CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

DevOps

(3) Continuous X und Deployment Pipelines

PROF. DR.
NANE KRATZKE

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 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 CCo 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.

KAPITEL 3 + 6

Cloud Computing und Cloud-native

3 DevOps

3.1 Prinzipien des Flow
Arbeit sichtbar machen, Work in Progress beschränken,Flaschenhälse minimieren

3.2 Prinzipien des Feedbacks
Probleme früh erkennen, Probleme sofort lösen, Probleme professionell verantworten

3.3 DevOps-geeignete Architekturen
Randbedingungen für die Entwicklung, Nutzung von Orchesterierungsplattformen, Randbedingungen für den Betrieb

6 Deployment-Pipelines

6.1 Deployment-Pipelines as Code
Phasen-, Gerichtete, Hierarische Pipelines, Steuerung von Pipelines

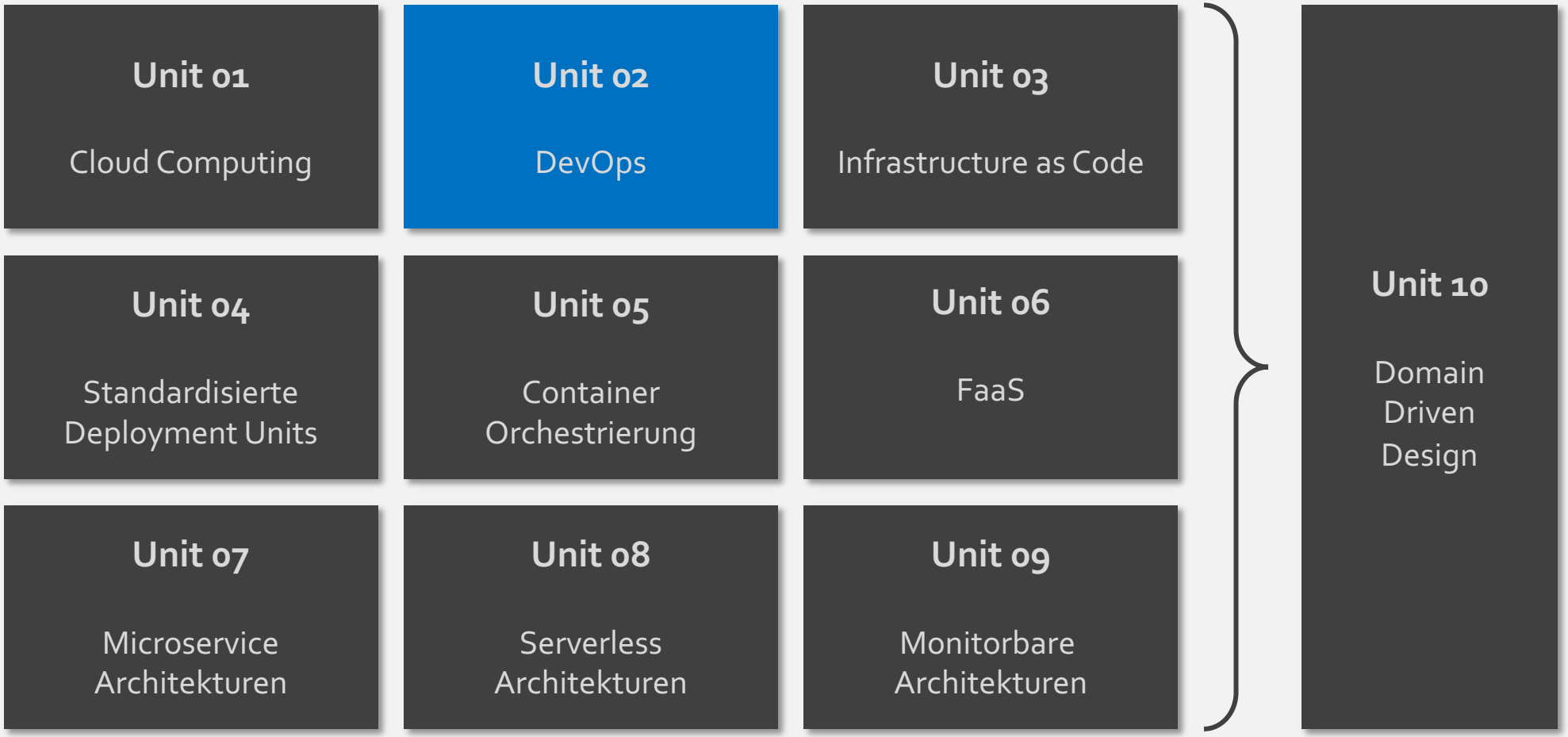
6.2 DevOps-geeignete Branching Strategien
Git-Flow, GitHub-Flow, Trunk-basierte Entwicklung

6.3 Zusammenfassung

CLOUD NATIVE COMPUTING

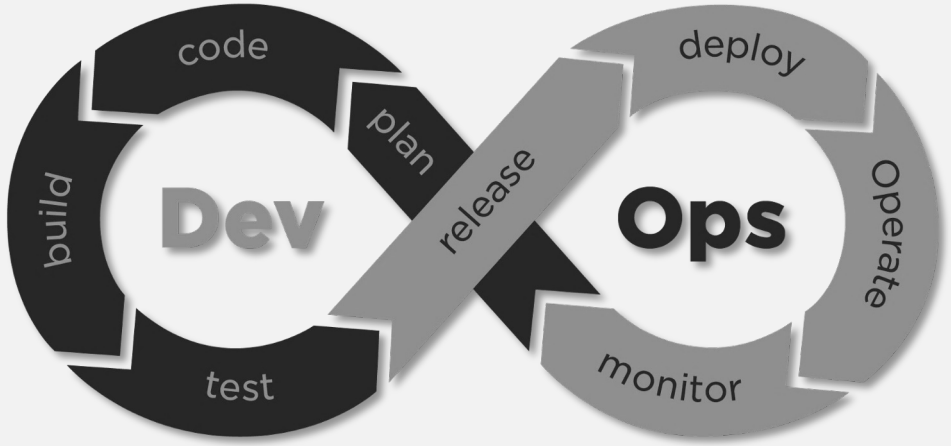
INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



