# DevOps

## Von der Idee bis in den Betrieb

---

Schulungsunterlage für Auszubildende,
Umschüler und Quereinsteiger in den IT-Betrieb

*Diese Unterlage begleitet eine einzige Anwendung — den KFZ-Rechner — von einem Ordner auf einem Notebook bis zu einem versionierten, getesteten, containerisierten und überwachten Dienst, der mehrmals täglich ohne Ausfall ausgeliefert wird.*

*Neunzehn Kapitel, jedes mit Lernziel, Praxisteil, Anti-Pattern und einer Frage an den eigenen Betrieb.*

Stand: 29. Juli 2026

# Inhaltsverzeichnis

# Kapitel 1

# DevOps

*Schulungsunterlage für IT-Auszubildende, Umschüler und Quereinsteiger in den IT-Betrieb.*

DevOps ist keine Software, die man installiert, und keine Stelle, die man besetzt. Es ist eine Arbeitsweise, die Entwicklung und Betrieb wieder zusammenbringt — organisatorisch, kulturell und technisch. Diese Unterlage behandelt alle drei Ebenen, mit deutlichem Schwerpunkt auf dem, was man tatsächlich tut.

## 1.1 Für wen ist das gedacht

| Zielgruppe | Vorkenntnisse |
|---|---|
| Auszubildende Fachinformatik (Systemintegration und Anwendungsentwicklung) | Linux-Kommandozeile, Grundverständnis Netzwerk |
| Umschüler und Quereinsteiger | erste Programmier- oder Administrationserfahrung |
| Administratoren, die in Richtung Automatisierung wollen | Berufserfahrung im Betrieb |

Programmierkenntnisse sind hilfreich, aber keine Voraussetzung. Wer ein Shell-Skript lesen kann, kommt durch.

## 1.2 Wie diese Unterlage aufgebaut ist

Jedes Kapitel folgt demselben Muster:

- **Lernziel** — was du danach können sollst
- **Theorie** — knapp gehalten, nur so viel wie nötig
- **Praxis** — ein Lab zum Mitmachen
- **Anti-Pattern** — was in der Praxis regelmäßig schiefgeht
- **Reflexionsfrage** — bezogen auf deinen eigenen Betrieb

Die Kapitel bauen aufeinander auf. Wer quer einsteigt, sollte zumindest Teil A gelesen haben.

## 1.3 Das durchgehende Fallbeispiel

Durch alle Kapitel begleitet uns dieselbe Anwendung: **KFZ-Rechner**, ein kleiner Webdienst, der eine Versicherungsprämie berechnet. Zu Beginn liegt er als Ordner auf einem Notebook. Am Ende der Unterlage wird er versioniert, automatisch getestet, containerisiert, überwacht und mehrmals täglich ohne Ausfall ausgeliefert.

Jedes Kapitel fügt genau ein Stück hinzu. Das ist Absicht: DevOps entsteht nicht durch eine große Umstellung, sondern durch viele kleine.

## 1.4 Zeitbedarf

| Umfang | Dauer |
|---|---|
| Nur lesen | etwa 6 Stunden |
| Mit allen Labs | 3 bis 4 Tage |
| Als begleitete Schulung | 5 Tage |

## 1.5 Werkzeuge für die Labs

Alles, was gebraucht wird, läuft auf einem Linux-System mit 8 GB RAM — ein Notebook, eine VM oder ein Raspberry Pi 5 genügen.

- Git
- Podman oder Docker
- ein Editor
- ein Git-Server (Gitea, GitLab oder GitHub — auch selbst gehostet)
- optional: Ansible, Prometheus, Grafana

Verwandte Unterlagen in diesem Wiki:

- DevOps und ITIL
- Development
- Linux Server

## 1.6 Legende

Farbig hinterlegte Kästen bedeuten:

**Tipp**

Ein Praxistipp oder eine Empfehlung aus der Erfahrung.

**Wichtig**

Etwas, das man wissen muss — typischerweise eine Stelle, an der es sonst klemmt.

**Achtung**

Eine Warnung. Hier kann echter Schaden entstehen.

*Weiter mit Kapitel 01 — Warum DevOps*

# Teil I

# Teil A — Warum überhaupt

# Kapitel 2

# Warum DevOps

**Lernziel**

Nach diesem Kapitel kannst du erklären, welches konkrete organisatorische Problem DevOps löst, was ein Wertstrom ist und warum Übergaben zwischen Abteilungen teuer sind. Du kannst außerdem drei verbreitete Missverständnisse über DevOps benennen.

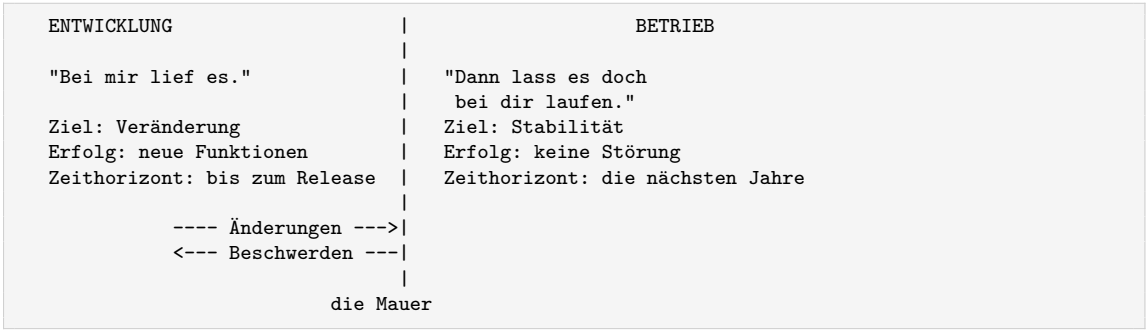
## 2.1 Zwei Abteilungen, zwei Zielvorgaben

Stell dir zwei Teams vor, die beide hervorragende Arbeit leisten und sich trotzdem gegenseitig blockieren.

