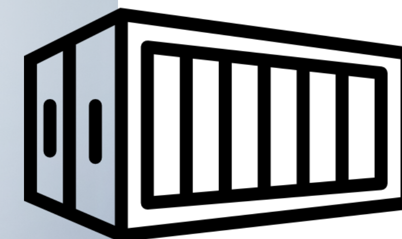
CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Container

(4) 12 Factor Apps und Container (Anti-)Pattern



PROF. DR.
NANE KRATZKE

creative commons

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CC0 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CC0 Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

CLOUD NATIVE COMPUTING

KAPITEL 8

Standardisierung von Deployment Units (Container)

8 Standardisierung von Deployment Units

8.1 Hintergrund (PaaS)

8.2 Betriebssystem-Virtualisierung

8.3 Container Runtime Environments

- Kernel-Namespaces
- Process Capabilities
- Control Groups
- Union Filesystems
- High- und Low-Level Container-Laufzeitumgebungen

8.4 Bau und Bereitstellung von Container-Images

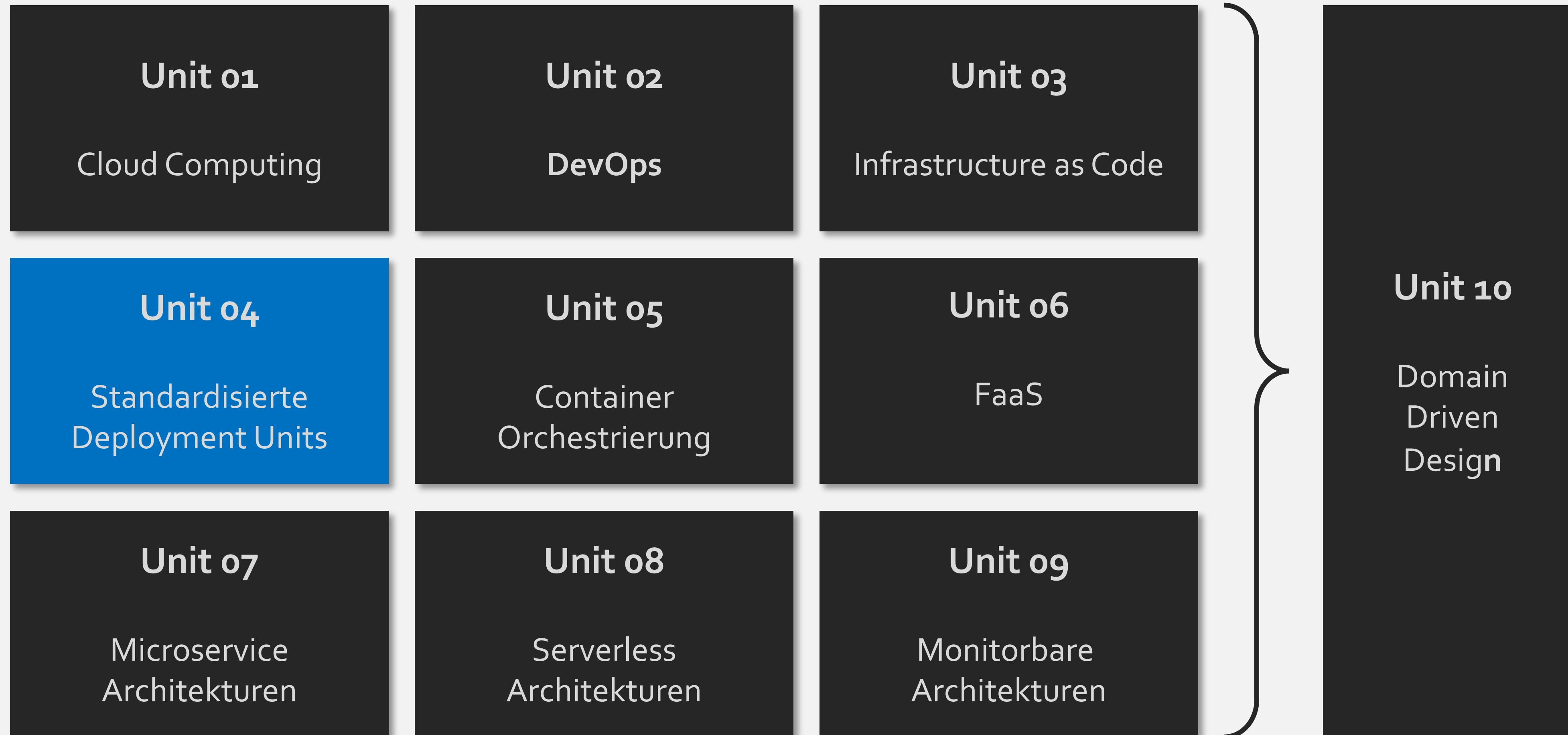
8.5 Faktoren gut betreibbarer Container

- Codebase
- Abhängigkeiten und Konfigurationen
- Unterstützende Services und Port Binding
- Build-, Release- und Run-Phase
- Horizontale Skalierung über Prozesse
- Umgebungen, Logs und Betrieb

8.6 Zusammenfassung

INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



CLOUD NATIVE COMPUTING

INHALTE

Hintergrund

- Platform as a Service
- Das PaaS-Problem
- Das CaaS-Versprechen

Betriebssystem-Virtualisierung

- OS-Virtualisierung
- Linux-basierte Techniken zur OS-Virtualisierung
- Standardisierung von Deployment-Einheiten => Container

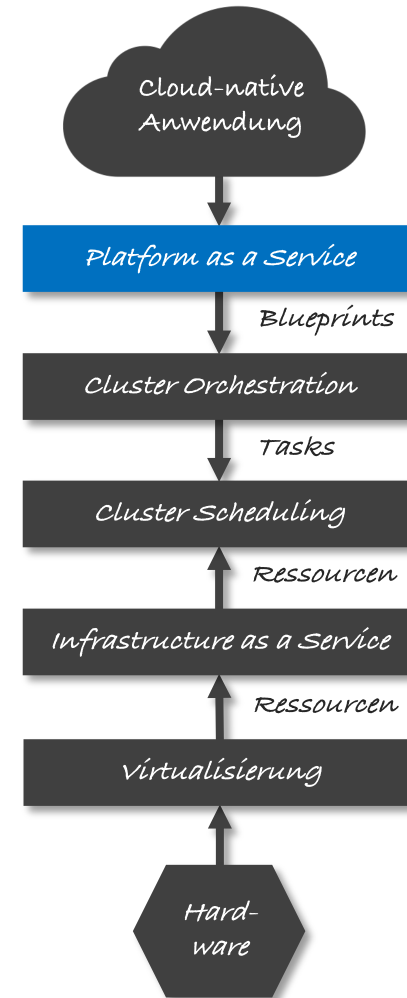
Laufzeitumgebungen für Container

- Container Laufzeitumgebungen und Standards
- Docker
- Image Building und Registries

Container-Pattern

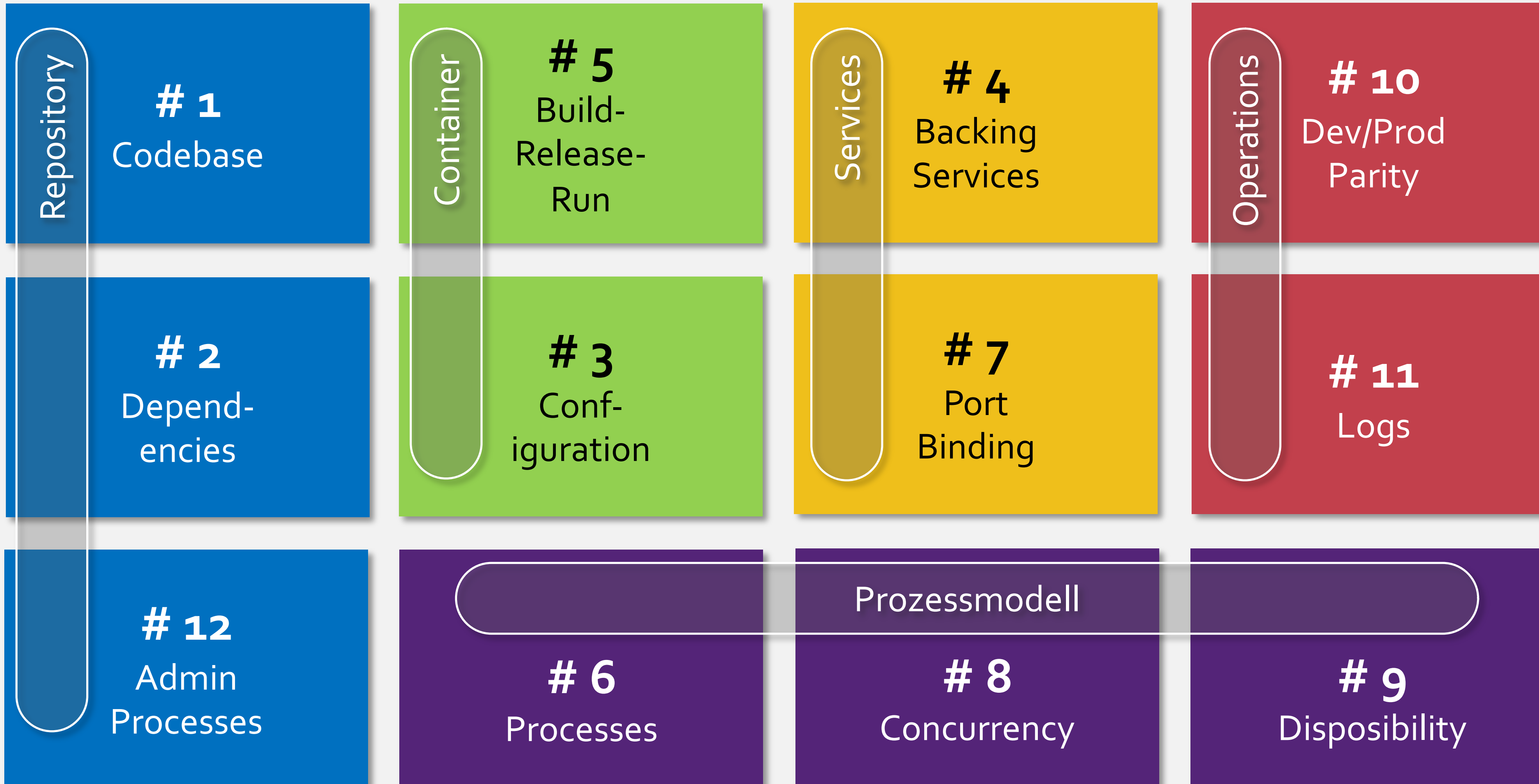
- Container (Anti-)Pattern
- 12-Factor Apps