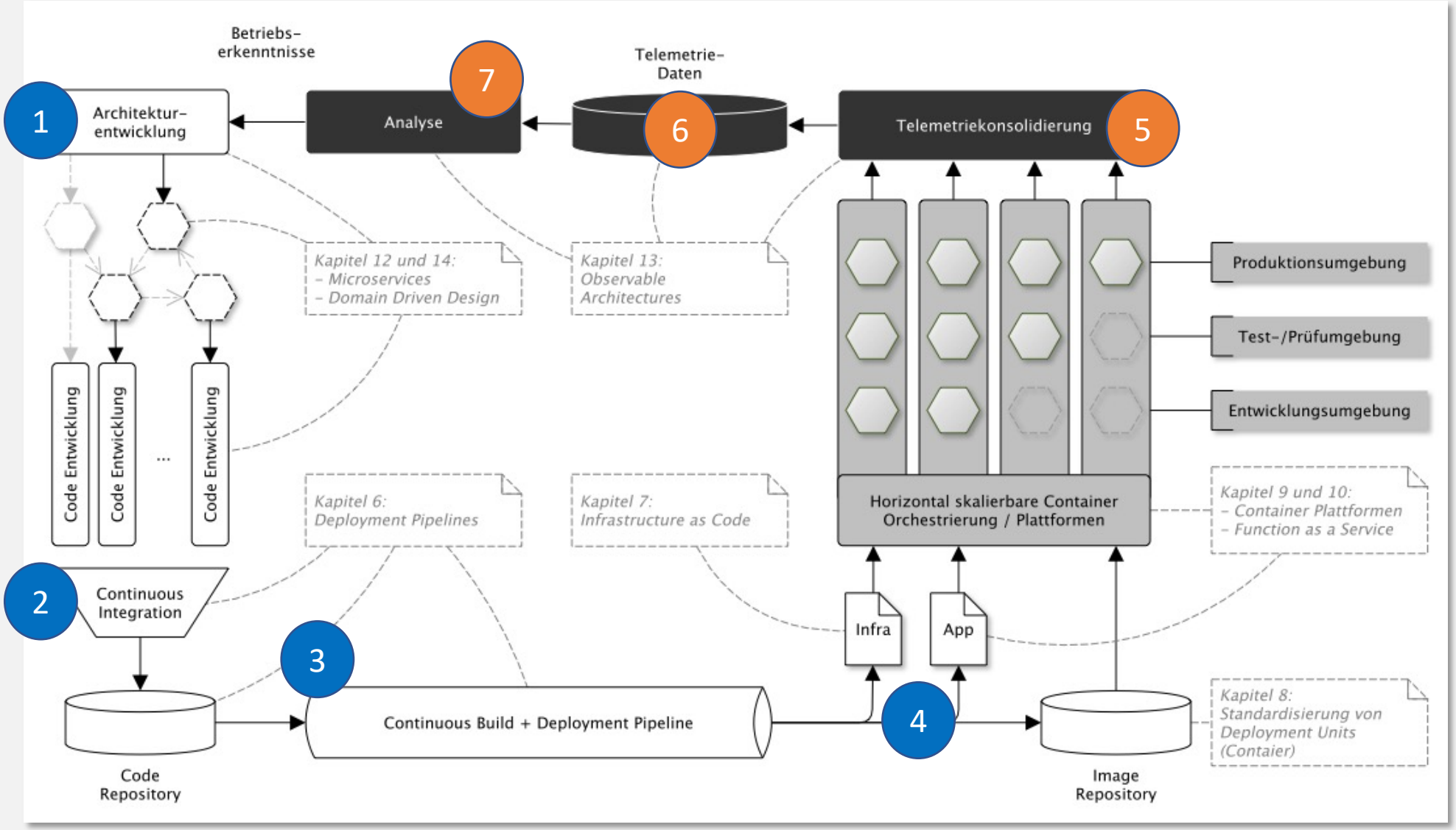
INHALTE

- DevOps Prinzipien
- DevOps-Cycle konforme Architekturen und Umgebungen
- **Continuous Begriffe und Deployment Pipelines**



DEVOPS CYCLE

konforme Architekturen und Infrastrukturen



Prinzipien des Flow

Prinzipien des Feedbacks

verknüpft über automatisierbare technische Plattform

CONTINUOUS X

Vorsicht vorm Begriffswirrwarr

CLOUD NATIVE COMPUTING

Continuous Integration (CI)

- Alle Änderungen werden sofort in den aktuellen Entwicklungsstand **integriert** und **getestet**.
- Dadurch wird kontinuierlich getestet, ob eine **Änderung inkompatibel** mit anderen Änderungen ist.

Continuous Delivery (CD)

- Der **Code kann jederzeit deployed** werden.
- Der Code muss allerdings nicht immer deployed werden.
- Der Code muss (möglichst) zu jedem Zeitpunkt gebaut, getestet und debuggt werden können.

Continuous Deployment

- Jede **stabile Änderung** wird **in Produktion** deployed.
- Ein Teil der Qualitätstests findet dadurch in Produktion statt.
- Die Möglichkeit mit Fehlern umzugehen muss also vorhanden sein und vom DevOps Team beherrscht werden (z.B. Canary Releases).

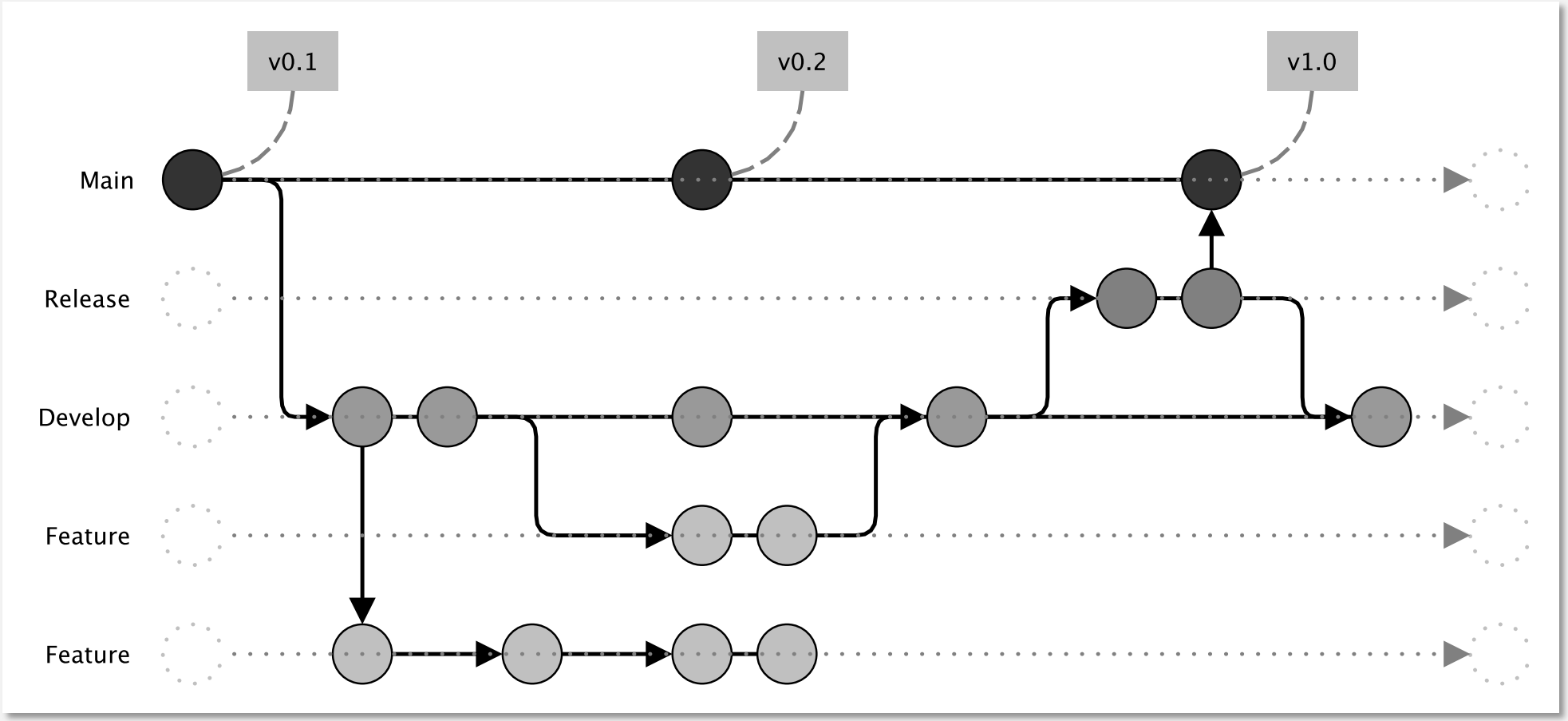
Achtung:

Diese Begrifflichkeiten gehen gerne mal durch einander.

