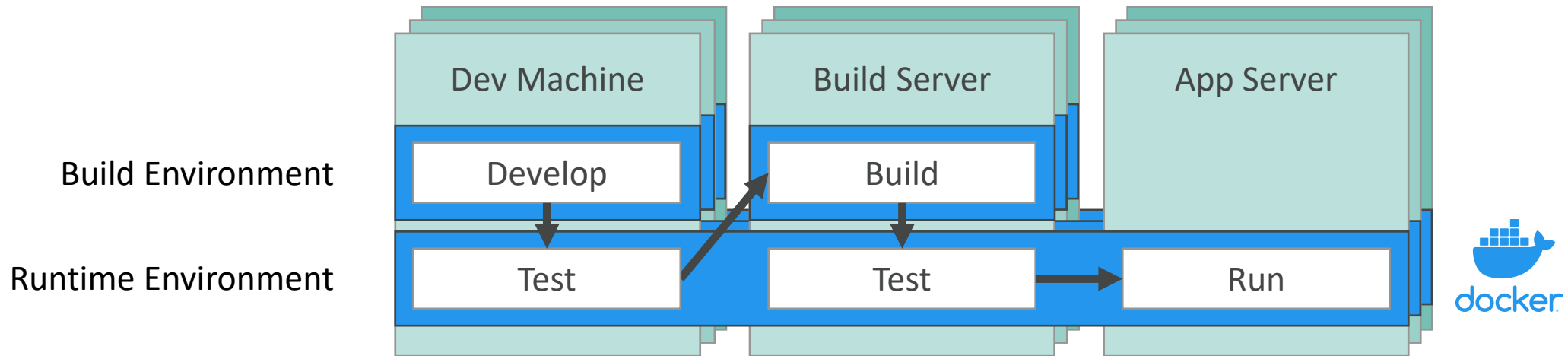
CI/CD MIT DOCKER



Ein Containerimage für alles? Nicht sinnvoll!

Production Image sollte minimal sein
(Performance optimieren, Angriffsfläche minimieren)

CI/CD MIT DOCKER



Ein Containerimage für alles? Nicht sinnvoll!


Production Image sollte minimal sein
(Performance optimieren, Angriffsfläche minimieren)

Lösung: Multi-stage builds

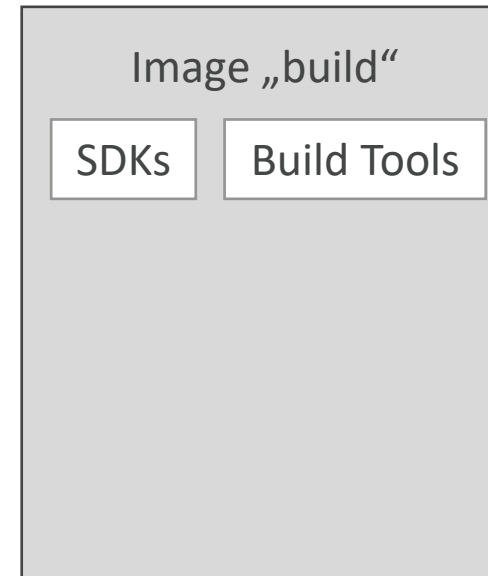
Ein Dockerfile, verschiedene Images je nach Einsatzzweck

BAUEN IM CONTAINER

Dockerfile


 `FROM node:16 as build`
`WORKDIR /workspace`
`COPY . .`
`RUN npm install`
`RUN npm run build`

`FROM nginx:stable-alpine`
`COPY --from=build /workspace/build ↵`
`/usr/share/nginx/html`

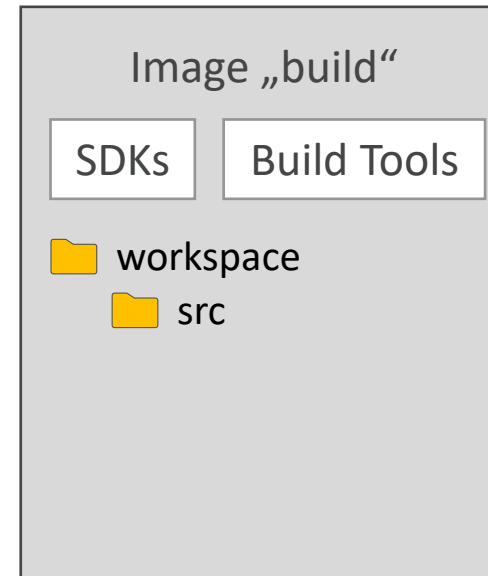


BAUEN IM CONTAINER

Dockerfile


```
FROM node:16 as build
WORKDIR /workspace
 COPY . .
RUN npm install
RUN npm run build

FROM nginx:stable-alpine
COPY --from=build /workspace/build ↵
  /usr/share/nginx/html
```