CLOUD NATIVE COMPUTING



12 FAKTOREN

Best Practices für den Betrieb von Containern

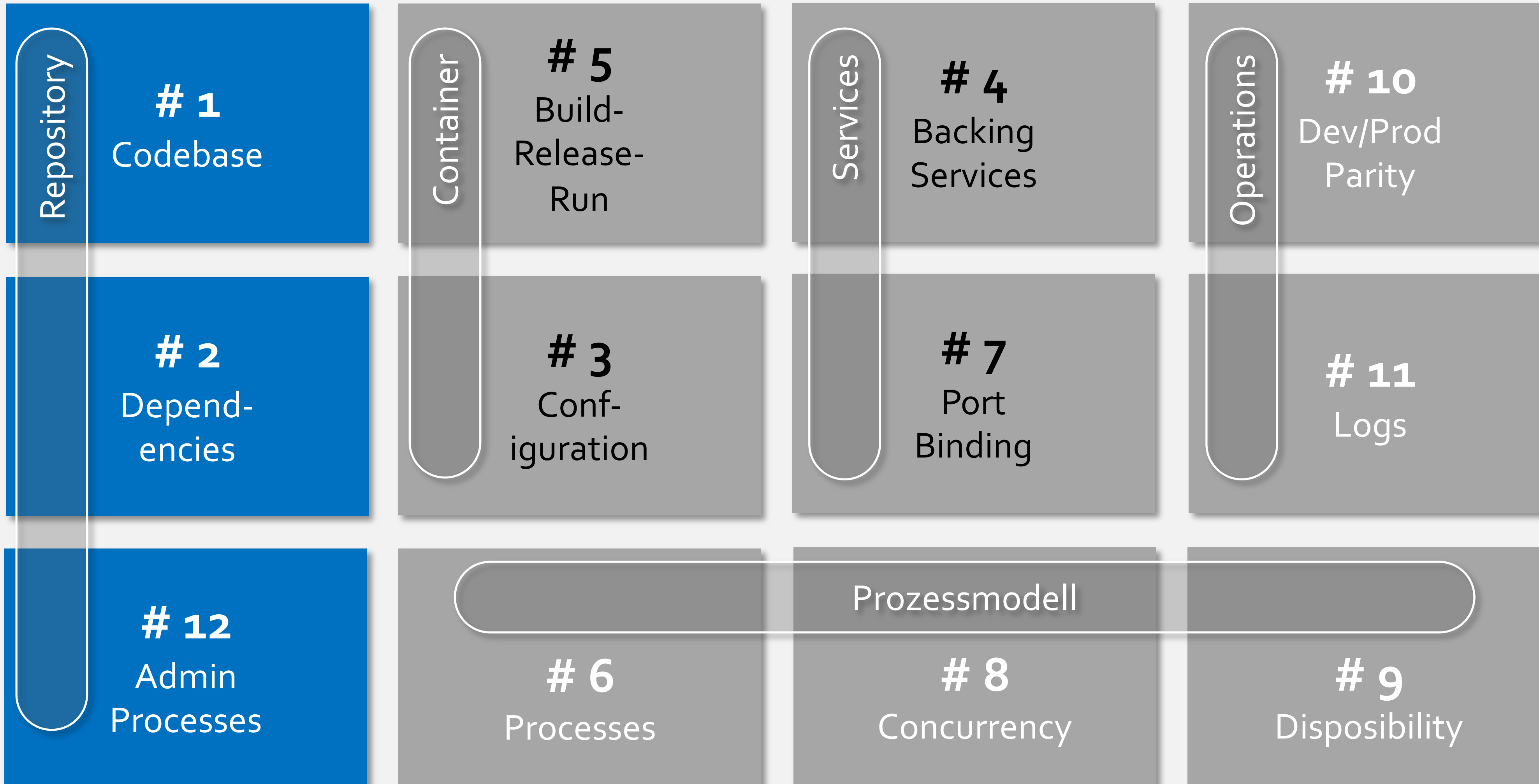


CLOUD NATIVE COMPUTING

Die 12-Faktoren-Methodik ist eine Reihe von Best Practices für die Entwicklung von Cloud-nativen Anwendungen, die portabel, skalierbar, robust, einfach zu betreiben und skalierbar sind. Die Faktoren erleichtern die Bereitstellung und den Betrieb in verschiedenen Umgebungen wie Entwicklungs-,Test- und Produktivsystemen in Private oder Public Cloud-Infrastrukturen oder Container-Plattformen wie Kubernetes.

12 FAKTOREN

Best Practices für den Betrieb von Containern

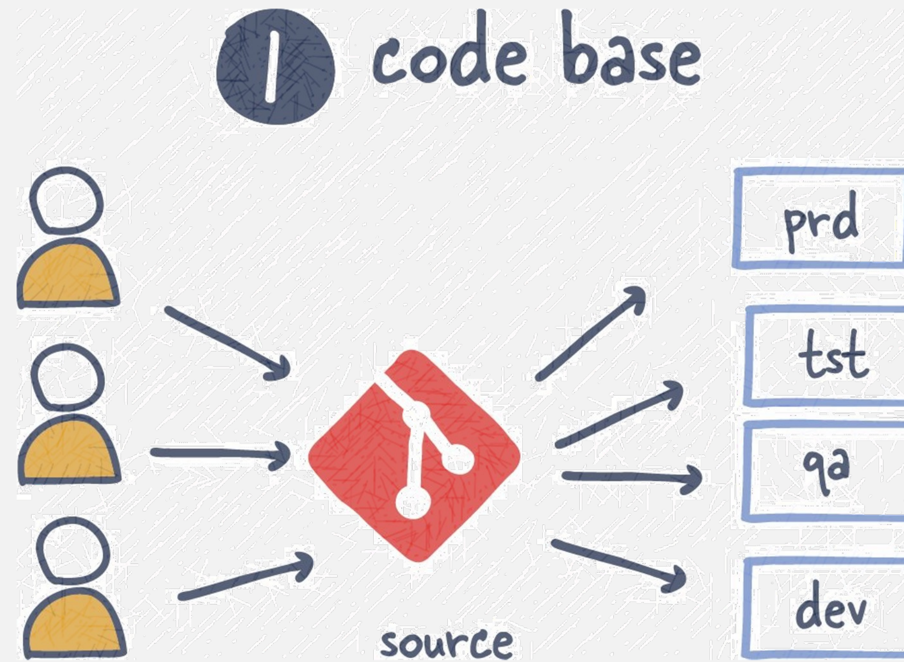


CLOUD NATIVE COMPUTING

12 FAKTOREN METHODE

I. Codebase (Baue Anwendungen aus einer einzigen Codebase)

- Eine Anwendung soll aus einer einzigen Codebase aus einem Versionskontrollsystem heraus gebaut werden
- Die Codebase umfasst den gesamten Code, der für die Entwicklung und Deploy der Anwendung notwendig ist:
  - Frameworks
  - Bibliotheken
  - und weitere Abhängigkeiten (Daten, etc.)



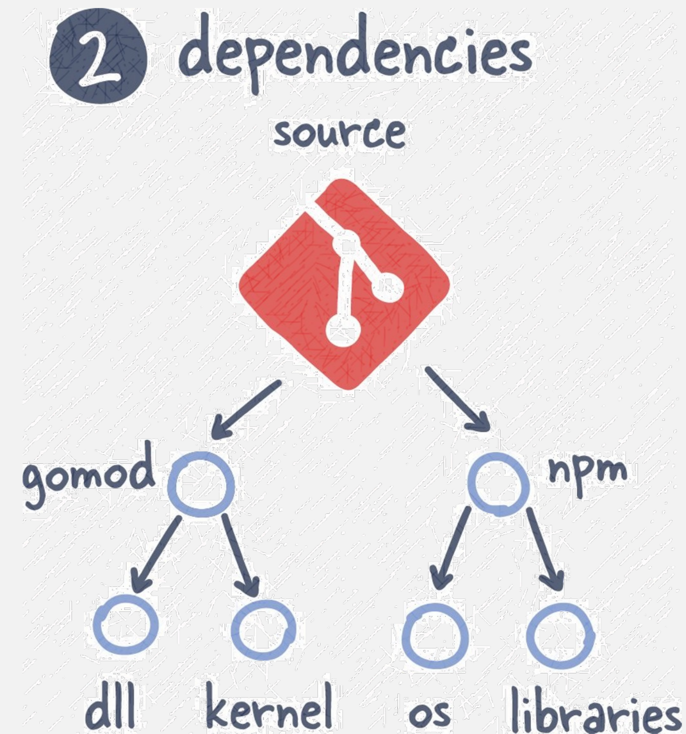
Beispiel: Eine Node.js Anwendung wird in einem Git-Repository versioniert. Wenn Entwickler Änderungen an der Anwendung vornehmen, können alle Projektbeteiligten mit derselben Version der Anwendung arbeiten. Die Anwendung kann so jederzeit in verschiedenen Versionen deployt werden.

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

12 FAKTOREN METHODE

II. Abhängigkeiten (Deklariere und isoliere Abhängigkeiten)

- Abhängigkeiten von Bibliotheken und anderen Komponenten deklarativ mit Dependency-Management-Systemen erfassen
- So wird eine einheitliche und vorhersehbare Abhängigkeitsumgebung geschaffen
- Minimierung von Abhängigkeitskonflikten und Inkompatibilitäten
- Zwölf-Faktor-Anwendungen verlassen sich nie auf die implizite Existenz von Systemtools, wie ImageMagick oder curl und installieren diese notfalls selber.



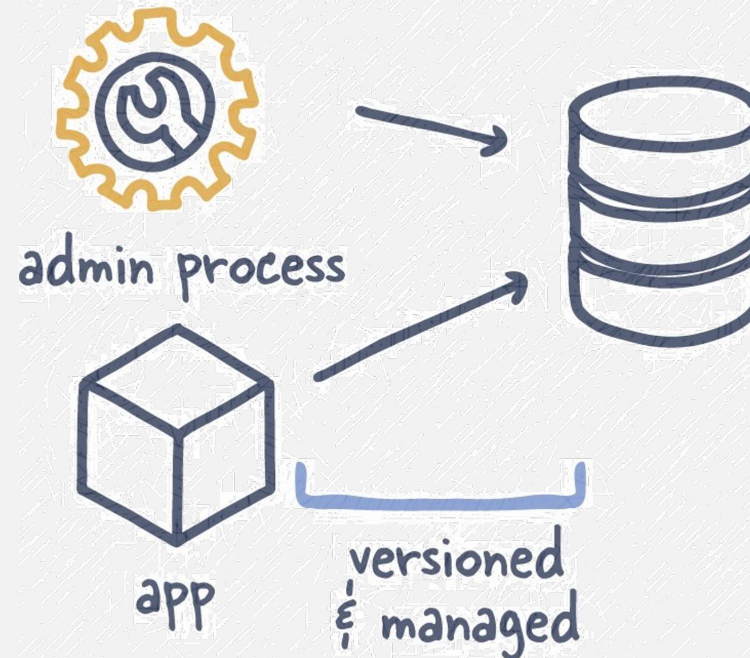
Beispiel: Eine Anwendung, die eine bestimmte Version einer Bibliothek benötigt, sollte diese in der Datei des Package-Management-Systems (wie bspw. pip, Gemfile, Maven) definieren, die zusammen mit dem Anwendungscode im Versionskontrollsystem gespeichert wird. Diese Datei kann dann von der Build-Pipeline der Anwendung genutzt werden, um sicherzustellen, dass immer die richtigen Abhängigkeiten in der Anwendungs- oder Testumgebung installiert werden.