**Die Entwicklung** wird daran gemessen, wie viele Funktionen sie liefert. Ihr Erfolg ist Veränderung. Je mehr sie ausliefert, desto besser ihre Bilanz.

**Der Betrieb** wird daran gemessen, wie stabil die Systeme laufen. Sein Erfolg ist Beständigkeit. Und die größte Einzelursache für Störungen ist — statistisch belegbar — die Veränderung.

Beide Teams verhalten sich also völlig rational, wenn sie das jeweils andere ausbremsen. Die Entwicklung wirft Änderungen über den Zaun, der Betrieb baut immer höhere Zäune. Man hat dafür einen Namen gefunden: die **Mauer der Verwirrung**.




Das Bemerkenswerte daran: Es gibt keinen Schuldigen. Niemand ist faul, niemand ist böswillig. Das Problem steckt in der Zielvorgabe, nicht in den Menschen. Wer das verstanden hat, hat die halbe Miete — denn jeder Lösungsversuch, der auf „die Kollegen müssen sich mal zusammenreißen"hinausläuft, ist damit schon widerlegt.

## 2.2 Der Wertstrom

Der **Wertstrom** ist der gesamte Weg einer Idee bis zu dem Moment, in dem ein Kunde etwas davon hat. Bei Software heißt das: von „jemand hat einen Wunsch"bis „die Änderung läuft in Produktion".

Interessant ist dabei weniger die Summe der Arbeitszeit als das Verhältnis zwischen Arbeiten und Warten. Ein realistisches Beispiel:

| Schritt | Bearbeitungszeit | Wartezeit davor |
|---|---|---|
| Anforderung aufnehmen | 2 h | 5 Tage |
| Entwickeln | 6 h | 3 Tage |
| Code Review | 1 h | 4 Tage |
| Testumgebung anfordern | 15 min | 9 Tage |
| Testen | 4 h | 2 Tage |
| Änderungsantrag stellen | 1 h | 14 Tage bis zum nächsten CAB |
| Ausrollen | 3 h | 6 Tage bis zum Wartungsfenster |
| **Summe** | **etwa 2 Arbeitstage** | **etwa 43 Tage** |

Die eigentliche Arbeit macht keine fünf Prozent der Durchlaufzeit aus. Der Rest ist Warten — auf Freigaben, auf Umgebungen, auf Termine, auf andere Menschen.

**Wichtig**

**Daraus folgt die vielleicht wichtigste Erkenntnis dieser Unterlage:** Wer schneller werden will, muss nicht schneller arbeiten. Er muss die Wartezeiten beseitigen.
Ein Entwickler, der doppelt so schnell programmiert, verkürzt die Durchlaufzeit im obigen Beispiel von 45 auf 44,6 Tage. Ein automatisch bereitgestelltes Testsystem spart neun Tage. Das ist der Unterschied zwischen Optimierung an der falschen und an der richtigen Stelle.

## 2.3 Warum Übergaben so teuer sind

Jede Übergabe zwischen zwei Teams kostet mehr, als sie auf dem Papier sollte:

- **Wissen geht verloren.** Was der Entwickler über die Änderung weiß, steht nie vollständig im Ticket.
- **Es entsteht Wartezeit.** Der Empfänger arbeitet gerade an etwas anderem.
- **Rückfragen dauern.** Jede Rückfrage ist ein weiterer Wartezyklus.
- **Verantwortung verdunstet.** Nach drei Übergaben fühlt sich niemand mehr zuständig.

Bei sieben Übergaben von der Idee bis zur Produktion — was in großen Organisationen keineswegs viel ist — potenziert sich das. Die Arbeit ist irgendwann fertig; das Warten nicht.

Man kann sich das wie die Bearbeitungswege einer Verwaltung vorstellen, in der jeder Antrag in dreifacher Ausfertigung an eine andere Stelle weitergereicht wird, wobei mindestens eine Ausfertigung stets im Keller landet. Wer die Anhalter-Bücher kennt, weiß, dass die Umgehungsstraße dann trotzdem gebaut wird — nur eben ohne dass jemand vorher davon erfahren hätte.

## 2.4 Was DevOps ist — und was nicht

Der Begriff entstand 2009. Auf der Velocity-Konferenz zeigten zwei Mitarbeiter von Flickr, dass ihr Unternehmen mehr als zehnmal am Tag ausliefert — zu einer Zeit, als vierteljährliche Releases als ambitioniert galten. Wenige Monate später organisierte Patrick Debois in Gent die ersten *devopsdays*. Das Kunstwort aus *Development* und *Operations* blieb hängen.