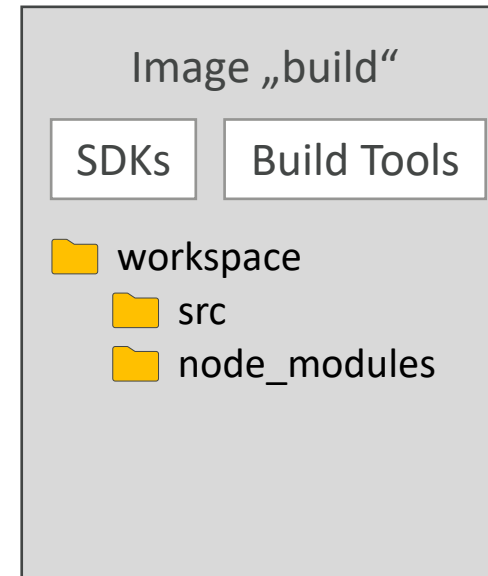


BAUEN IM CONTAINER

Dockerfile


```
FROM node:16 as build
WORKDIR /workspace
COPY . .
 RUN npm install
RUN npm run build

FROM nginx:stable-alpine
COPY --from=build /workspace/build ↵
  /usr/share/nginx/html
```

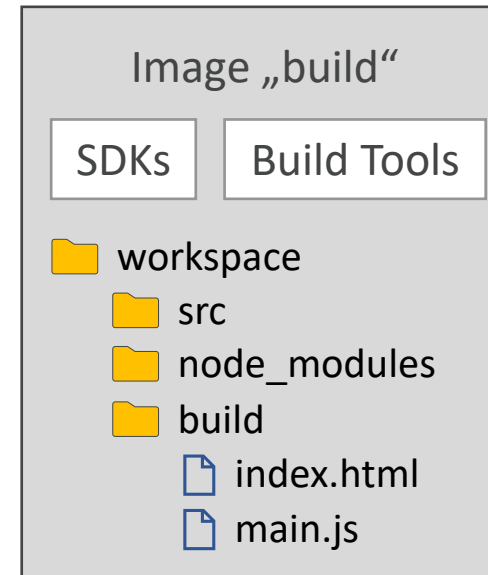


BAUEN IM CONTAINER

Dockerfile

```
FROM node:16 as build
WORKDIR /workspace
COPY . .
RUN npm install
 RUN npm run build

FROM nginx:stable-alpine
COPY --from=build /workspace/build ↵
  /usr/share/nginx/html
```



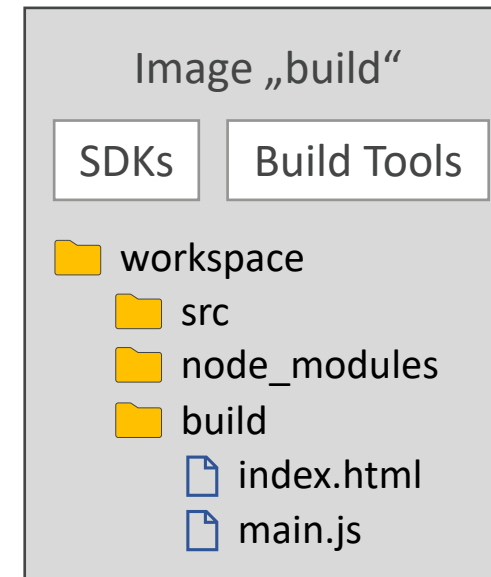
BAUEN IM CONTAINER

Dockerfile

```
FROM node:16 as build
WORKDIR /workspace
COPY . .
RUN npm install
RUN npm run build
```