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

12 FAKTOREN METHODE

XII. Admin Prozesse (Vergesse nicht deinen Admin-Code)

- Alles, was den Zustand einer Anwendung ändert, sollte wie die Anwendung selbst behandelt und in die Codebase eingecheckt werden, z.B.:
  - Skripte zur Durchführung von Datenbankmigrationen oder um die Funktionstüchtigkeit der Anwendung zu überprüfen (mittels REPL-Shells)
  - Ausführen einmaliger Skripte (fix_bad_records.php)
  - usw.
- Auch Admin-Code muss in der identischen Umgebung wie die regulären Prozesse der Anwendung ausgeführt werden

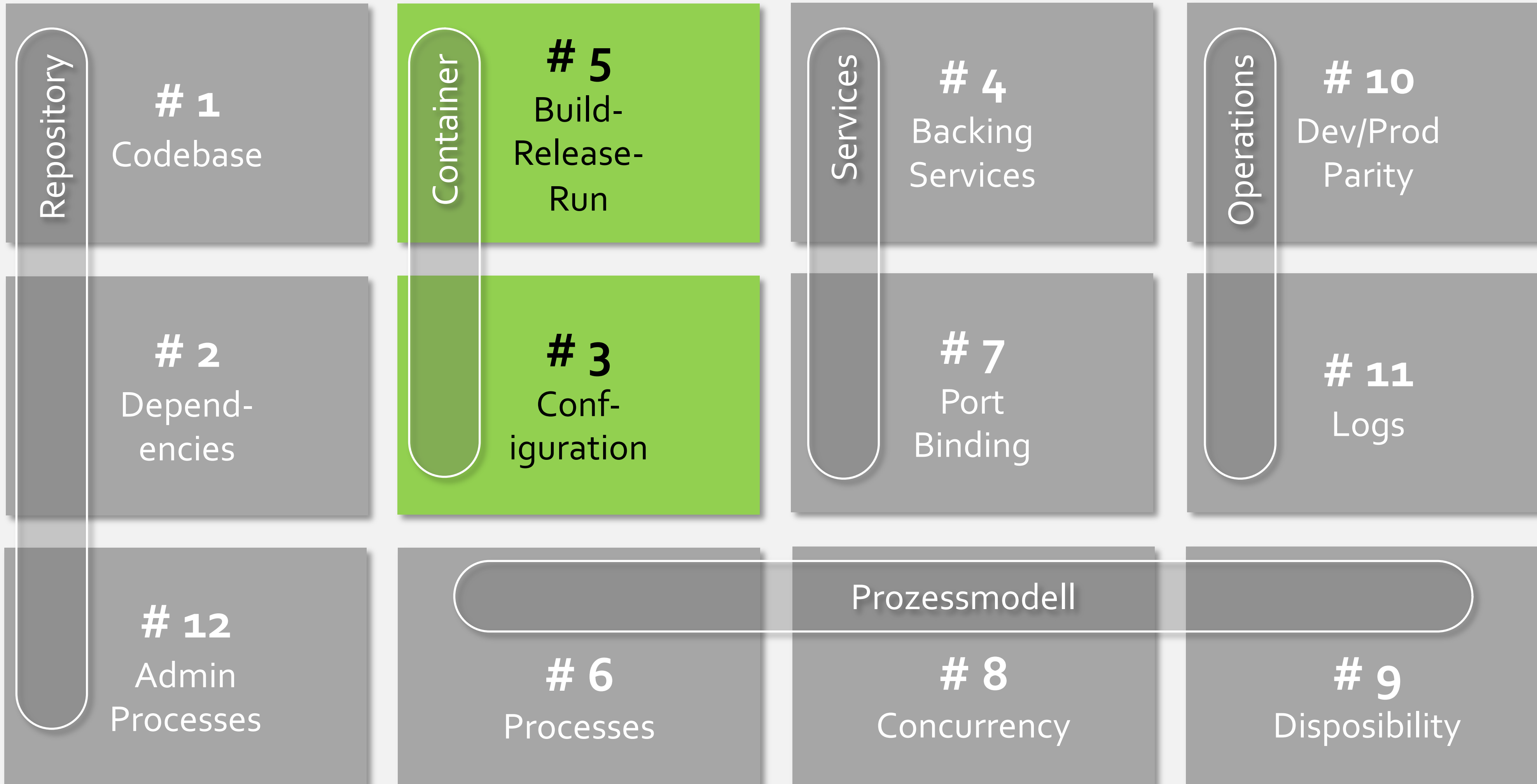


Bildquelle: https://architecturenotes.co/12-factor-app-revisited/x

CLOUD NATIVE COMPUTING
PROF. DR. NANE KRATZKE

12 FAKTOREN

Best Practices für den Betrieb von Containern

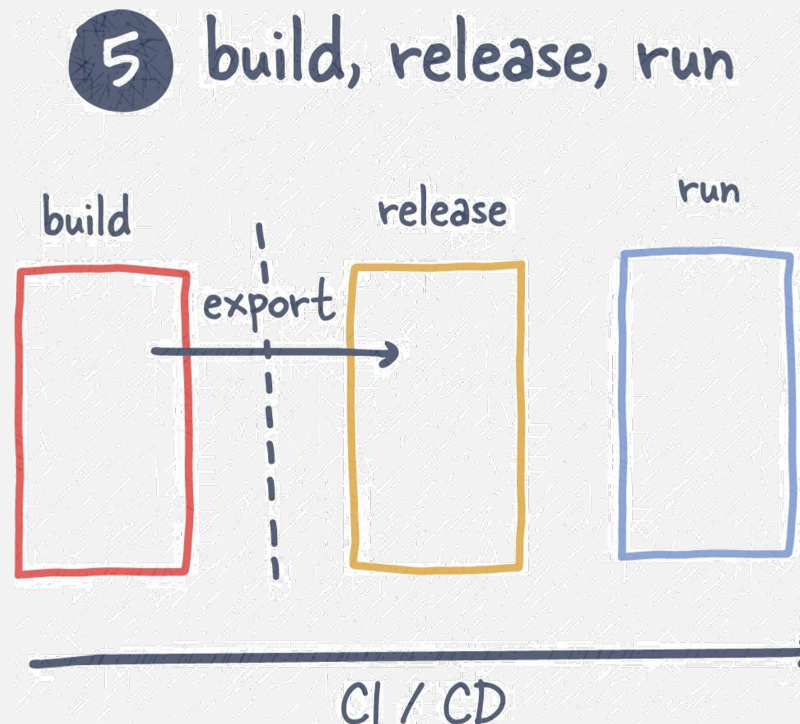


CLOUD NATIVE COMPUTING

12 FAKTOREN METHODE

V. Unterscheide Build, Release und Run Phasen

- Klare Trennung zwischen den Phasen einer Anwendung:
  - Bauen (**Build**),
  - dem Veröffentlichen (**Release**) und
  - dem Betreiben bzw. Ausführen (**Run**)
- Jede Phase hat eindeutige Aufgaben und Verantwortlichkeiten
- Container entsprechen diesem Workflow
- So wird die Verwendbarkeit von Versionen in unterschiedlichen Umgebungen gewährleistet



Beispiel: Ein Container basierend auf dem Python-Framework "FastAPI".

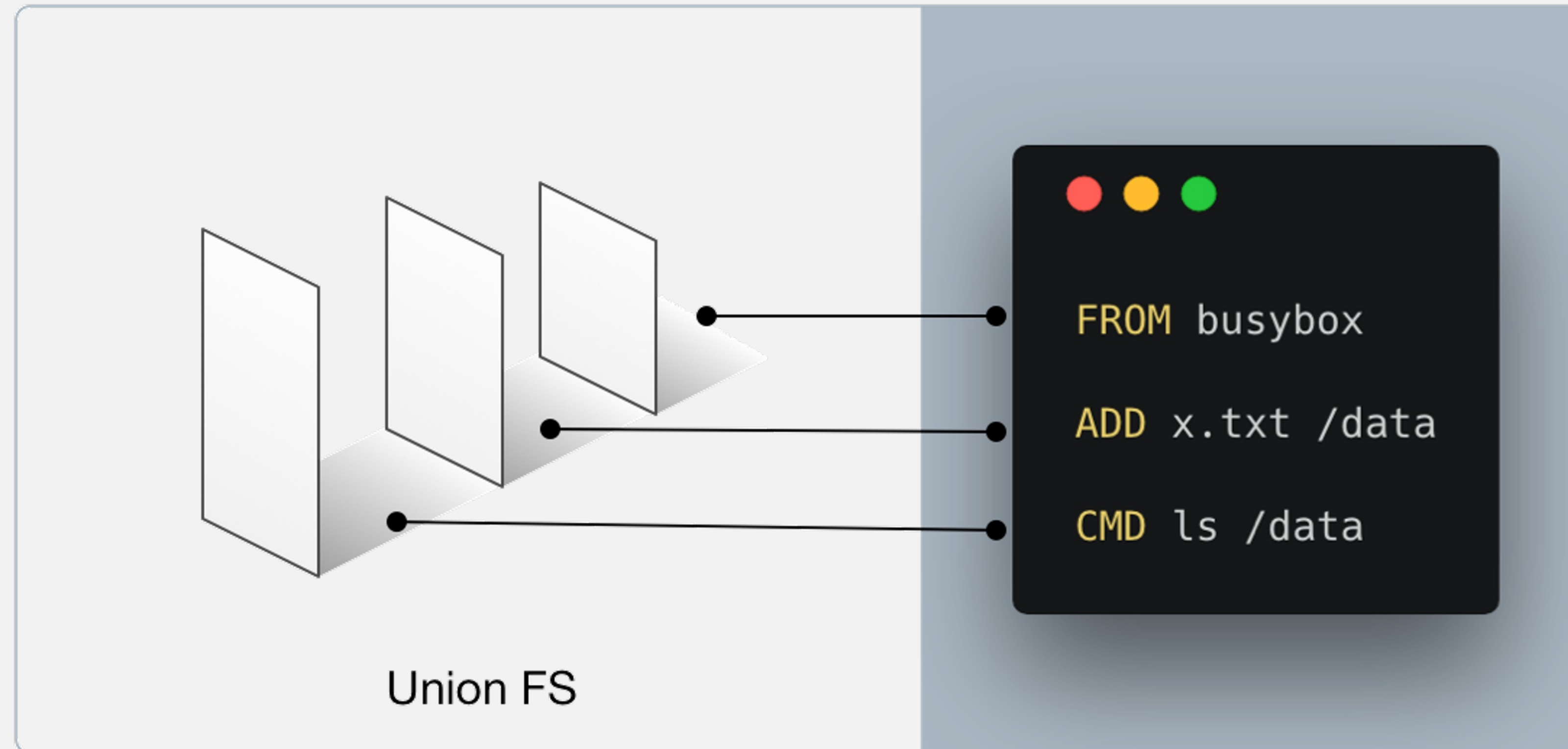
- In der **Build-Phase** wird zunächst das grundlegende **Container-Image** erstellt, inkl. aller Abhängigkeiten. Das kann z.B. mithilfe eines Dockerfiles geschehen.
- In der **Release-Phase** wird das erstellte Image in einer **Container-Registry** bereitgestellt (gepushed). Z. B. in Docker Hub oder Amazon ECR
- In der **Run-Phase** kann dann das vorher veröffentlichte Image von Systemen wie Kubernetes **ausgeführt** werden