**DevOps ist:**

- eine Arbeitsweise, die den gesamten Wertstrom betrachtet statt einzelner Abteilungsziele
- geteilte Verantwortung für das Ergebnis, nicht nur für den eigenen Arbeitsschritt
- Automatisierung von allem, was wiederholt und fehleranfällig ist
- kurze Rückmeldeschleifen

**DevOps ist nicht:**

| Missverständnis | Warum es falsch ist |
|---|---|
| „Wir richten ein DevOps-Team ein" | Dann gibt es drei Silos statt zwei. Die Mauer wandert nur. |
| „DevOps ist Jenkins"(oder GitLab, oder Kubernetes) | Werkzeuge unterstützen die Arbeitsweise. Sie erzeugen sie nicht. |
| „DevOps heißt, dass Entwickler den Betrieb machen" | Es heißt geteilte Verantwortung, nicht verschobene. |
| „DevOps ersetzt ITIL" | Beide beantworten unterschiedliche Fragen. Siehe Kapitel 17. |
| „DevOps bedeutet, ohne Tests direkt in Produktion" | Das Gegenteil ist der Fall — mehr Tests, nur automatisiert. |

**Achtung**

**Das teuerste Missverständnis** ist die DevOps-Stellenausschreibung. Wer „einen DevOpsßucht, sucht in Wahrheit meist einen Automatisierungsingenieur — und benennt damit eine Arbeitsweise in eine Person um. Das Ergebnis ist regelmäßig eine einzelne Person, die zwischen zwei unveränderten Abteilungen aufgerieben wird.

## 2.5 Das Fallbeispiel: KFZ-Rechner

Ab hier begleitet uns durchgehend dieselbe Anwendung.

**Ausgangslage.** Der *KFZ-Rechner* ist ein kleiner Webdienst, der aus Fahrzeugtyp, Alter und Schadenfreiheitsklasse eine Prämie errechnet. Er besteht aus etwa 400 Zeilen Python und läuft seit drei Jahren.

| Merkmal | Zustand heute |
|---|---|
| Quellcode | ein Ordner auf dem Notebook eines Kollegen, dazu ein ZIP-Archiv auf einem Netzlaufwerk |
| Test | manuell, „einmal durchklicken" |
| Auslieferung | Dateien per SCP kopieren, Dienst neu starten |
| Häufigkeit | etwa alle zwei Monate |
| Wartungsfenster | Samstag, 4 Stunden |
| Rollback | Sicherung zurückspielen, Dauer unbekannt |
| Überwachung | ein Kollege ruft morgens die Seite auf |
| Dokumentation | eine Wiki-Seite von 2022 |

Nichts davon ist ungewöhnlich. Genau so sehen sehr viele gewachsene Anwendungen aus, und meistens funktionieren sie sogar — bis sie es nicht mehr tun.

**Zielzustand am Ende dieser Unterlage.** Der Code liegt in Git, jede Änderung wird automatisch getestet, gebaut und containerisiert, mehrmals täglich ohne Wartungsfenster ausgeliefert, überwacht, und ein Rollback dauert unter einer Minute.

Wir kommen dorthin in neunzehn kleinen Schritten. Nicht in einem großen.

## 2.6 Lab: Den eigenen Wertstrom aufzeichnen

Für dieses Lab brauchst du keinen Rechner, sondern Papier.

1. Wähle eine echte Änderung, die zuletzt in Produktion gegangen ist.
2. Zeichne jeden Schritt vom ersten Wunsch bis zum Produktivgang auf — auch die, die dir selbstverständlich erscheinen.
3. Notiere je Schritt zwei Zahlen: die reine Bearbeitungszeit und die Wartezeit davor.
4. Markiere jede Übergabe an eine andere Person oder Abteilung mit einem Kreuz.
5. Rechne aus: Bearbeitungszeit geteilt durch Gesamtdauer. Das ist deine **Flow-Effizienz**.

Werte zur Einordnung: Unter 5 Prozent ist üblich. Über 25 Prozent ist bereits sehr gut. Wer 100 Prozent herausbekommt, hat vermutlich falsch gemessen.

**Tipp**

Mach das Lab wirklich. Diese eine Zeichnung erklärt in fünf Minuten mehr über die eigene Organisation als jede Schulungsfolie — und sie liefert dir die Argumente, mit denen du später Verbesserungen begründen kannst. Wer mit „wir sollten mal agiler werdenïn eine Besprechung geht, wird höflich angehört. Wer mit „unsere Testumgebung kostet uns neun Wartetage pro Änderung"hineingeht, wird gehört.

## 2.7 Anti-Pattern

| Anti-Pattern | Woran man es erkennt | Was stattdessen hilft |
|---|---|---|
| Das DevOps-Team | Eine neue Abteilung mit neuem Namen, alte Übergaben | Verantwortung in die bestehenden Teams verlagern |
| Werkzeug zuerst | „Wir führen jetzt GitLab ein, dann wird alles gut" | Erst den Engpass finden, dann das Werkzeug wählen |
| Lokale Optimierung | Ein Team wird doppelt so schnell, die Durchlaufzeit bleibt gleich | Den gesamten Wertstrom betrachten |
| Kennzahl ohne Zweck | Deployment-Frequenz steigt, Störungen auch | Immer paarweise messen, siehe Kapitel 15 |
| Kultur per Rundmail | Ein Poster mit Werten hängt im Flur | Anreize und Zielvorgaben ändern |

## 2.8 Reflexionsfrage

*Wie lange dauert es in deinem Betrieb von der fertigen Codeänderung bis zum Produktivgang — und welcher einzelne Schritt darin ist der längste?*

Wenn die Antwort „das weiß ich nicht"lautet, ist das keine Wissenslücke, sondern schon der erste Befund.