👉

```
FROM nginx:stable-alpine
COPY --from=build /workspace/build ↵
  /usr/share/nginx/html
```



BAUEN IM CONTAINER

Dockerfile

```
FROM node:16 as build
```


```
WORKDIR /workspace
```

```
COPY . .
```

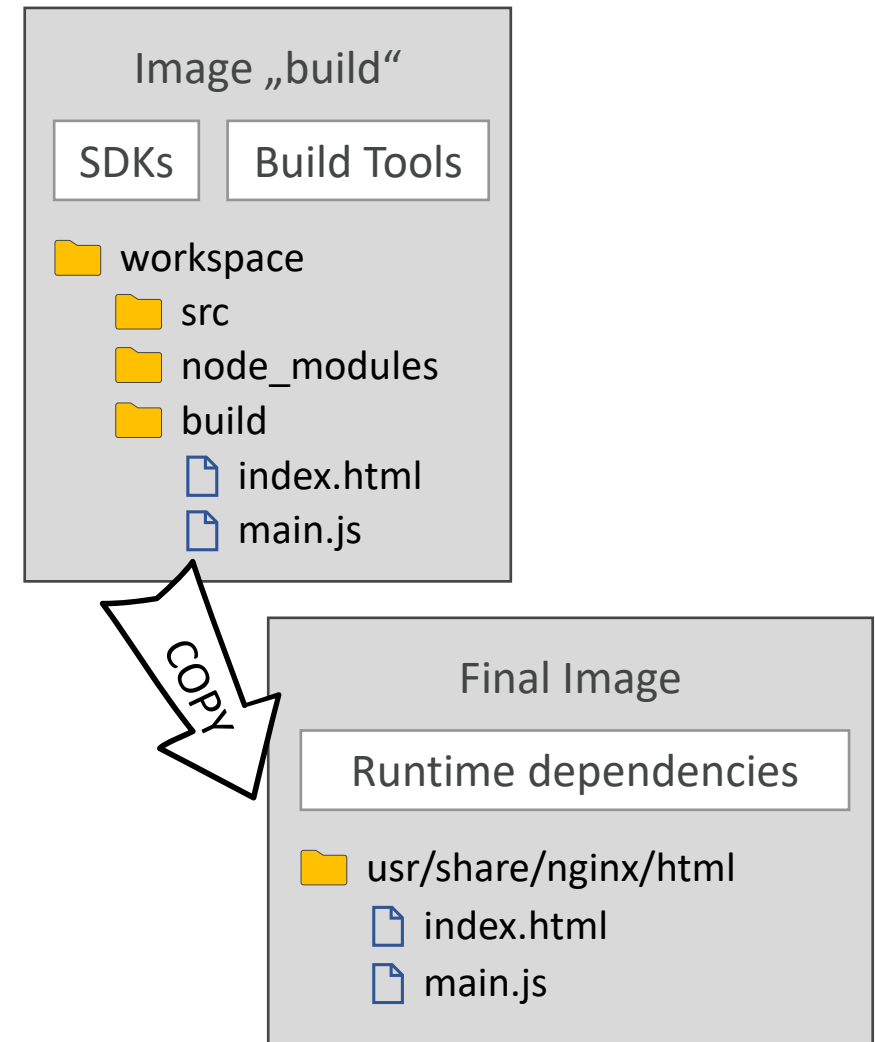
```
RUN npm install
```

```
RUN npm run build
```

```
FROM nginx:stable-alpine
```



```
COPY --from=build /workspace/build ↵  
/usr/share/nginx/html
```

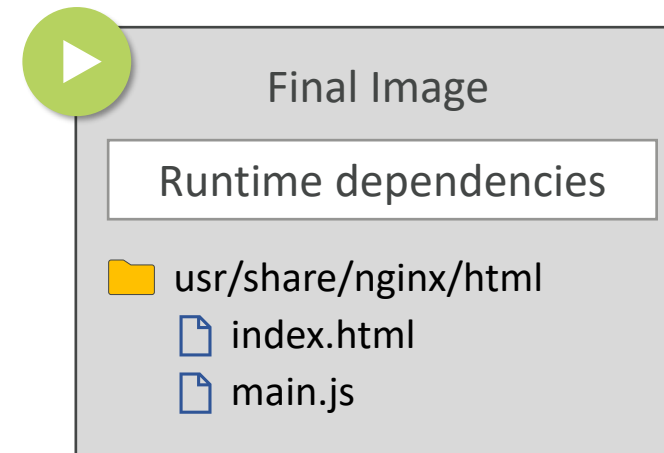


BAUEN IM CONTAINER

Dockerfile

```
FROM node:16 as build
WORKDIR /workspace
COPY . .
RUN npm install
RUN npm run build

FROM nginx:stable-alpine
COPY --from=build /workspace/build ↵
  /usr/share/nginx/html
```