CLOUD NATIVE COMPUTING

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

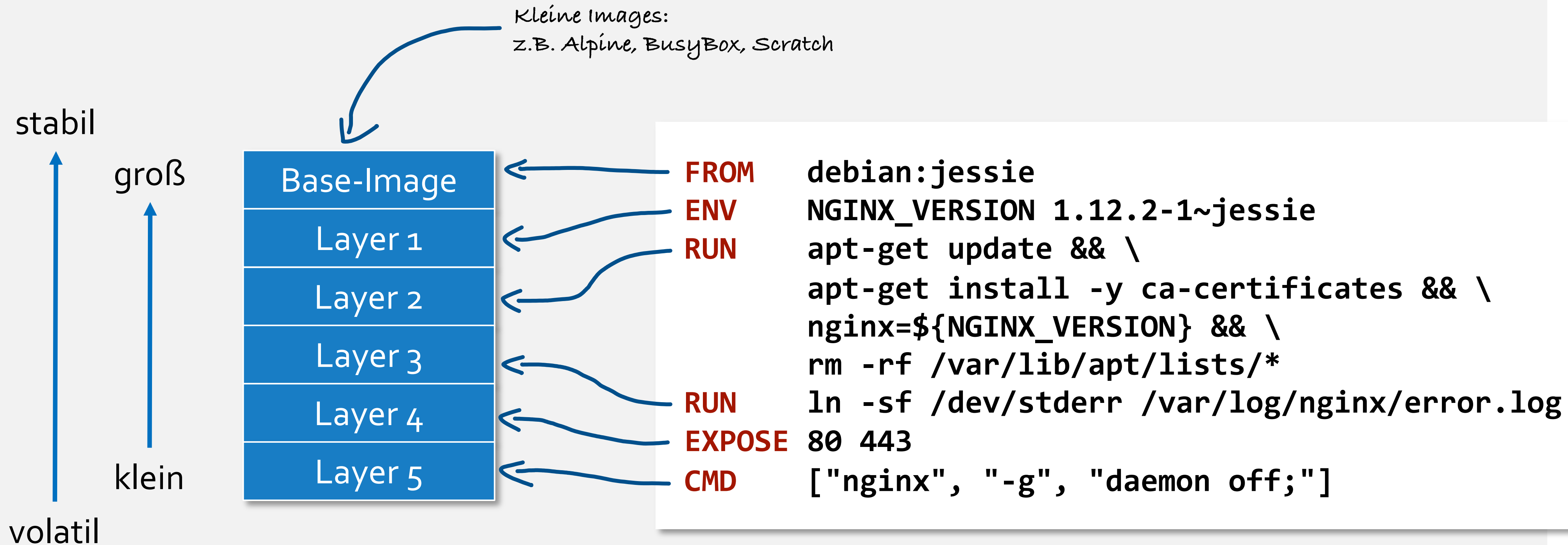
DOCKER (ANTI-)PATTERNS

Achtung: (Fast) jede Zeile im Dockerfile erzeugt einen neuen Image Layer



DOCKER PATTERN

Image-Shrinking



Diese Abfolge ermöglicht Images, die gut cachebar sind.

1. Abhängigkeiten
2. Applikation
3. Konfiguration
4. Schnittstelle

Häufige Änderungen nur an den Stellen die kleinen Beitrag zur Imagegröße haben.

RUN-Chaining:

```
RUN apk add --update wget git && \
    rm -rf /var/cache/apk/*
```

Platzschonende Installation von Paketen

```
RUN apt-get update && \
    apt-get install -y --no-install-recommends apache2 wget && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

Quelle: QAware GmbH https://github.com/qaware/cloudcomputing

DOCKER PATTERN

Multi-Stage Builds

In einem Dockerfile kann man mehrere Images definieren, die sequentiell nacheinander gebaut werden.

Dies nennt man **Multi-Stage Builds**.

Name der Stage

```
1  FROM dart:2.16-sdk AS build
2
3  ENV PATH="${PATH}:.pub-cache/bin"
4  ADD pubspec.yaml pubspec.yaml
5  ADD web/ web/
6  RUN dart pub global activate webdev && \
7      dart pub get
8  RUN webdev build --output web:build
9
10 FROM nginx:alpine
11 COPY --from=build /root/build /usr/share/nginx/html
```

Definition eines ersten Container Images

Definition eines zweiten Container Images, dass Dateien aus dem ersten Image kopiert.

Kopieren von Dateien aus einer vorherigen Stage

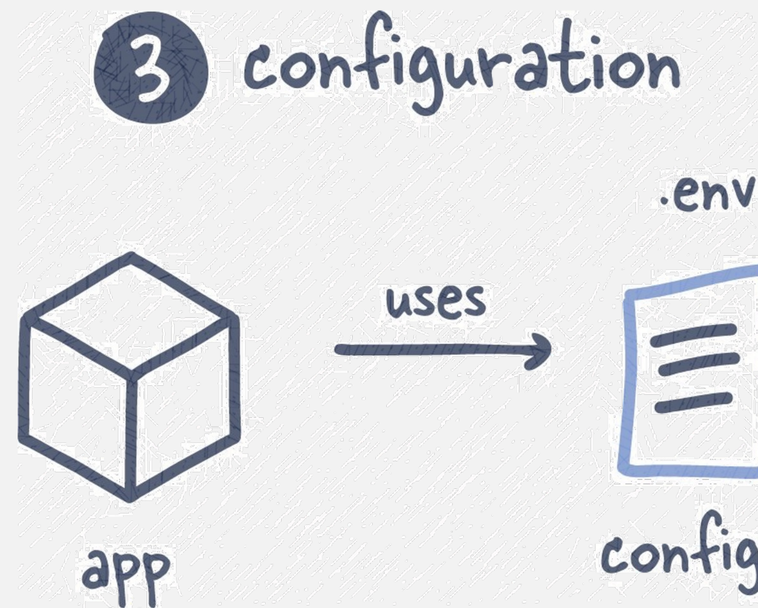
Multi-Stage Builds können genutzt, um in Pipelines das weiterreichen von Artefakten zwischen Jobs zu vermeiden/ minimieren.

12 FAKTOREN METHODE

III. Konfiguration (Konfiguriere Anwendungen mit Umgebungsvariablen)

CLOUD NATIVE COMPUTING

- Konfigurationsdaten der Anwendung sollten aus Umgebungsvariablen (env oder env vars) bezogen werden
- Ermöglicht eine bessere Portabilität und Skalierbarkeit
- Konfigurationsänderungen können so leicht durchgeführt werden, ohne dass der Code der Anwendung selbst geändert werden muss (z.B. Parsing von Konfigdateien, etc.)



Beispiel: In Kubernetes kann man hierzu **ConfigMaps** oder Secrets (für geheime Zugangsdaten, Schlüssel, etc.) verwenden. Eine ConfigMap ist ein Objekt in Kubernetes, das Konfigurationsdaten in Form von Schlüssel-Wert-Paaren speichert. Diese Konfigurationsdaten können von Containern in Pods über Umgebungsvariablen oder als Dateien in Containern verwendet werden. Auf diese Weise können **Konfigurationsdateien in Container eingeblendet** werden, ohne dass Änderungen an den Containern selbst erforderlich sind.

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

12 FAKTOREN METHODE

CLOUD NATIVE COMPUTING

III. Konfiguration

Definieren von Umgebungsvariablen in einem Dockerfile

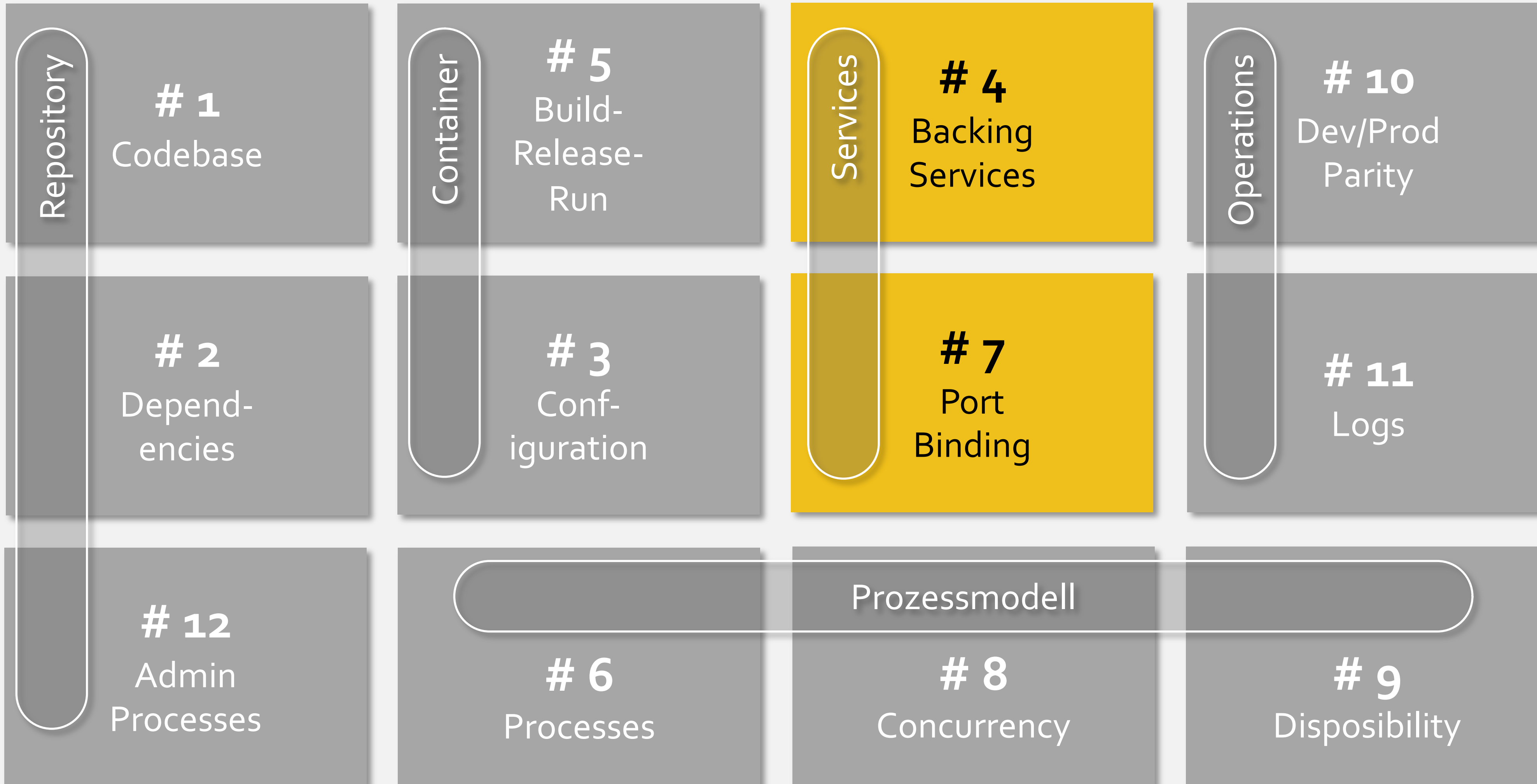
```
1   FROM dart:2.16-sdk AS build
2
3   ENV PATH="${PATH}:.pub-cache/bin"
4   ADD pubspec.yaml pubspec.yaml
5   ADD web/ web/
6   RUN dart pub global activate webdev && \
7       dart pub get
8   RUN webdev build --output web:build
9
10  FRO
11  COP
```