# Kapitel 3

# Kultur und Prinzipien

**Lernziel**

Nach diesem Kapitel kennst du die fünf CALMS-Dimensionen und die Drei Wege und kannst sie an konkreten Beispielen erläutern. Du kannst außerdem erklären, warum die Organisationsstruktur die Softwarearchitektur bestimmt, und weißt, was das DIKW-Modell für die Fehlersuche taugt.

## 3.1 CALMS

CALMS ist ein Ordnungsrahmen, mit dem sich beurteilen lässt, wo eine Organisation steht. Er entstand um 2010 als **CAMS** bei Damon Edwards und John Willis; das *L* für Lean kam später durch Jez Humble hinzu.

| Buchstabe | Bedeutung | Prüffrage |
|---|---|---|
| **C**ulture | Zusammenarbeit, geteilte Verantwortung, Umgang mit Fehlern | Was passiert bei uns nach einer Störung — Ursachenanalyse oder Schuldsuche? |
| **A**utomation | Alles, was wiederholt und fehleranfällig ist, wird automatisiert | Welchen Handgriff machen wir mehr als zwölfmal im Jahr von Hand? |
| **L**ean | Verschwendung reduzieren, kleine Arbeitspakete, Fluss statt Stapel | Wie groß ist unser durchschnittliches Release? |
| **M**easurement | Entscheidungen auf Daten stützen | Woher wissen wir, ob die letzte Änderung etwas verbessert hat? |
| **S**haring | Wissen und Erkenntnisse teilen, nicht nur Ergebnisse | Wo steht, was wir bei der letzten Störung gelernt haben? |

Der praktische Wert liegt in den Prüffragen. Wer sie ehrlich beantwortet, hat binnen einer halben Stunde eine belastbare Standortbestimmung.

**Wichtig**

**Culture steht nicht zufällig vorn.** Automatisierung in einer Organisation, in der nach jeder Störung ein Schuldiger gesucht wird, führt zu einem vorhersagbaren Ergebnis: Niemand traut sich mehr, etwas auszurollen. Die Werkzeuge sind dann installiert, aber

ungenutzt.
Kultur lässt sich allerdings nicht anordnen. Sie folgt den Anreizen. Wenn Beförderungen an Störungsfreiheit hängen, wird niemand Risiken eingehen — unabhängig davon, was im Leitbild steht.

## 3.2 Die Drei Wege

Das Modell stammt von Gene Kim und ist in *The Phoenix Project* sowie im *DevOps Handbook* ausgeführt. Es beschreibt drei aufeinander aufbauende Prinzipien.

### 3.2.1 Der Erste Weg: Fluss

Arbeit fließt von links nach rechts — von der Idee über die Entwicklung und den Betrieb bis zum Kunden. Ziel ist, diesen Fluss zu beschleunigen.

**Konkret heißt das:**

- Arbeit sichtbar machen (Kanban-Board, Ticketsystem)
- Arbeitspakete klein halten — kleine Änderungen fließen schneller und brechen seltener
- angefangene Arbeit begrenzen (Work in Progress)
- Engpässe finden und dort ansetzen, nicht anderswo
- niemals einen bekannten Fehler nach rechts weitergeben

**Tipp**

**Der Merksatz für den Ersten Weg:** Eine kleine Änderung, die schiefgeht, ist ein Ärgernis. Eine große Änderung, die schiefgeht, ist ein Vorfall. Bei einer Auslieferung mit zweihundert Änderungen weiß niemand, welche davon das Problem verursacht hat — und die Fehlersuche dauert länger als die Entwicklung.

### 3.2.2 Der Zweite Weg: Rückmeldung

Information fließt von rechts nach links — und zwar möglichst schnell und in möglichst vielen Schleifen.

- automatisierte Tests melden Fehler in Minuten statt in Wochen
- Überwachung meldet Probleme, bevor Anwender anrufen
- Entwickler sehen, wie sich ihr Code in Produktion verhält
- der Betrieb ist früh eingebunden, nicht erst beim Ausrollen

Der Wert einer Rückmeldung sinkt mit ihrem Alter. Ein Fehler, der beim Commit auffällt, kostet Minuten. Derselbe Fehler in Produktion kostet je nach Quelle das Zehn- bis Hundertfache — und das liegt weniger am Aufwand der Korrektur als am verlorenen Kontext: Nach sechs Wochen weiß niemand mehr, warum diese Zeile so geschrieben wurde.

### 3.2.3 Der Dritte Weg: Lernen und Experimentieren

Eine Kultur, in der Experimente erwünscht sind und Fehler als Erkenntnisquelle gelten.