DOCKER CACHING

Gratis Bulddauer-Optimierung!

Teilergebnisse vorheriger Builds werden wiederverwendet, wenn sich Abhängigkeiten nicht geändert haben

Insbesondere für **Monorepos** relevant!

Dockerfile

```
FROM node:16 as build
WORKDIR /workspace
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build

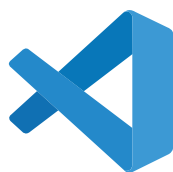
FROM nginx:stable-alpine
COPY --from=build /workspace/build ↵
  /usr/share/nginx/html
```



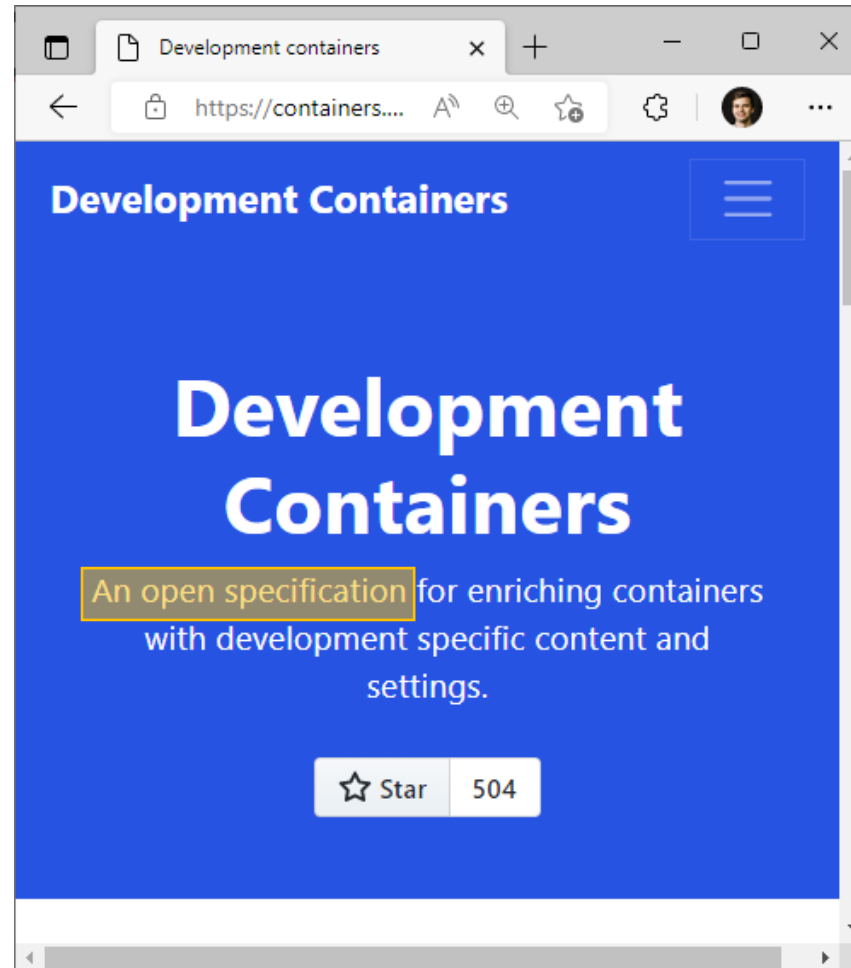
DEVELOPMENT CONTAINERS

A development container allows you to use a container as a full-featured development environment. It can be used to run an application, to separate tools, libraries, or runtimes needed for working with a codebase, and to aid in continuous integration and testing. Dev containers can be run locally or remotely, in a private or public cloud.