Vor allem Continuous Delivery und Deployment (können gleich abgekürzt werden).

CONTINUOUS INTEGRATION

Gitflow



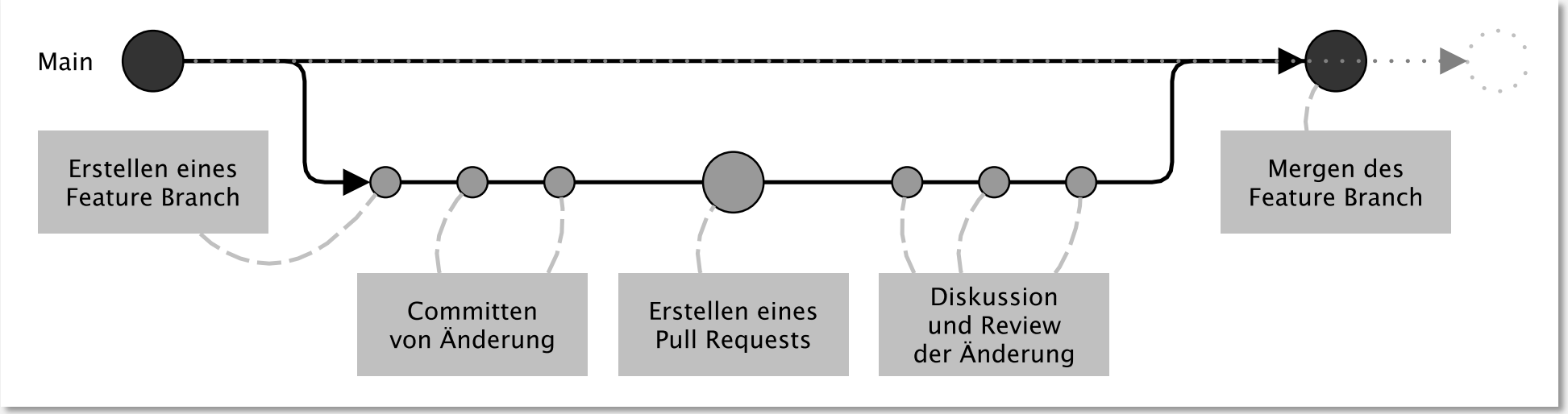
CLOUD NATIVE COMPUTING

Anmerkung:

Der Klassiker, geeignet für größere und komplexere Projekte. Allerdings besteht die Gefahr das recht viele Branches gebildet werden, in denen man sich als Team dann verliert.

CONTINUOUS INTEGRATION

GitHub Flow



GitHub Flow fördert die kontinuierliche Integration und automatisierte Deployments durch die regelmäßige, inkrementelle Entwicklung und Veröffentlichung von Features, wodurch das Risiko von Integrationsproblemen reduziert und die Produktqualität verbessert wird.

CLOUD NATIVE COMPUTING

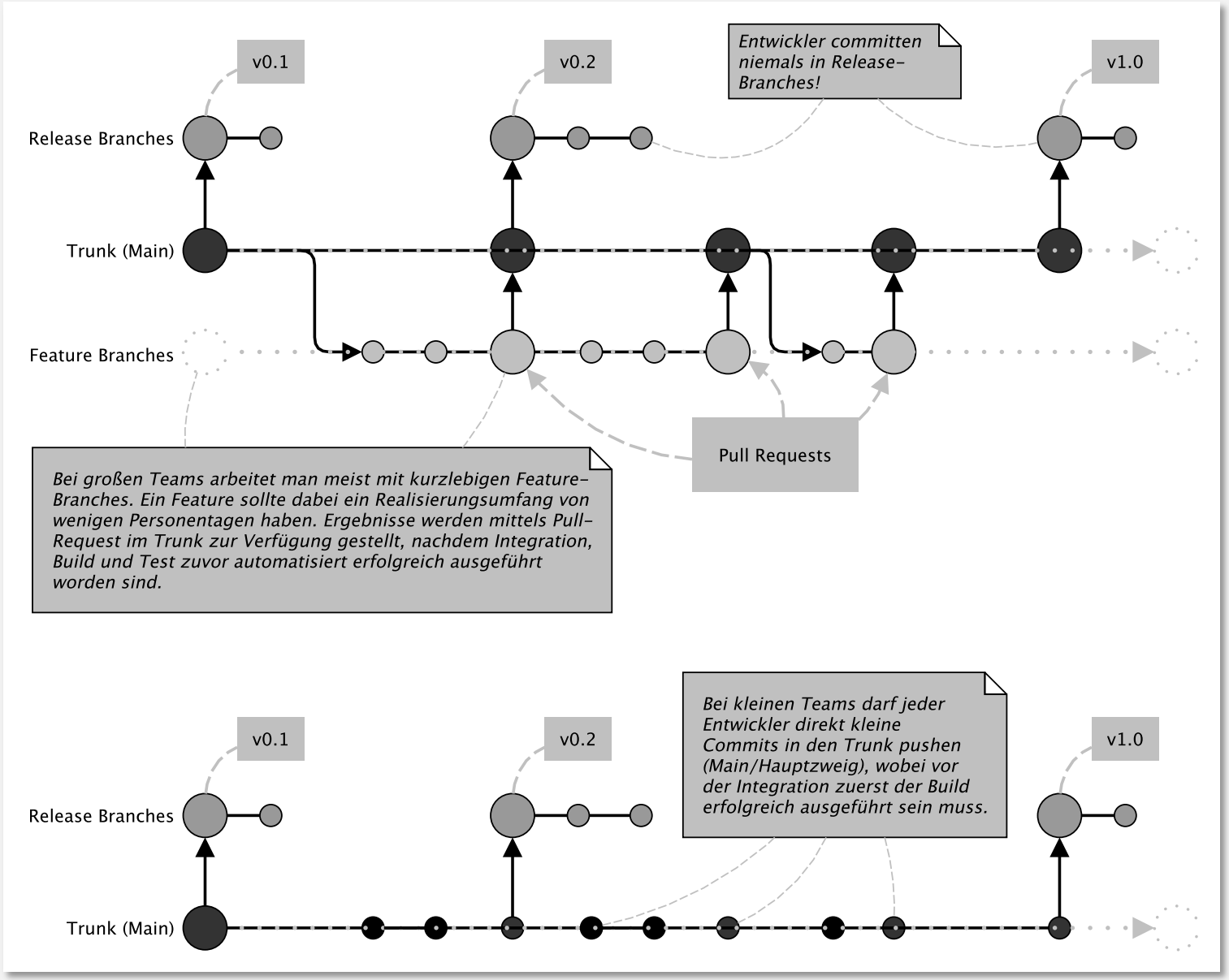
Anmerkung:

Der Einfache, geeignet für kleinere (Web-)Projekte. Da nur ein Branch existiert, für komplexere Projekte oft zu einfach, da keine „Vor-/Beta-Versionen" gehandhabt werden können.