- Zeit für Verbesserung ist eingeplant, nicht „wenn mal Luft ist"
- Störungen werden ohne Schuldzuweisung aufgearbeitet
- Erkenntnisse werden geteilt, nicht in Teams eingeschlossen
- bewusst wird Belastung erzeugt, um Widerstandsfähigkeit zu prüfen

Der dritte Weg ist der, der am häufigsten übersprungen wird. Er lässt sich nicht kaufen und zeigt keine schnellen Ergebnisse — er ist aber der einzige, der die ersten beiden dauerhaft trägt.

## 3.3 Conways Gesetz

1967 formulierte Melvin Conway eine Beobachtung, die seither erstaunlich zuverlässig zutrifft: **Organisationen entwerfen Systeme, die ihre eigenen Kommunikationsstrukturen abbilden.**

Praktische Folgen:

| Organisation | Erwartbare Architektur |
|---|---|
| Vier Abteilungen, die selten miteinander sprechen | Vier Module mit umständlichen Schnittstellen |
| Eine getrennte Datenbankabteilung | Alle Anwendungen teilen sich eine zentrale Datenbank |
| Ein zentrales Betriebsteam als Nadelöhr | Alles muss durch dieses Team, egal wie klein |
| Kleine, eigenständige Produktteams | Kleine, eigenständige Dienste |

Das erklärt, warum Architekturprojekte scheitern, die die Organisation unangetastet lassen. Wer Microservices will, aber die Freigabe jeder Änderung zentral bündelt, bekommt einen verteilten Monolithen — also alle Nachteile beider Ansätze.

Die praktische Anwendung heißt **umgekehrtes Conway-Manöver**: Man verändert die Teamstruktur so, dass sie der gewünschten Architektur entspricht — und lässt die Architektur folgen.

## 3.4 Das DIKW-Modell

DIKW unterscheidet vier Stufen, auf denen Erkenntnis entsteht.

| Stufe | Bedeutung | Beispiel aus dem Betrieb |
|---|---|---|
| **D**ata | Rohdaten ohne Zusammenhang | `HTTP 500` um 14:32:07 |
| **I**nformation | verknüpfte Daten: wer, was, wann, wo | 340 Fehler in fünf Minuten, alle im Prämienrechner |
| **K**nowledge | daraus abgeleitete Erkenntnis | Der Fehler trat nach dem Ausrollen von Version 2.4 auf |
| **W**isdom | Erfahrung, Werte, Urteilsvermögen | Wir rollen zurück und untersuchen morgen — nicht umgekehrt |

Der Nutzen für die Praxis liegt in der Diagnose: Bei einer Störung lässt sich fragen, auf welcher Stufe es hakt.

- Fehlen **Daten**? Dann fehlt Instrumentierung — siehe Kapitel 12.
- Fehlt **Information**? Dann sind die Daten da, aber nicht auswertbar. Logs ohne Zeitstempel, Metriken ohne Bezeichner.
- Fehlt **Wissen**? Dann fehlt Erfahrung oder Dokumentation. Runbooks helfen.
- Fehlt **Weisheit**? Dann fehlt Entscheidungsspielraum — meist ein Führungsthema.

Die meisten Organisationen ertrinken übrigens in Daten und verhungern an Information. Ein Dashboard mit vierzig Diagrammen, das niemand liest, ist der Regelfall, nicht die Ausnahme.

## 3.5 Lab: CALMS-Selbsteinschätzung

Bewerte deinen Betrieb — oder deine Ausbildungsabteilung — in jeder Dimension von 1 bis 5.

```
Culture      1 -- 2 -- 3 -- 4 -- 5
  1 = Störungen enden mit der Frage "wer war das?"
  5 = Störungen enden mit einer dokumentierten Ursachenanalyse

Automation   1 -- 2 -- 3 -- 4 -- 5
  1 = Auslieferung ist eine Anleitung mit 30 Handgriffen
  5 = Auslieferung ist ein Knopfdruck

Lean         1 -- 2 -- 3 -- 4 -- 5
  1 = zwei große Releases im Jahr
  5 = viele kleine Änderungen, laufend

Measurement  1 -- 2 -- 3 -- 4 -- 5
  1 = wir merken Störungen, wenn Anwender anrufen
  5 = wir sehen Abweichungen, bevor sie wirken

Sharing      1 -- 2 -- 3 -- 4 -- 5
  1 = Wissen steckt in Köpfen
  5 = Wissen steht dokumentiert und wird gepflegt
```

Anschließend die entscheidende Frage: **Welche Dimension hat den niedrigsten Wert — und was wäre der kleinste sinnvolle Schritt, um von 2 auf 3 zu kommen?**

Nicht auf 5. Auf 3. Wer Verbesserungen zu groß plant, setzt sie nicht um.