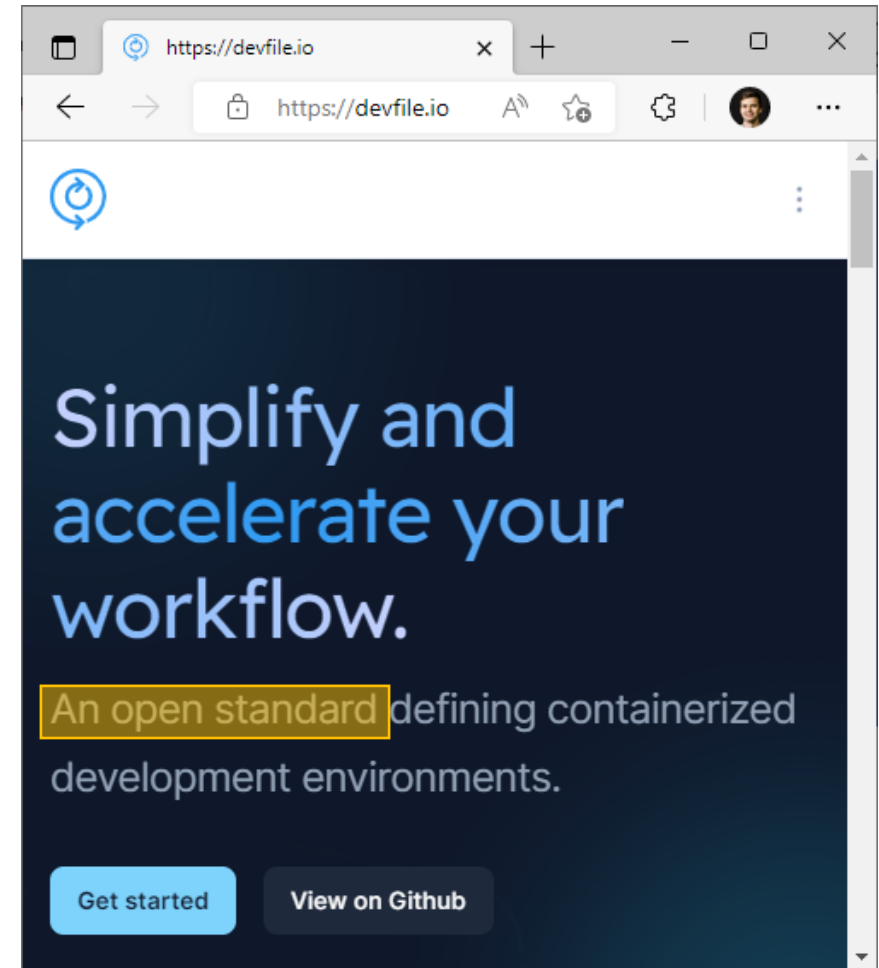
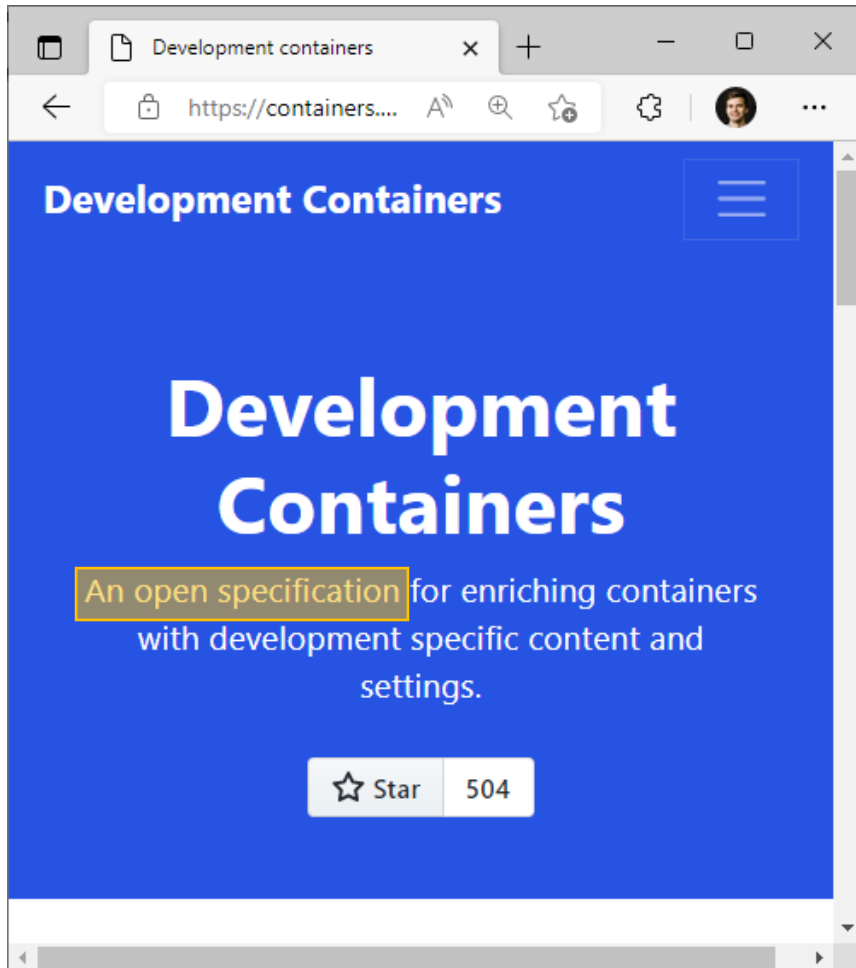
<https://containers.dev/>