CONTINUOUS INTEGRATION

Trunk-basierte Entwicklung

Pragmatische Mischung aus Einfachheit und „Vor-/Beta-Versions-handling“



CLOUD NATIVE COMPUTING

Merke:

Releases werden aus speziellen Branches gefahren, ansonsten Ähnlichkeiten mit dem sehr einfachen Github Flow .

Der Main-Trunk ist quasi die „Vor-/Beta-Version“.

CONTINUOUS X

Vorsicht vorm Begriffswirrwarr

CLOUD NATIVE COMPUTING

Continuous Integration (CI)

- Alle Änderungen werden sofort in den aktuellen Entwicklungsstand **integriert** und **getestet**.
- Dadurch wird kontinuierlich getestet, ob eine **Änderung inkompatibel** mit anderen Änderungen ist.

Continuous Delivery (CD)

- Der **Code kann jederzeit deployed** werden.
- Der Code muss allerdings nicht immer deployed werden.
- Der Code muss (möglichst) zu jedem Zeitpunkt gebaut, getestet und debuggt werden können.

Continuous Deployment

- Jede **stabile Änderung wird in Produktion deployed**.
- Ein Teil der Qualitätstests findet dadurch in Produktion statt.
- Die Möglichkeit mit Fehlern umzugehen muss also vorhanden sein und vom DevOps Team beherrscht werden (z.B. Canary Releases).

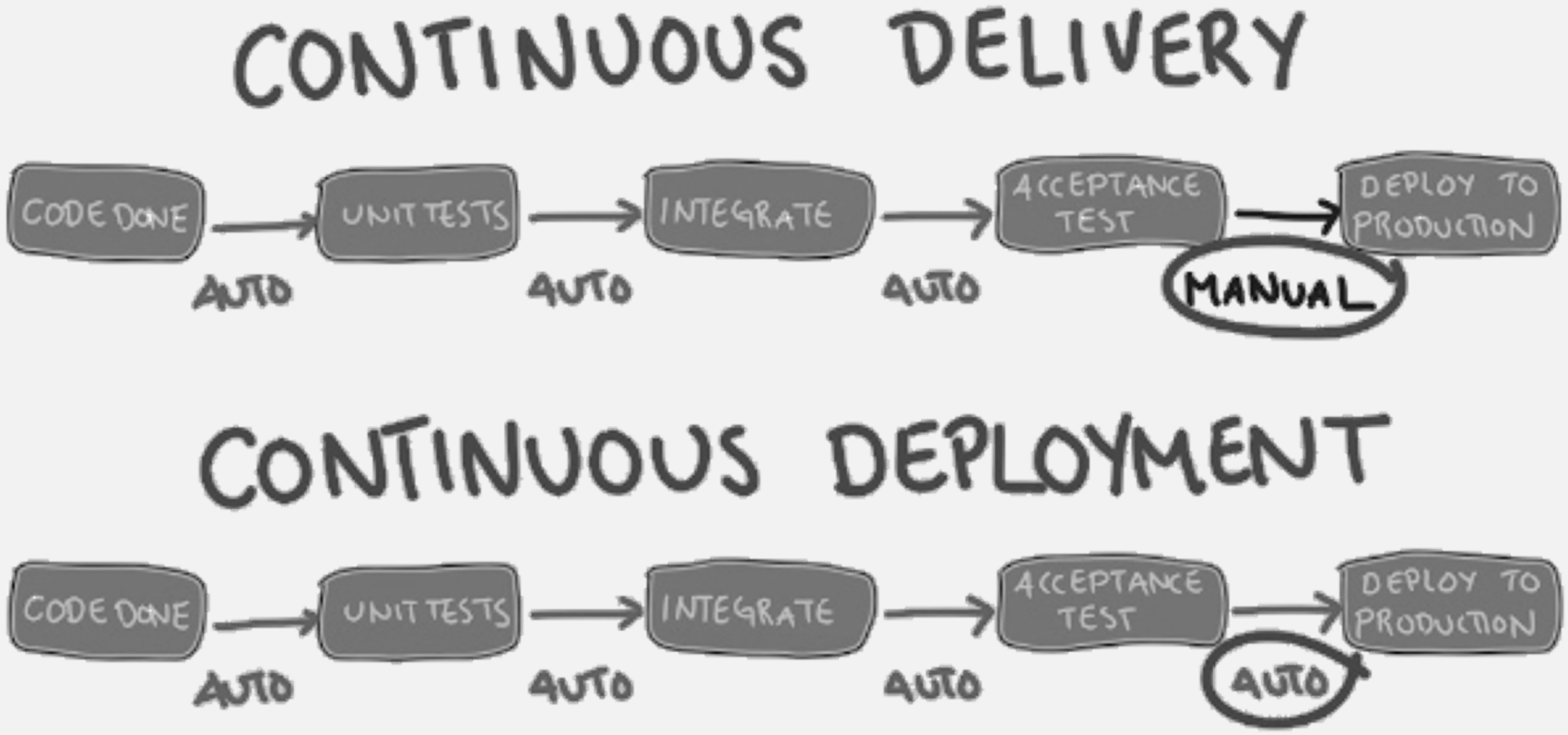
Achtung:

Diese Begrifflichkeiten gehen gerne mal durch einander.

Vor allem Continuous Delivery und Deployment (können gleich abgekürzt werden).