```
1   import wikipedia, structlog, logging, sys, os, redis, datetime, traceback
2   from tqdm import tqdm
3   from pymongo import MongoClient
4
5   BATCH = int(os.environ.get('BATCH', '50'))  # Processing n documents until pushing to mongo
6   ROUNDS = int(os.environ.get('ROUNDS', '1000')) # Restart crawler after n rounds
7   PAGES = int(os.environ.get('PAGES', '50')) # Start with n random documents the crawling
8   REDIS = os.environ.get('REDIS', 'redis')   # DNS name of Redis service
9   MONGO = os.environ.get('MONGO', 'mongo')   # DNS name of MongoDB service
10  LANG  = os.environ.get('LANGUAGE', 'de')   # Language code
11  WAIT  = int(os.environ.get('WAIT', '100')) # Wait in ms between two calls
```

Auslesen von Umgebungs-variablen in einem Prozess (hier Python)

12 FAKTOREN

Best Practices für den Betrieb von Containern

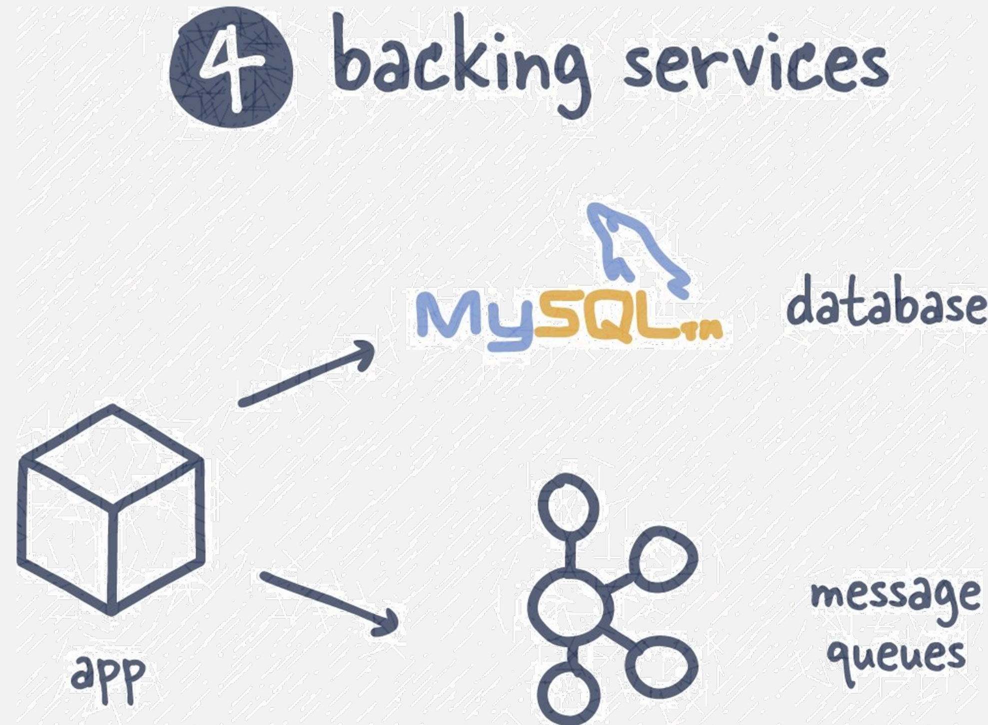


CLOUD NATIVE COMPUTING

12 FAKTOREN METHODE

IV. Unterstützende Dienste (Behandle Services als angehängte Ressourcen)

- Die Verbindung zu Diensten (Backing Services), auf die die Anwendung angewiesen ist, sollte über Umgebungsvariablen definiert werden
- Zugangsdaten (wie z.B. Datenbank-Zugangsdaten) sollten nicht direkt im Code der Anwendung oder im Container codiert sein
- Sondern über Umgebungsvariablen von bereitgestellt werden
- Vereinfacht die Konfiguration der Anwendung für verschiedene Umgebungen, wie z.B. Entwicklung, Test oder Produktion



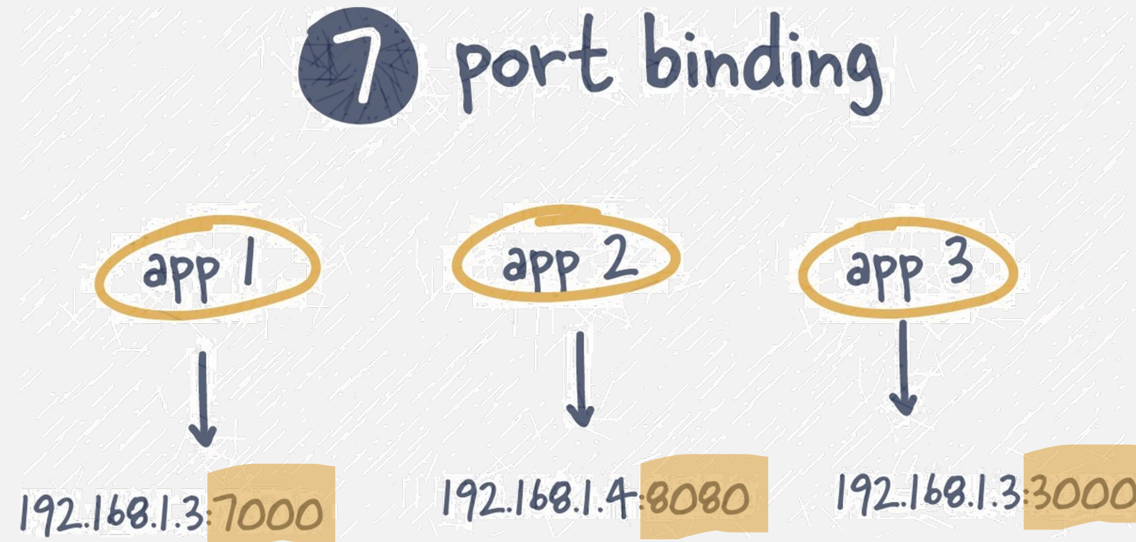
Beispiel: In Kubernetes können mittels einer Secret Ressource die geheimen Zugangsdaten für Datenbank-Verbindung definiert werden. In der Deployment-Beschreibung für die Anwendung könnten diese Zugangsdaten dann in Umgebungsvariablen definiert und an den Container innerhalb des Pods übergeben werden. So würde die Anwendung die Zugangsdaten zwar verwenden können, ohne dass diese direkt im Code enthalten oder im Container in einer Konfigdatei codiert sind.

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

12 FAKTOREN METHODE

VII. Bindung an Ports (Exportiere Services über Ports)

- Eine Anwendung sollte ihre Dienste über einen eindeutigen Port exportieren
- Die Anwendung ist dabei nicht auf die Ausführungsumgebung eines Webservers (Apache o. ähnl.) angewiesen, um einen webbasierten Dienst zu erstellen
- Die Webanwendung exportiert HTTP als Dienst, indem sie sich selber an einen Port bindet und die an diesem Port eingehenden Anfragen bearbeitet



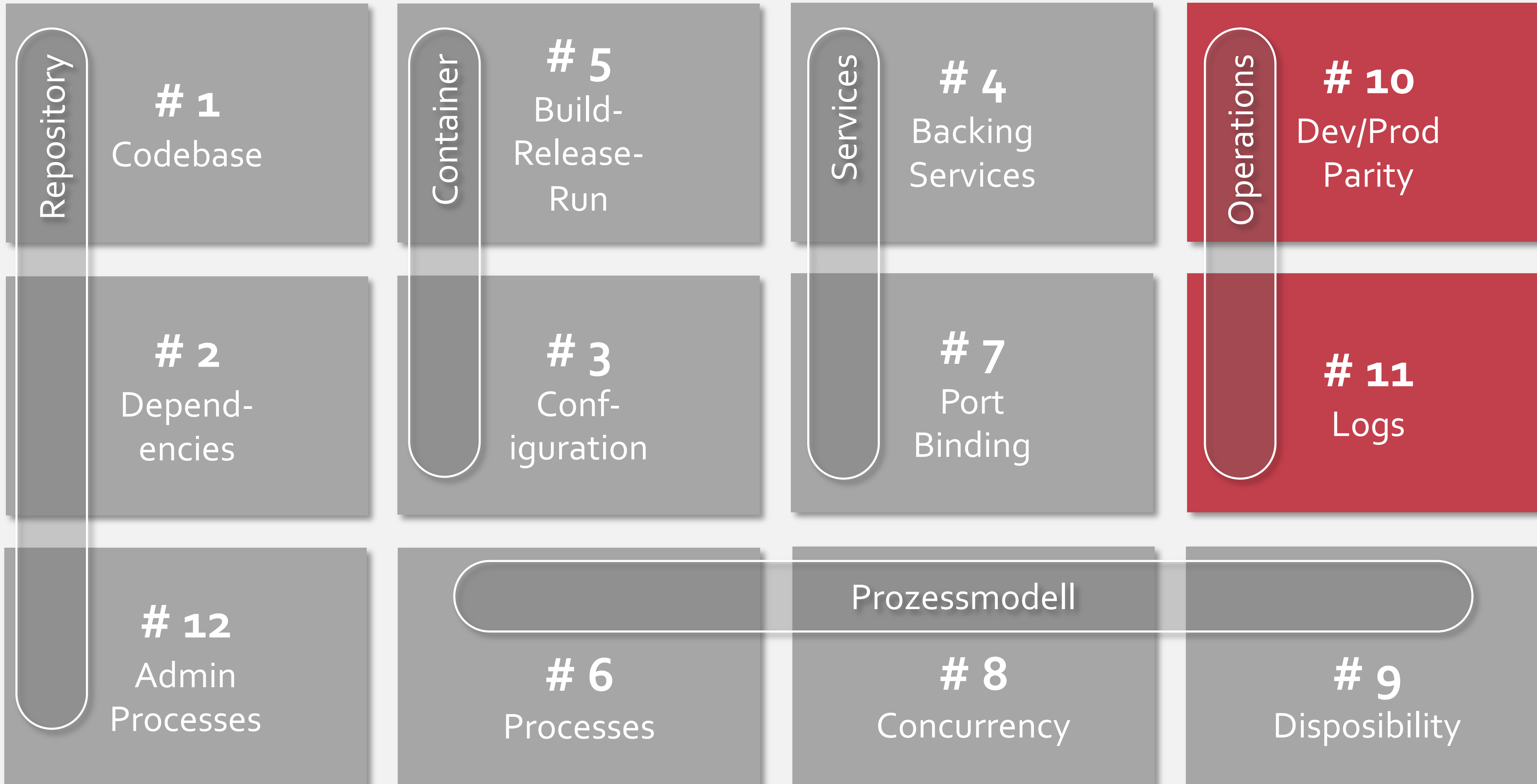
Beispiel: Eine API-Anwendung stellt ihren Service mittels einer REST-Schnittstelle bereit (z.B. mittels des Python Frameworks FastAPI), die auf einem Port lauscht, um die Dienste nach außen hin zugänglich zu machen. Ein möglicher Port wäre bspw. der Port 8080.

Hinweise:
Durch Port Binding kann eine App ein unterstützender Dienst (siehe Faktor IV) für eine andere App werden, indem die URL der unterstützenden App der konsumierenden App als Resource-Handle zur Verfügung gestellt wird.