DEVELOPMENT CONTAINERS



DEVELOPMENT CONTAINERS



DEVCONTAINER

Ein Dockerfile,
drei Stages:

- **dev** zur lokalen Entwicklung
- **build** für CI-Builds und automatisierte Tests
- **Finale Stage** fürs Deployment

Dockerfile

```
FROM [...] /devcontainers/typescript-node:[...] as dev

FROM dev as build
WORKDIR /ws
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build

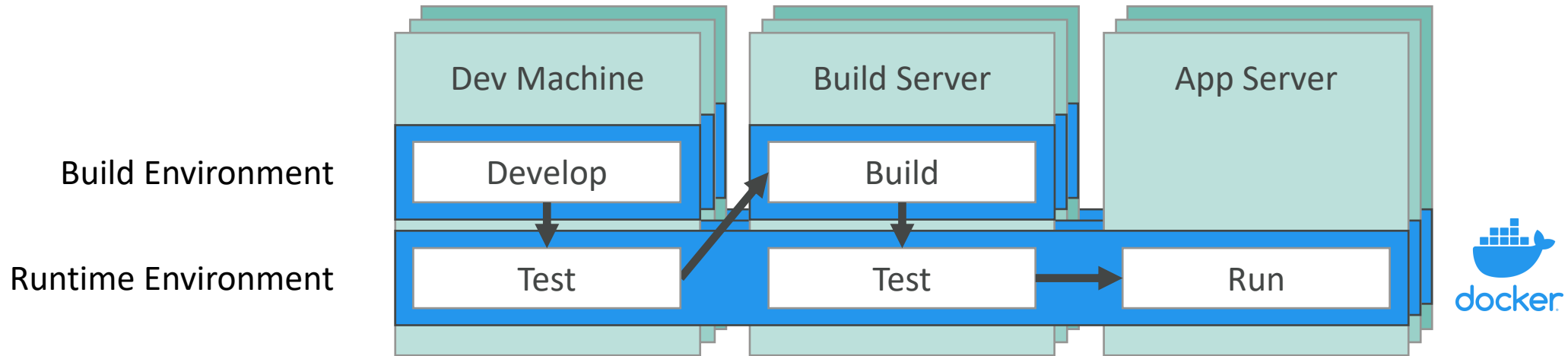
FROM nginx:stable-alpine
COPY --from=build /ws/build /usr/share/nginx/html
```

devcontainer.json

```
{
  "workspaceFolder": "/ws",
  "build": {
    "dockerfile": "Dockerfile",
    "target": "dev"
  },
  "postCreateCommand": "npm install",
  // ...
}
```



ERGEBNIS



- ✓ **Effizient:** Reduzierte Bulddauer durch Caching
- ✓ **Produktiv:** Reduzierter Wartungsaufwand, schnelles Onboarding neuer Devs
- ✓ **Unabhängig:** Geringe Kopplung an Vendor-spezifische Technologien