CONTINUOUS X

Vorsicht vorm Begriffswirrwarr

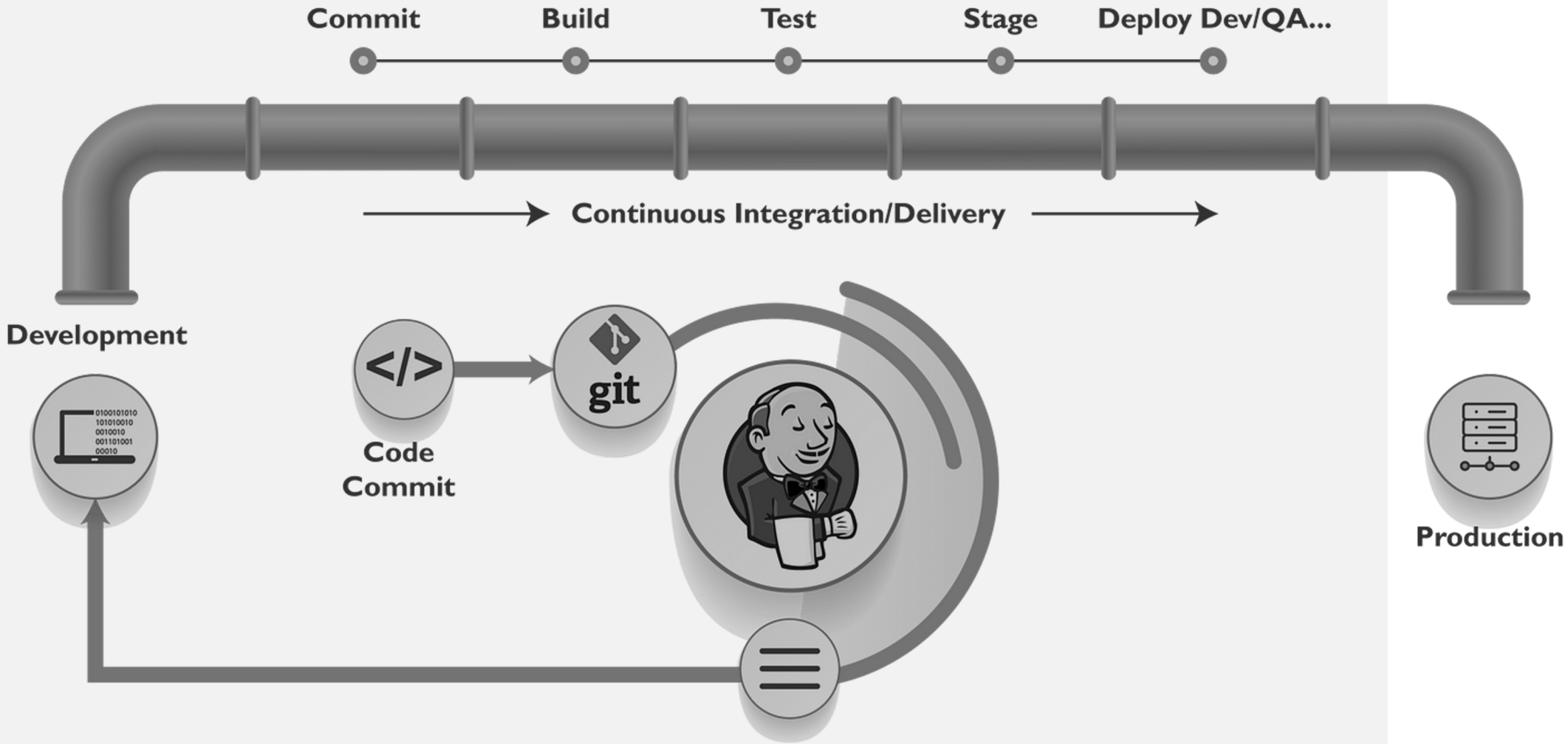


Achtung:

Der wesentliche Unterschied, ist also das beim Continuous Deployment der Deploy in Production **automatisiert** erfolgt!

DEPLOYMENT PIPELINE

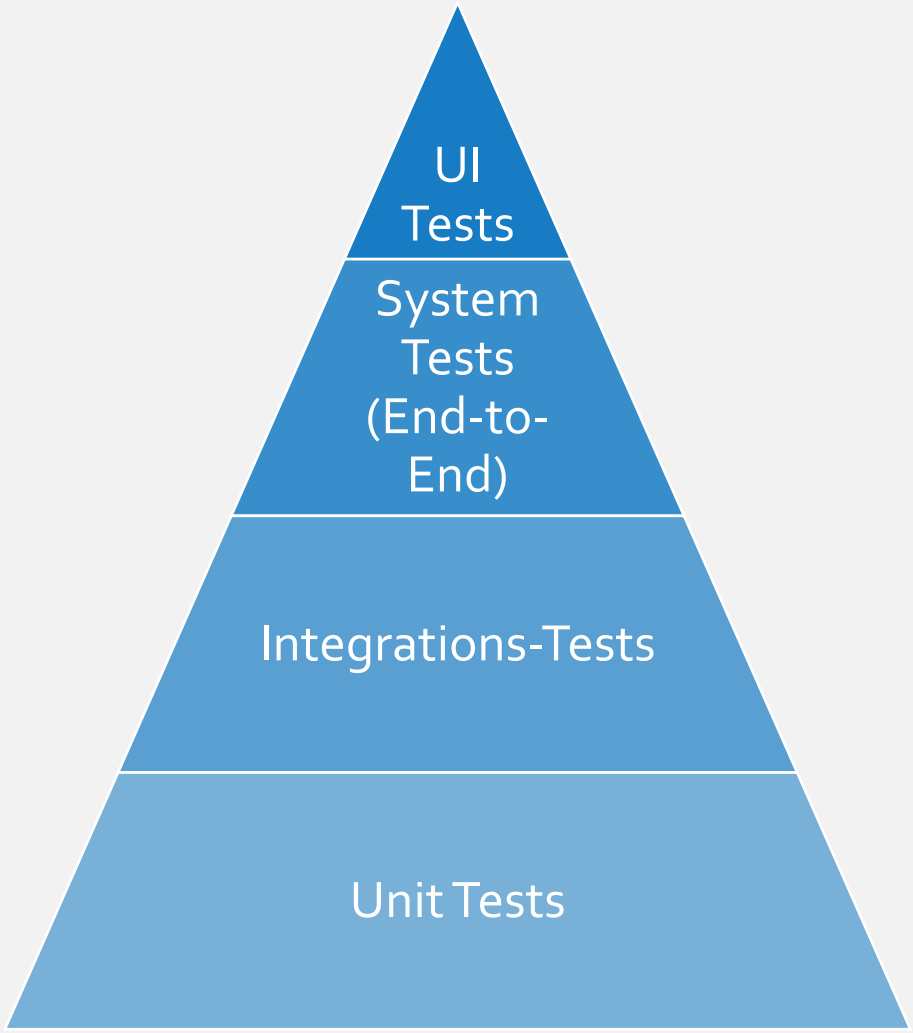
Funktionsprinzip



TEST-PYRAMIDE

Sofortiges Feedback bei Fehlern

- **Ggf. UI-Tests:** Manchmal wird eine zusätzliche Ebene für UI-Tests oder Akzeptanztests dargestellt. Diese Tests konzentrieren sich auf das Benutzerinterface und die Benutzererfahrung.
- **System-Tests:** Überprüfen das gesamte System von Ende zu Ende, oft unter Einbeziehung der Benutzeroberfläche.
- **Integration Tests:** Überprüfen die Interaktionen zwischen verschiedenen Komponenten oder Systemen.
- **Unit Tests:** Testen einzelner Komponenten oder Funktionen isoliert vom Rest des Systems.