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

12 FAKTOREN

Best Practices für den Betrieb von Containern

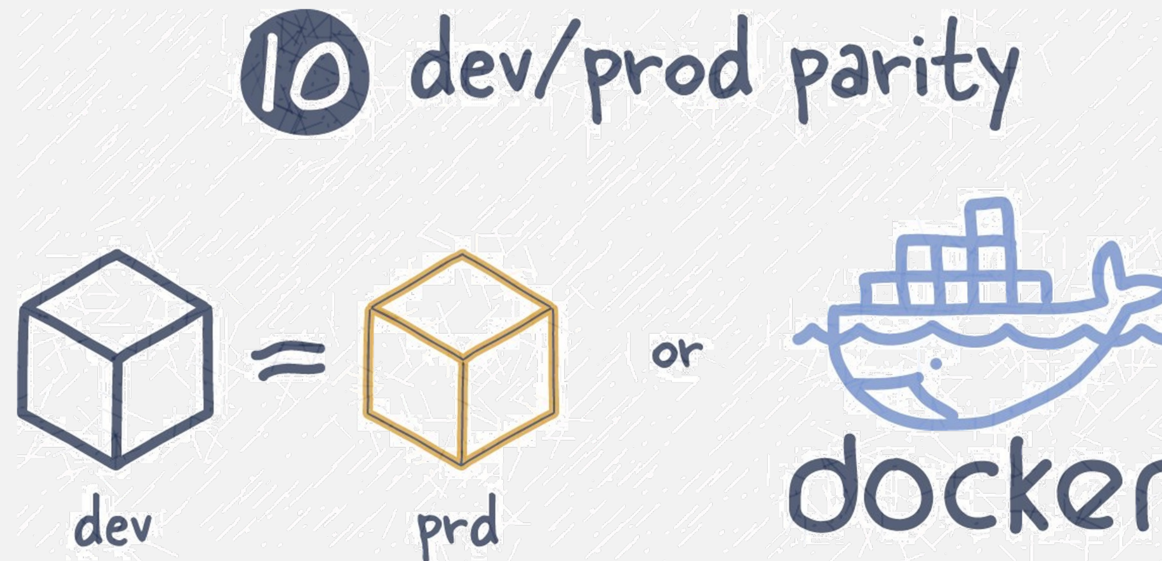


CLOUD NATIVE COMPUTING

12 FAKTOREN METHODE

X. Dev-Prod-Vergleichbarkeit (Unterscheide nicht zwischen Dev- und Produktionsumgebungen)

- Entwicklungs- und Produktionsumgebungen einer Anwendung sollten so ähnlich wie möglich sein
- Verhindert unvorhergesehene Probleme im Produktionsprozess



Beispiel: Die Verwendung von Docker-Containern zur Bereitstellung von Anwendungen entspricht diesem Prinzip. Wenn Entwickler eine Anwendung in einem Docker-Container erstellen, können diese Container einfach in die Produktionsumgebung übertragen werden, da die Laufzeitumgebung inkl. Abhängigkeiten identisch ist.

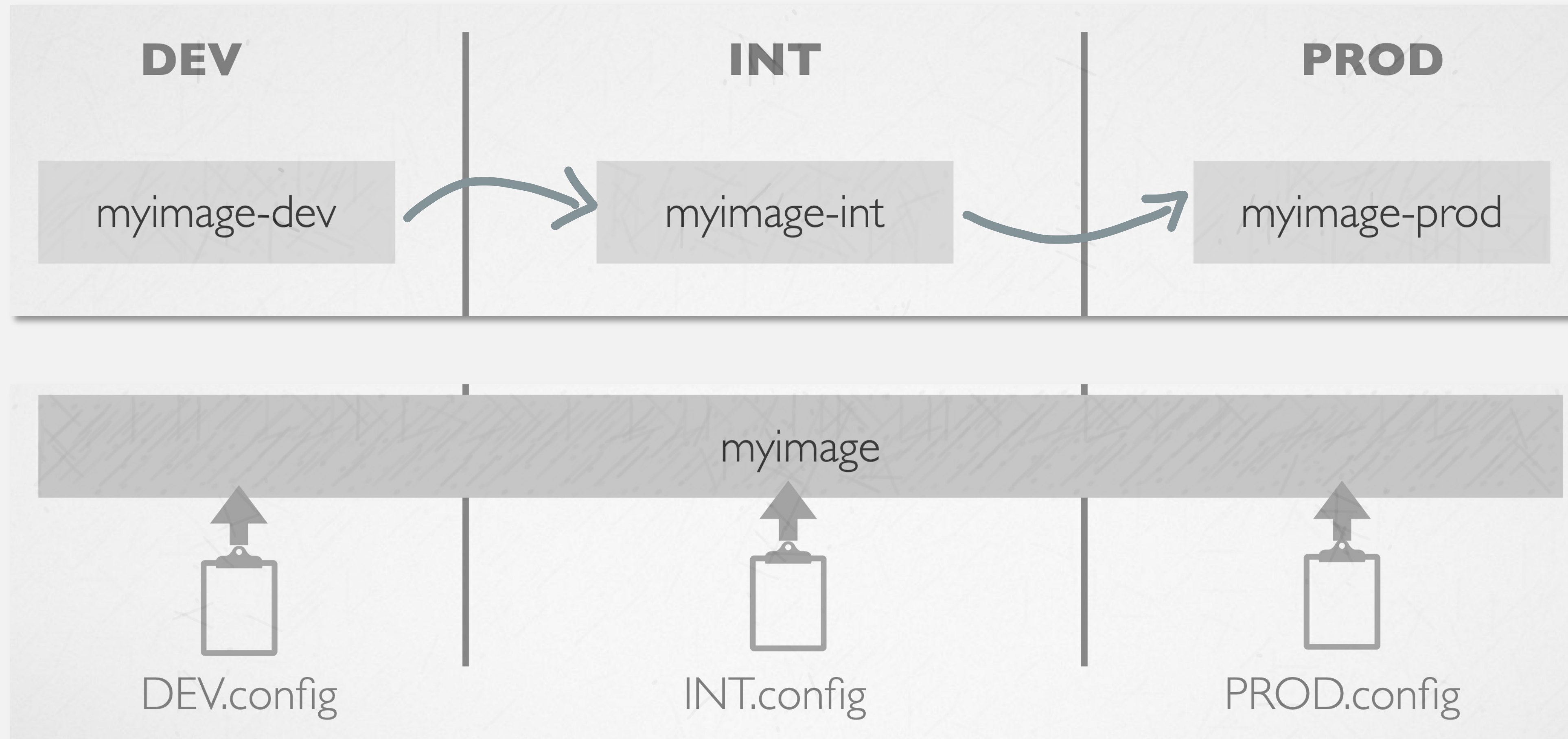
Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

CLOUD NATIVE COMPUTING

Merke:
Die Zwölf-Faktor-Entwicklung widersteht dem Drang, verschiedene unterstützende Lösungen in Entwicklung und Produktion zu verwenden.

DOCKER ANTI-PATTERN

Image Metamorphosis



So nicht!
Image wandelt sich von Umgebung zu Umgebung.

Besser so:
Ein Image für unterschiedliche Umgebungen mit der jeweiligen Umgebung angepassten Konfigurationen.

Quelle: QAware GmbH
https://github.com/qaware/cloudcomputing

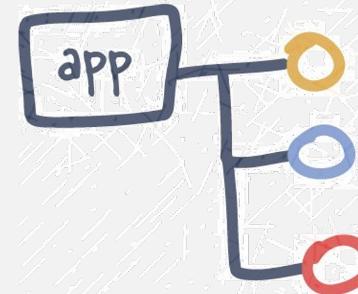
12 FAKTOREN METHODE

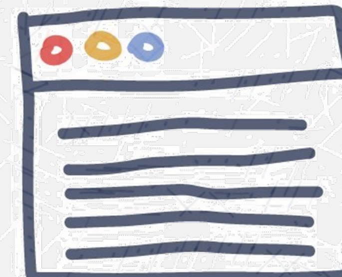
XI. Logs (Verstehe Logs als Ereignisströme)

- Ereignisse und Logmeldungen von Anwendungen sollten als strukturierte Ereignisströme behandelt werden.
- Anwendung sollten Logs in einem standardisierten, maschinenlesbaren Format ausgeben (JSON, XML, o. ähnl.) und hierzu ihre Logs einfach auf stdout schreiben (nicht in Log-Dateien)
- In Staging- oder Produktionsumgebungen können diese Datenströme der einzelnen Prozesse von der Ausführungsumgebung erfasst, zusammengeführt und zur Anzeige und langfristigen Archivierung an ein oder mehrere Endziele weitergeleitet werden

Beispiel: In Kubernetes gibt es bspw. FileBeat, welches alle Log-Dateien aller Pods erfasst, mit Metadaten der Kubernetes-Umgebung anreichert und an eine zentrale Datenbank (ElasticSearch) zu Observability-Zwecken weiterleitet. Dies funktioniert besonders dann sehr gut, wenn Anwendungen wirklich nur auf diese triviale Art loggen.

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/





CLOUD NATIVE COMPUTING

Merke:
Logs werden in Server-basierten Umgebungen üblicherweise in eine Datei auf der Platte geschrieben (eine Logdatei) - das ist aber nur ein Ausgabeformat und man ist nicht daran gebunden.

12-Faktor-Apps schreiben den Stream von Ereignissen ungepuffert auf stdout damit Logs einer Logkonsolidierung möglichst unkompliziert unterworfen werden können.

12 FAKTOREN METHODE

XI. Logs