## 3.6 Anti-Pattern

| Anti-Pattern | Woran man es erkennt | Was stattdessen hilft |
|---|---|---|
| Kultur per Dekret | Werteplakate im Flur, unveränderte Zielvorgaben | Anreize ändern, nicht Poster aufhängen |
| Automatisierung ohne Kultur | Pipeline vorhanden, aber niemand traut sich auszuliefern | Erst Vertrauen und Rückrollbarkeit, dann Tempo |
| Sharing als Einbahnstraße | Nur Erfolge werden geteilt | Auch Fehlschläge dokumentieren — die lehren mehr |
| Kennzahlen als Druckmittel | Teams werden anhand der Deployment-Frequenz verglichen | Kennzahlen zur Selbststeuerung, nicht zur Bewertung |
| Architektur ohne Organisation | Microservices im Entwurf, zentrale Freigabe im Alltag | Conway ernst nehmen |

## 3.7 Reflexionsfrage

*Wenn morgen ein Produktivsystem drei Stunden ausfällt — wie läuft die Aufarbeitung bei euch ab? Wer sitzt im Raum, welche Frage wird zuerst gestellt, und was ist das schriftliche Ergebnis?*

Die Antwort auf diese eine Frage sagt mehr über den Reifegrad einer Organisation aus als jede Werkzeugliste.

# Teil II

# Teil B — Wie man arbeitet

# Kapitel 4

# Versionskontrolle und Branching

**Lernziel**

Nach diesem Kapitel kannst du ein Repository sinnvoll strukturieren, brauchbare Commits schreiben und die gängigen Branching-Strategien gegeneinander abwägen. Du weißt, warum langlebige Branches Continuous Integration unmöglich machen, und hast den KFZ-Rechner unter Versionskontrolle gebracht.

## 4.1 Warum das hier zuerst kommt

Alles Weitere in dieser Unterlage setzt eines voraus: **Es gibt genau einen Ort, an dem der aktuelle Stand steht, und dieser Ort kennt seine eigene Geschichte.**

Ohne Versionskontrolle gibt es keine Pipeline, keine Nachvollziehbarkeit, kein Rollback und keine Automatisierung. Der Ordner `anwendung_final_v3_neu_KORRIGIERT` ist kein Scherz aus einer Präsentation, sondern der Normalzustand in erschreckend vielen Fachabteilungen.

Git ist dabei nicht einfach ein Ablageort. Es ist die **Quelle der Wahrheit** — und später auch der Auslöser für alles, was automatisch passiert.

## 4.2 Das Minimum an Git

```
# Repository anlegen
git init
git config user.name  "Vorname Nachname"
git config user.email "vorname.nachname@firma.de"

# Stand ansehen
git status
git diff
git log --oneline --graph --decorate --all

# Änderungen aufnehmen
git add datei.py
git add -p              # abschnittsweise – sehr zu empfehlen
git commit -m "Nachricht"

# Mit dem Server abgleichen
git pull --rebase
git push
```

```
# Zweig anlegen und wechseln
git switch -c feature/schadenfreiheitsklasse
git switch main

# Zusammenführen
git merge feature/schadenfreiheitsklasse
```

**Tipp**

`git add -p` geht die Änderungen abschnittsweise durch und fragt jeweils, ob sie in den Commit sollen. Das erzwingt, dass man den eigenen Code noch einmal liest, bevor man ihn festschreibt — und verhindert zuverlässig, dass Debug-Ausgaben oder auskommentierte Experimente mitwandern. Wer sich diese eine Angewohnheit aneignet, hebt die Qualität seiner Commits sofort.

## 4.3 Was gehört ins Repository — und was nicht

| Gehört hinein | Gehört nicht hinein |
|---|---|
| Quellcode | Passwörter, Schlüssel, Zertifikate |
| Konfigurationsvorlagen | echte Konfiguration mit Zugangsdaten |
| Containerfile, Pipeline-Definition | Build-Ergebnisse, kompilierte Dateien |
| Datenbankmigrationen | Abhängigkeiten (`node_modules`, `venv`) |
| Tests und Testdaten | Produktivdaten |
| Dokumentation | große Binärdateien |

`.gitignore`

```
__pycache__/
*.pyc
.venv/
.env
*.log
.idea/
.vscode/
dist/
```

**Achtung**

**Ein einmal eingecheckter Zugangsschlüssel ist kompromittiert — auch nach dem Löschen.** Git vergisst nichts; der alte Commit bleibt in der Historie und in jedem geklonten Repository auf jedem Notebook erhalten.
Die einzig richtige Reaktion lautet: **den Schlüssel oder das Passwort sofort ändern.** Das Bereinigen der Historie ist optional und aufwendig, das Wechseln des Geheimnisses ist Pflicht.
Vorbeugen lässt sich mit einem Pre-Commit-Hook und Werkzeugen wie `gitleaks` — mehr dazu in Kapitel 14.

## 4.4 Brauchbare Commits

Ein Commit sollte **eine abgeschlossene, für sich verständliche Änderung** enthalten. Die Nachricht erklärt das *Warum*; das *Was* steht bereits im Diff.

```
Schlecht                        Gut
-------------------------       ------------------------------------
"fix"                           "Division durch null bei SF-Klasse 0
"Änderungen"                     verhindern"
"asdf"
"funktioniert jetzt endlich"    "Prämienberechnung auf Dezimaltyp
"WIP"                            umgestellt

                                 Fließkomma führte bei Beträgen über
                                 10.000 EUR zu Rundungsdifferenzen
                                 von bis zu 2 Cent.
                                 Siehe Ticket #4711."
```

Verbreitet und praktisch ist die Konvention **Conventional Commits**:

```
feat:     neue Funktion
fix:      Fehlerbehebung
docs:     nur Dokumentation
refactor: Umbau ohne Verhaltensänderung
test:     Tests ergänzt oder geändert
chore:    Wartungsarbeiten, Abhängigkeiten

Beispiel:
feat(praemie): Rabatt fuer Wenigfahrer ergaenzen
fix(api): Zeitzone bei Vertragsbeginn korrigieren
```

Der Nutzen ist nicht kosmetisch: Aus solchen Nachrichten lassen sich Änderungsprotokolle automatisch erzeugen, und die Versionsnummer kann maschinell abgeleitet werden — `feat` erhöht die Nebenversion, `fix` die Patch-Nummer.