CLOUD NATIVE COMPUTING

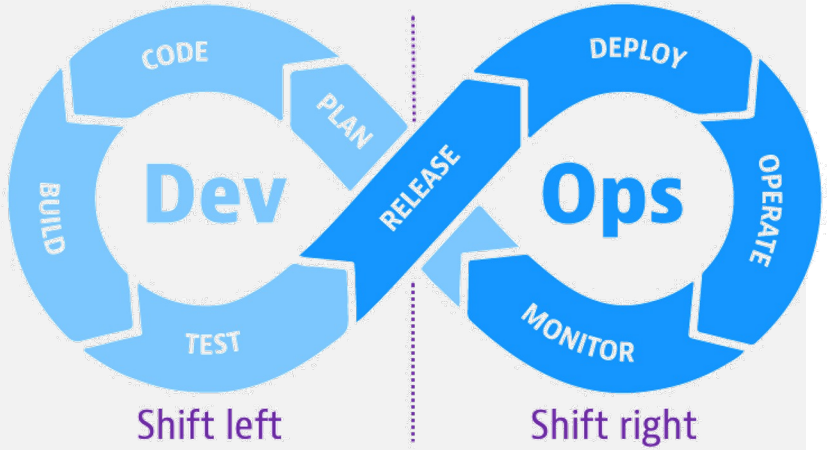
Alle Tests sollten so oft wie möglich ausgeführt werden. Idealerweise bei jedem Commit!

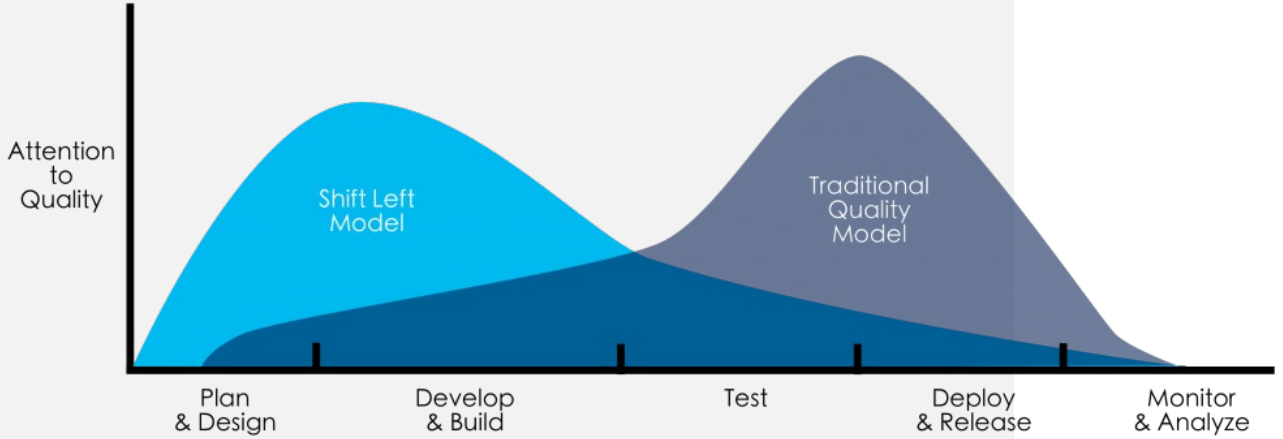
Bei aufwändigen Tests, wird man dies auf ggf. einmal pro Tag, pro Woche, o. ähnl. reduzieren.

SHIFT – LEFT

Testing

- Shift-Left Testing ist ein Ansatz bei dem das Testen von Software so früh wie möglich im Entwicklungszyklus durchgeführt wird.
- "Shift Left" bezieht sich auf die Verschiebung des Testens nach "links" im Entwicklungszeitplan, d.h. näher an den Anfang des Entwicklungsprozesses.
- Shift-Left Testing ist ein wesentlicher Bestandteil von DevOps und agilen Entwicklungsmodellen.





EVERYTHING

AS C<>DE

Build-as-Code

Beschreibt wie Anwendungen (oder Komponenten davon) gebaut werden. Z.B.:

- Maven
- Gradle
- …

Test-as-Code

Beschreibt wie das Projekt getestet wird. Z.B.:

- Unit-, Component-, Integrationsteste
- API-, UI-Teste
- Performance Teste

Infrastructure-as-Code

Beschreibt wie Laufzeitumgebungen (Deployment Environments) automatisiert aufgebaut werden. Z.B.:

- Docker
- Terraform, Vagrant, Ansible
- …

Pipeline-as-Code

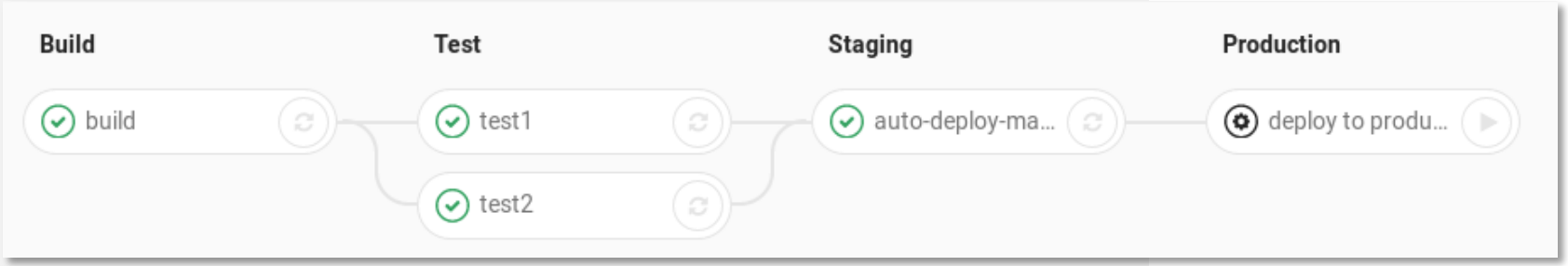
Beschreibt alle automatisiert ablaufenden Schritte bis zur lauffähigen Installation. Z.B.:

- Build-Pipeline per Jenkins
- Gitlab CI
- …

Alles was die Continuos Delivery und Deployment Umgebung mit Leben füllt kommt aus dem VCS

BEISPIEL

GitLab Pipelines



Pipelines

Bestehen aus Stages und Jobs

- Stages (Phasen) definieren wann etwas zu tun ist (erst kompilieren, dann testen)
- Jobs beschreiben, was zu tun ist (kompilieren, testen).

Pipelines werden als Textdatei notiert und stehen versioniert im Repository.