```
print("Ich bin ein Log-Entry in Python")
```

Sehr einfaches Logging (log to stdout), kein Einsatz spezifischer Logging-Libraries (reicht schon aus, um Logs konsolidieren zu können)

```
import json
print(json.dumps({
        "event": "Log-Entry",
        "message": "Ich bin ein Log-Entry",
        "language": "Python"
}))
```

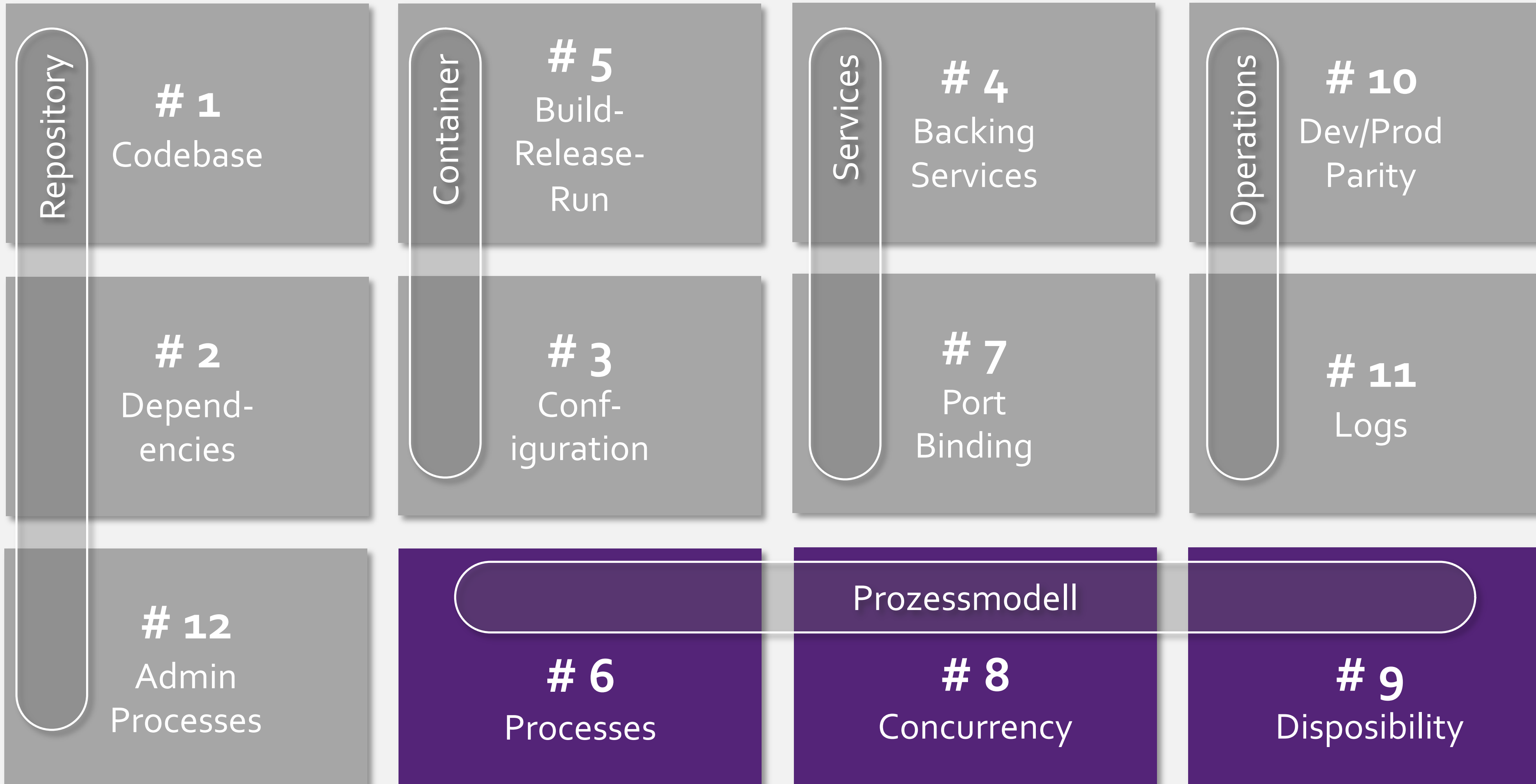
Sehr einfaches strukturiertes Logging (log JSON to stdout), ebenfalls ohne Einsatz spezifischer Logging-Libraries (reicht schon aus, um Logs konsolidieren und zielgerichteter auswerten zu können in einer Log-Konsolidierung)

Merke: Komplizierter muss es nicht unbedingt sein.

CLOUD NATIVE COMPUTING

12 FAKTOREN

Best Practices für den Betrieb von Containern

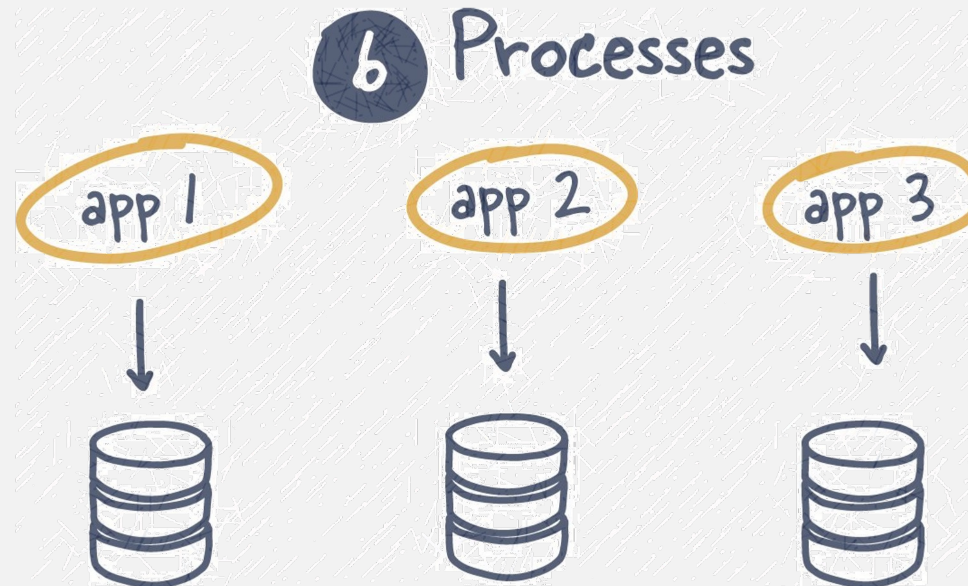


CLOUD NATIVE COMPUTING

12 FAKTOREN METHODE

VI. Prozesse (Konzipiere Anwendungsprozesse stateless)

- Anwendungsprozesse sind stateless (zustandslos) auszuführen
- Sie dürfen keine Daten in der **lokalen Umgebung** speichern
- Anwendungsprozesse haben keine lokal gespeicherten Informationen über den aktuellen Zustand
- Alle erforderlichen Informationen über einen zentralen Dienst wie eine Datenbank abrufen
- Durch Kopien des Anwendungsprozesses, kann so horizontal skaliert werden, um Lastspitzen zu bewältigen



Beispiel: Eine Webanwendung, die Inhalte veröffentlichen und anzeigen kann. Die Anwendung speichert alle Daten in einer Datenbank, die jeder Anwendungsprozess abrufen und manipulieren kann. Die Anwendungsprozesse haben keinen eigenen Zustand und die Skalierung erfolgt horizontal, indem Kopien des Anwendungsprozesses erstellt werden, die auf dieselbe Datenbank zugreifen. Dadurch können verschiedene Instanzen des Anwendungsprozesses problemlos zusammenarbeiten, da diese sich über die gemeinsame Datenbank abstimmen können.

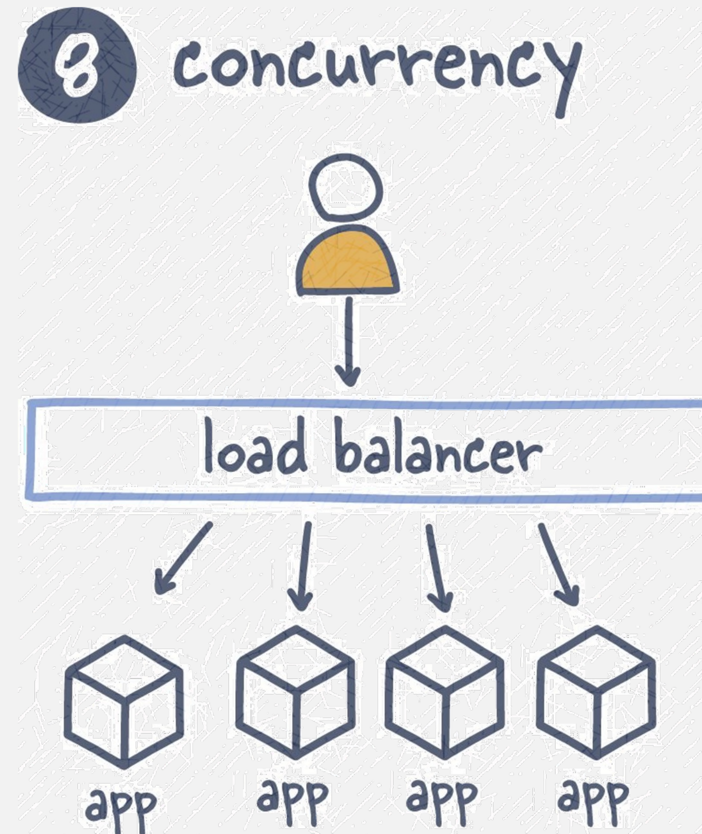
Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

Merke:
12 Factor Apps werden für horizontale Skalierbarkeit entworfen.

12 FAKTOREN METHODE

VIII. Nebenläufigkeit (Skaliere durch zusätzliche Prozessinstanzen)

- Eine Anwendung sollte horizontal und nicht vertikal skaliert werden
- Bei steigenden Lasten sollten zusätzliche Prozesse gestartet werden, um die Verarbeitung über mehrere Instanzen zu verteilen
- Hierdurch wird auch die Verfügbarkeit der Applikation erhöht, da ein Ausfall einer Prozessinstanz nicht das Gesamtsystem lahmlegt



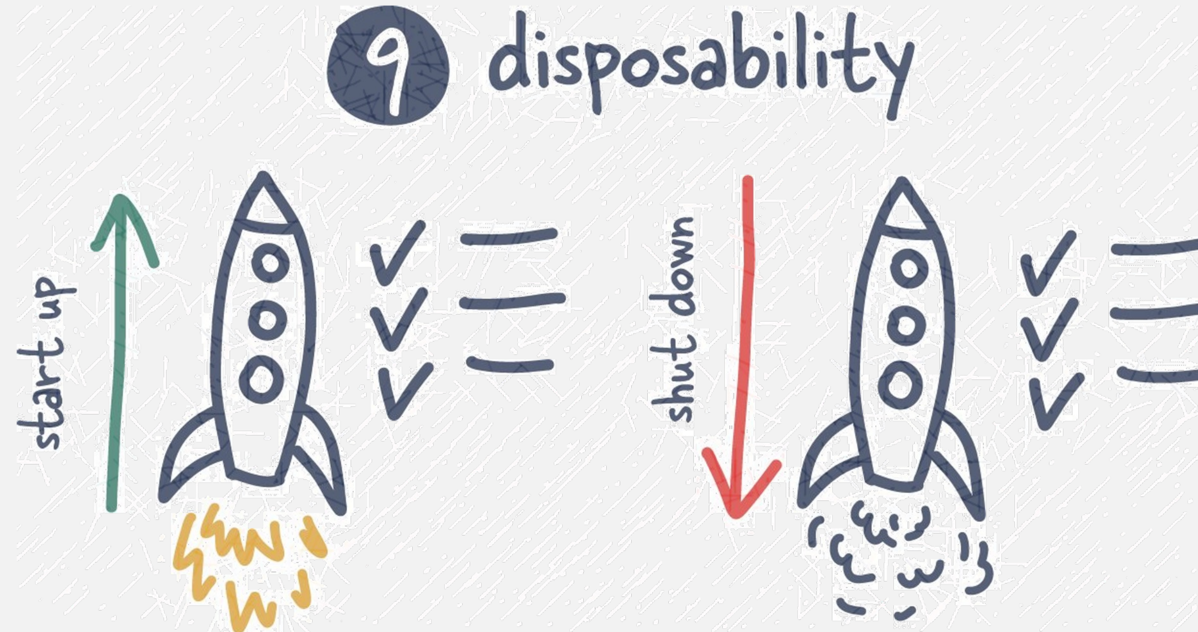