## 4.5 Semantische Versionierung

```
MAJOR . MINOR . PATCH
  |       |       +-- Fehlerbehebung, abwärtskompatibel
  |       +---------- neue Funktion, abwärtskompatibel
  +------------------ Änderung, die Bestehendes bricht

2.4.7 -> 2.4.8   Fehler behoben
2.4.8 -> 2.5.0   Funktion ergänzt
2.5.0 -> 3.0.0   Schnittstelle geändert
```

```
git tag -a v2.5.0 -m "Rabatt für Wenigfahrer"
git push --tags
```

Der entscheidende Punkt ist die **dritte Stelle von links**: Eine erhöhte Hauptversion ist ein Versprechen an alle, die das System benutzen — hier musst du etwas anpassen. Wer das ernst nimmt, erspart seinen Anwendern böse Überraschungen.

## 4.6 Branching-Strategien

Hier wird es entscheidungsrelevant, denn die Wahl bestimmt, ob Continuous Integration überhaupt möglich ist.

### 4.6.1 GitFlow

2010 von Vincent Driessen beschrieben. Sieht mehrere dauerhafte Zweige vor: `main`, `develop`, dazu `feature/`, `release/` und `hotfix/`.

```
main     --?--------------?--------------?-->  nur Releases
           \            /              /
release     \      ?---?              /
             \    /                  /
develop  --?---?--?------?---?-------?---->
           \       \       /
feature     ?---?---?     ?
```

**Passt zu:** Software mit mehreren parallel gepflegten Versionen, langen Freigabezyklen, ausgelieferten Produkten beim Kunden.

**Passt nicht zu:** Continuous Delivery. Der Autor selbst hat 2020 einen Hinweis nachgeschoben, dass GitFlow für Anwendungen mit laufender Auslieferung nicht die richtige Wahl sei — genau der Fall, um den es in dieser Unterlage geht.

### 4.6.2 GitHub Flow

Ein dauerhafter Zweig `main`, der jederzeit auslieferbar ist. Kurzlebige Feature-Branches, Pull Request, Zusammenführung, Auslieferung.

```
main  --?---?---?---?---?---?---?-->  jederzeit auslieferbar
        \     /     \     /
feature  ?---?       ?---?
         (1-2 Tage)
```

**Passt zu:** Webanwendungen, internen Diensten, allem mit einer einzigen produktiven Version. Für den KFZ-Rechner ist das die richtige Wahl.

### 4.6.3 Trunk-based Development

Die konsequenteste Variante: Alle arbeiten am selben Zweig, Branches leben Stunden statt Tage. Unfertige Funktionen werden über **Feature Flags** verborgen statt in Zweigen versteckt.

```
main  --?-?-?-?-?-?-?-?-?-?-->   mehrere Commits pro Tag
         \_/   \_/
       wenige Stunden
```

**Passt zu:** erfahrenen Teams mit guter Testabdeckung. Es ist die Strategie mit den nachweislich besten Werten in der DORA-Forschung — setzt aber Vertrauen in die eigene Testautomatisierung voraus.

### 4.6.4 Gegenüberstellung

| | GitFlow | GitHub Flow | Trunk-based |
|---|---|---|---|
| Lebensdauer eines Zweigs | Wochen | 1–2 Tage | Stunden |
| Konfliktrisiko | hoch | gering | sehr gering |
| Eignung für CI | schlecht | gut | ideal |
| Nötige Testabdeckung | mittel | gut | sehr gut |
| Mehrere Versionen parallel | ja | nein | nein |
| Einstiegshürde | hoch | niedrig | mittel |

**Wichtig**

**Der Kern der Sache:** Continuous Integration bedeutet wörtlich *fortlaufendes Zusammenführen*. Ein Feature-Branch, der drei Wochen lebt, ist genau das Gegenteil — dort integriert niemand, dort divergiert etwas.
Je länger ein Zweig lebt, desto größer der Unterschied zum Hauptzweig, desto schmerzhafter die Zusammenführung. Teams, die wochenlange Branches führen, verbringen einen erheblichen Teil ihrer Zeit mit Konfliktauflösung und nennen das dann Entwicklung.
**Faustregel: Kein Zweig lebt länger als zwei Tage.** Was länger dauert, ist zu groß geschnitten.

## 4.7 Code Review

Der Pull Request — bei GitLab *Merge Request* — ist mehr als eine Formalie. Er erfüllt drei Zwecke gleichzeitig: Qualitätssicherung, Wissensverteilung und, in regulierten Umgebungen, den Nachweis des Vier-Augen-Prinzips. Dazu mehr in Kapitel 17.