`.gitlab-ci.yaml`

Pipeline as code

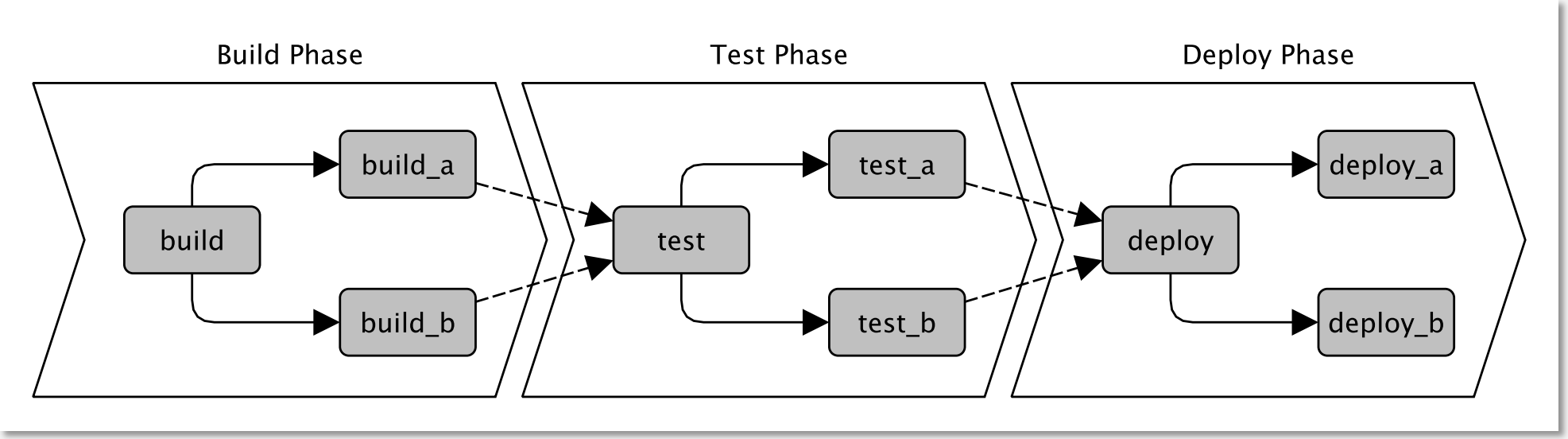
```
stages:
-Build
-Test
-Staging
-Production

build:
   stage: build
   script: make build dependencies

test 1:
   stage: Test
   script: make test1

test 2:
   stage: Test
   script: make test2

auto-deploy-machine:
   stage: Staging
   script: terraform apply

[…]
```

Schon gewusst?

GitLab Pipelines können Sie schon immer im GitLab THL Service nutzen.

DEPLOYMENT PIPELINES AS CODE

Phasen Pipelines



- Jobs werden von einer Deployment-Infrastruktur ausgeführt.
- Mehrere Jobs in derselben Phase werden parallel ausgeführt, wenn es genügend Ausführungsressourcen der Deployment-Infrastruktur gibt.
- Jobs werden meist als Liste von Shell-Anweisungen (also z. B. Bash-Skripte) ausgedrückt.
- Ob ein Job erfolgreich ist (und damit die Pipeline), ergibt sich aus dem Exit Code des Prozesses.
  - Ein **Exit Code von 0** bedeutet, dass ein Job **erfolgreich** ist und die Pipeline fortgesetzt werden kann.
  - Ein **Exit Code ungleich 0** bedeutet, dass ein Job **fehlerhaft** ist und die weitere Bearbeitung der Pipeline abzubrechen ist.

CLOUD NATIVE COMPUTING

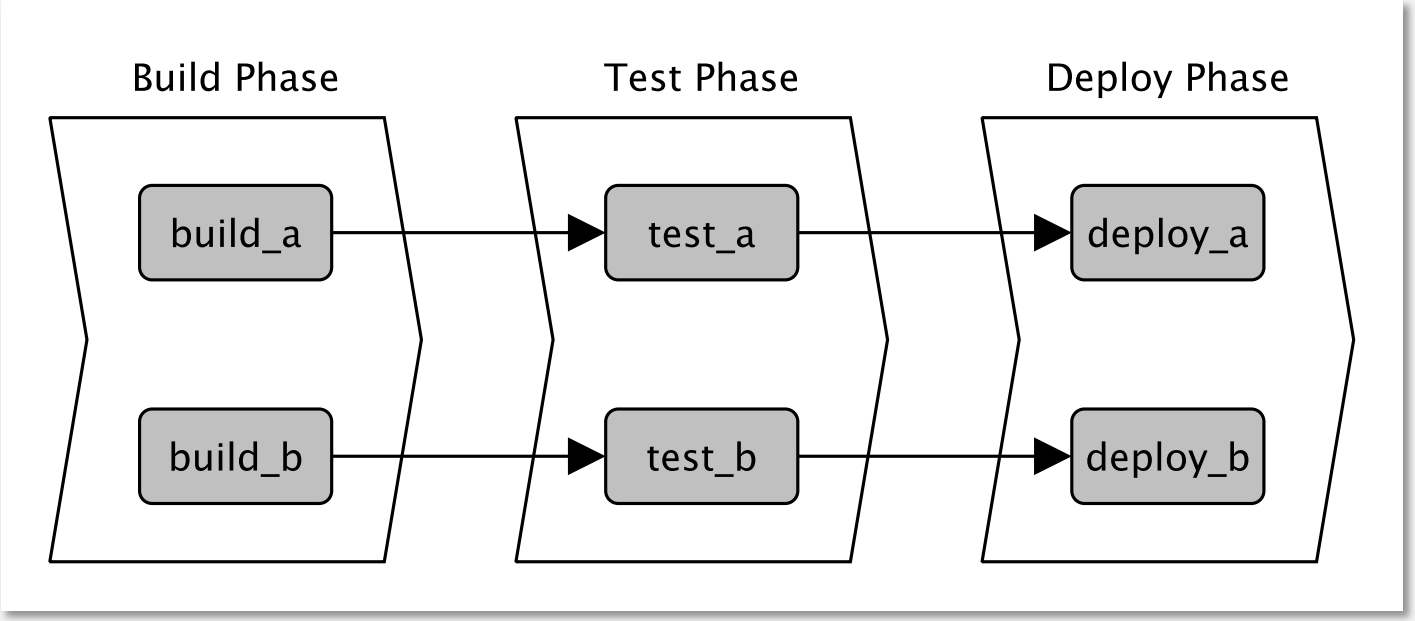
Pipelines werden meist durch einen der folgende Trigger gestartet:

- *Code Commits*
- *Periodische Trigger (z.B. nightly-builds)*
- *Manuell (z.B. Deinstallationsprozesse)*

Phasen Pipelines sind für einfache Projekte geeignet.

DEPLOYMENT PIPELINES AS CODE

Gerichtete Pipelines (DAG)



Gerichtete Pipelines oder DAG Pipelines (DAG = Directed Acyclic Graph) sind für größere und komplexere Projekte geeignet, die effizient ausgeführt werden müssen, da ansonsten die Build- und Deployment-Zeiten zu lang dauern und ineffizient werden würden.