Beispiel: Ein Beispiel ist der Horizontal Pod Autoscaler (HPA) von Kubernetes. Ein HPA ist ein Kubernetes-Betriebsmittel, das Pod-Deployments automatisch in Bezug auf CPU-Auslastung oder andere Metriken skalieren kann. Ein HPA skaliert horizontal, indem er bei steigenden Lasten einfach weitere Pods hinzufügt, statt die Ressourcen pro Pod zu erhöhen und bei fallenden Lasten, Pods einfach ausschaltet.

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

12 FAKTOREN METHODE

IX. Disposibility (Achte auf schnell startende/stoppende Prozesse)

- Anwendungsprozesse sollten schnell start- und stoppbar sein (nicht Minuten dafür benötigen)
- Jeder Prozess sollte in einer isolierten Umgebung ausgeführt werden
- Dies ermöglicht es, die Anwendung schnell und robust zu skalieren, indem einfach weitere Instanzen der Anwendung gestartet werden, um den Traffic zu bewältigen



Beispiel: Ein Beispiel ist das "Rolling Update" in Kubernetes. Wenn eine neue Version einer Anwendung bereitgestellt werden soll, startet Kubernetes allmählich neue Instanzen und stoppt die alten Instanzen nach und nach, um sicherzustellen, dass der Service für die Benutzer stets verfügbar bleibt. Diese Methode ermöglicht eine schnelle Aktualisierung der Anwendung ohne merkliche Ausfälle für die Benutzer, funktioniert aber nur, wenn die Prozesse auch schnell gestartet und gestoppt werden können (und nicht Minuten dafür benötigen).

Bildquelle: https://architecturenotes.co/12-factor-app-revisited/

CLOUD NATIVE COMPUTING

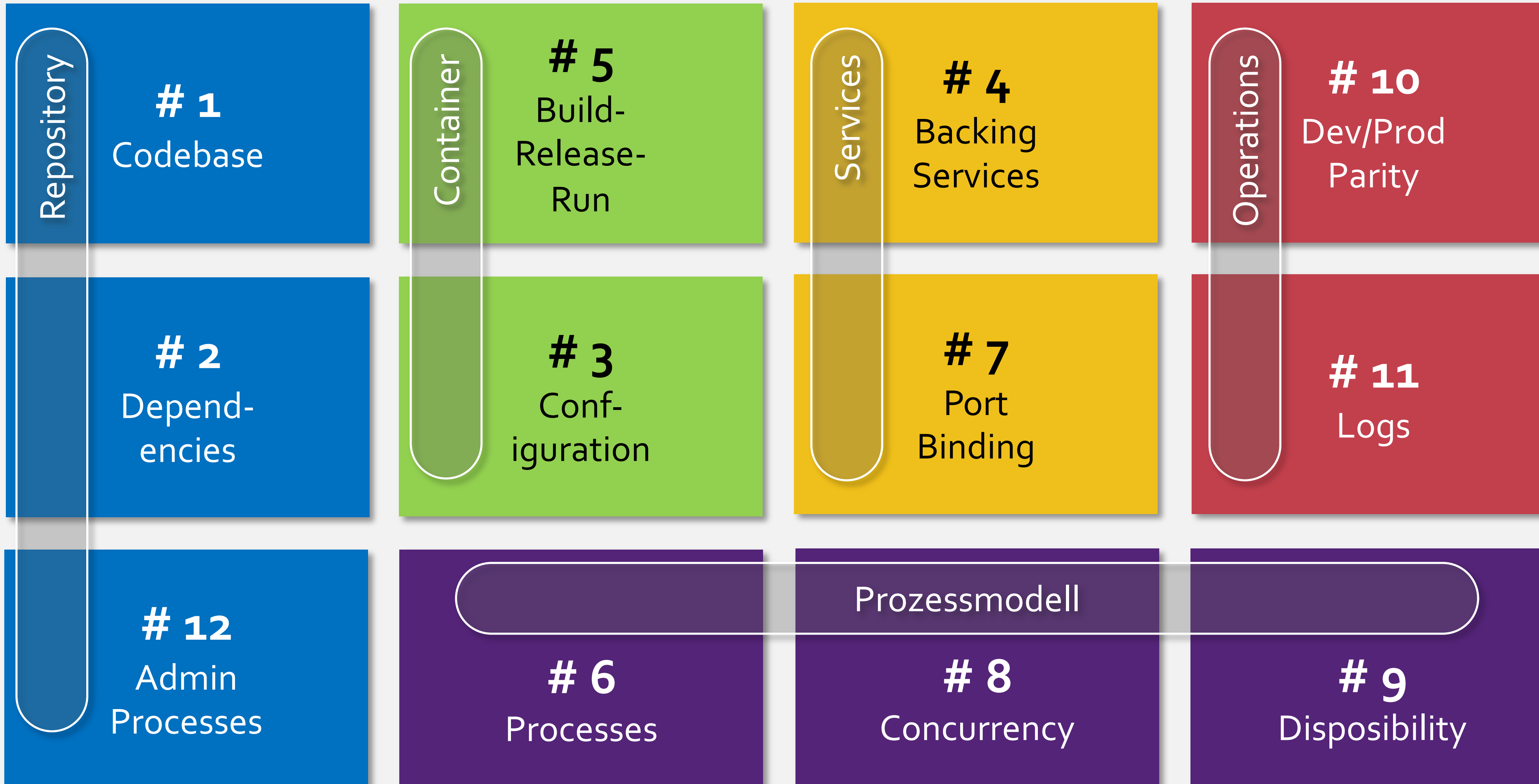
Hinweise:
Schnelle Starts und problemlose Stopps optimieren die Robustheit.

Dieser Faktor erleichtert schnelles elastisches Skalieren, schnelles Deployment von Code oder Konfigurations-änderungen und macht Produktionsdeployments robuster.

MERKE:
Anything fails all the time!

12 FAKTOREN

Best Practices für den Betrieb von Containern

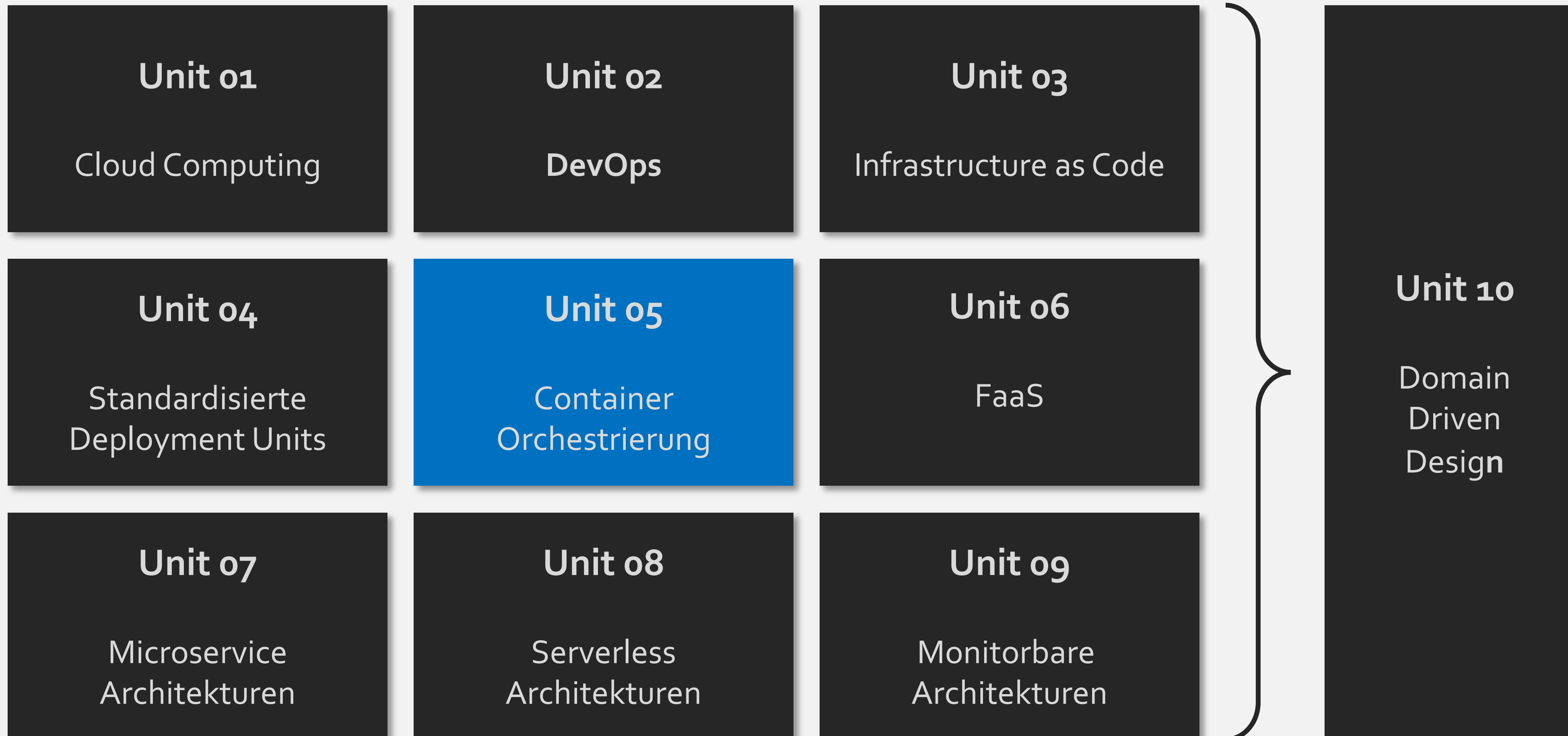


CLOUD NATIVE COMPUTING

Die 12-Faktoren-Methodik ist eine Reihe von Best Practices für die Entwicklung von Cloud-nativen Anwendungen, die portabel, skalierbar, robust, einfach zu betreiben und skalierbar sind. Die Faktoren erleichtern die Bereitstellung und den Betrieb in verschiedenen Umgebungen wie Entwicklungs-,Test- und Produktivsystemen in Private oder Public Cloud-Infrastrukturen oder Container-Plattformen wie Kubernetes.

AUSBLICK

Überblick über Units und Themen dieses Moduls



CLOUD NATIVE COMPUTING

KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de