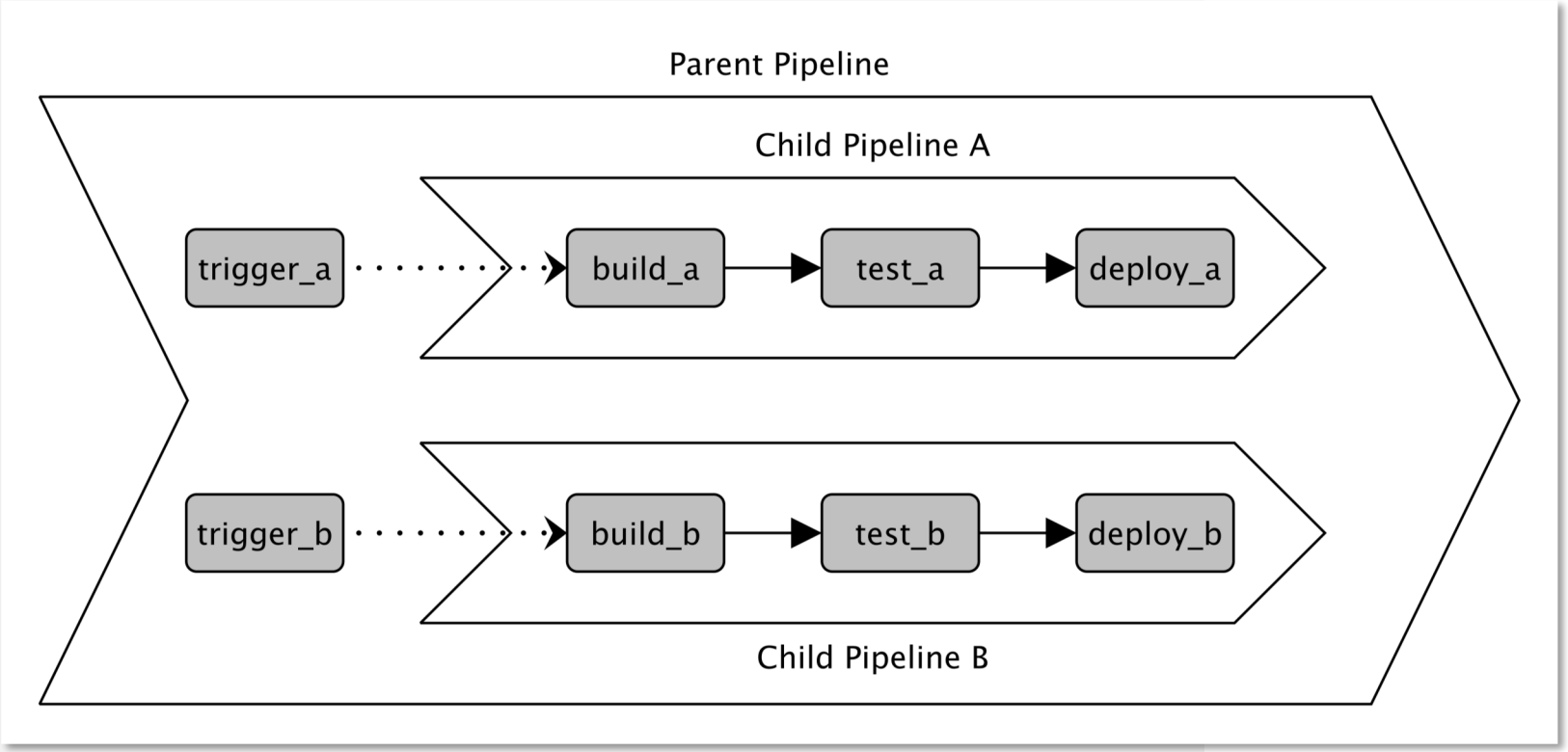
Gerichtete Pipelines können DevOps-Prozesse beschleunigen, indem der Pipeline zusätzliche Informationen zwischen Job Abhängigkeiten mitgegeben werden.

DEPLOYMENT PIPELINES AS CODE

Hierarchische Pipelines

Hierarchische Pipelines sind insbesondere für Monorepos und Projekte mit vielen unabhängig definierten Komponenten geeignet.

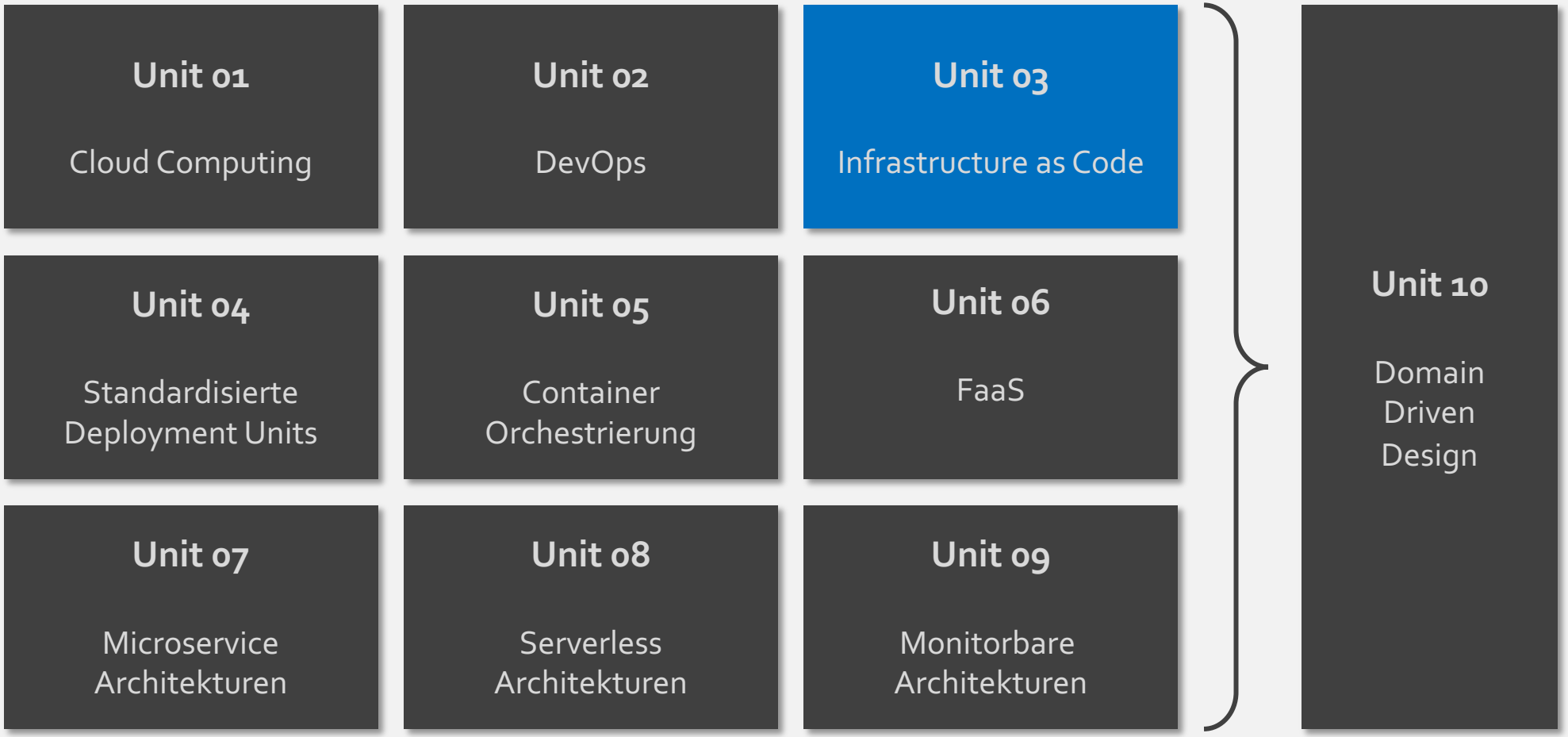
Unter einem Monorepo versteht man eine Software-entwicklungsstrategie, bei der Code für viele Projekte in einem einzigen großen Repository gespeichert wird.



Da Monorepositories üblicherweise alle Teilkomponenten eines großen Systems beinhalten, die aber dennoch durch jeweils eigenständige Teams betreut und deren Pipelines gebaut werden, bieten sich hierarchische Pipelines vor allem für den Bau von großen Systemen an.

AUSBLICK

Überblick über Units und Themen dieses Moduls



KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de