**Was ein Prüfer tun sollte:**

- innerhalb weniger Stunden reagieren — ein liegengebliebener Pull Request blockiert den Fluss
- auf Verständlichkeit, Testabdeckung und Sicherheit achten
- Fragen stellen statt Urteile fällen: „Was passiert hier bei einer leeren Liste?"
- Kleinigkeiten als solche kennzeichnen: „Randbemerkung, nicht blockierend"

**Was ein Prüfer nicht tun sollte:**

- Formatierung diskutieren — dafür gibt es Formatierungswerkzeuge, die das automatisch erledigen
- die gesamte Architektur in Frage stellen, wenn das im Entwurf schon geklärt war
- einen Pull Request mit 2.000 geänderten Zeilen ernsthaft prüfen wollen. Ab etwa 400 Zeilen sinkt die Fehlerentdeckungsrate messbar — geprüft wird dann nur noch pro forma

## 4.8 Lab: Der KFZ-Rechner kommt unter Versionskontrolle

```
# 1. Repository anlegen
mkdir kfz-rechner && cd kfz-rechner
git init -b main

# 2. Struktur erzeugen
mkdir -p src tests docs
cat > src/praemie.py <<'EOF'
"""Prämienberechnung für die KFZ-Versicherung."""
from decimal import Decimal

GRUNDBEITRAG = Decimal("480.00")

SF_RABATT = {
    0: Decimal("1.00"),
    1: Decimal("0.85"),
    5: Decimal("0.60"),
    10: Decimal("0.40"),
}


def berechne_praemie(sf_klasse: int, fahrzeugalter: int) -> Decimal:
    """Errechnet den Jahresbeitrag in Euro."""
    if sf_klasse < 0:
        raise ValueError("SF-Klasse darf nicht negativ sein")

    faktor = SF_RABATT.get(sf_klasse, Decimal("0.50"))
    beitrag = GRUNDBEITRAG * faktor

    if fahrzeugalter > 10:
        beitrag *= Decimal("0.90")

    return beitrag.quantize(Decimal("0.01"))
EOF

# 3. .gitignore anlegen
cat > .gitignore <<'EOF'
__pycache__/
*.pyc
.venv/
.env
EOF

# 4. Erster Commit
git add .
git commit -m "feat: Prämienberechnung als erste Fassung

Grundbeitrag mit SF-Rabatt und Altersnachlass.
Dezimaltyp statt Fließkomma, um Rundungsdifferenzen
bei der Beitragsberechnung auszuschließen."

# 5. Feature-Branch, Änderung, Zusammenführung
```

```
git switch -c feat/wenigfahrer
# ... Rabatt für Wenigfahrer ergänzen ...
git add -p
git commit -m "feat(praemie): Rabatt für Wenigfahrer ergänzen"
git switch main
git merge --no-ff feat/wenigfahrer
git branch -d feat/wenigfahrer

# 6. Version markieren
git tag -a v0.1.0 -m "Erste lauffähige Fassung"
git log --oneline --graph --all
```

**Zusatzaufgabe:** Lege das Repository auf einem Git-Server an (Gitea, GitLab oder GitHub) und richte einen Pull Request ein. Lass jemanden aus deinem Umfeld eine Anmerkung hinterlassen und arbeite sie ein.

## 4.9 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Langlebige Feature-Branches | Zusammenführungskonflikte, keine echte Integration | Zweige unter zwei Tagen, Feature Flags |
| Sammelcommits | „Änderungen an 47 Dateien"— nicht prüfbar, nicht zurückrollbar | Kleine, thematische Commits |
| Direkt auf `main` pushen | Prüfung entfällt, Historie wird unsauber | Branch-Schutz aktivieren |
| Geheimnisse im Repository | Kompromittierung, oft jahrelang unbemerkt | Secret-Scanning, `.gitignore`, Tresor |
| Build-Ergebnisse eingecheckt | Repository wird riesig, ständige Konflikte | `.gitignore`, Artefakt-Registry |
| `-force` auf gemeinsame Zweige | überschreibt fremde Arbeit | `-force-with-lease`, besser gar nicht |
| Riesige Pull Requests | Prüfung wird zur Formalie | Änderungen aufteilen |

## 4.10 Reflexionsfrage

*Wie lange lebt in deinem Umfeld der durchschnittliche Feature-Branch — und was wäre nötig, damit er nur noch zwei Tage lebt?*

Die Antwort führt fast immer zu denselben zwei Punkten: kleinere Arbeitspakete und bessere Tests. Womit wir beim nächsten Kapitel wären.

# Kapitel 5

# Testen und Qualität

**Lernziel**

Nach diesem Kapitel kennst du die Testpyramide und ihre Umkehrung als Anti-Pattern, kannst Shift-Left von Shift-Right unterscheiden und weißt, warum Testabdeckung eine irreführende Kennzahl ist. Du hast automatisierte Tests für den KFZ-Rechner geschrieben und kennst die datenschutzrechtlichen Grenzen bei Testdaten.

## 5.1 Warum Tests die Voraussetzung für Tempo sind

Es klingt widersprüchlich: Tests kosten Zeit, sollen aber schneller machen. Der Zusammenhang wird klar, wenn man fragt, was ohne sie passiert.

Ohne verlässliche Tests ist jede Auslieferung ein Risiko. Wer Risiken scheut, liefert seltener aus. Wer seltener ausliefert, packt mehr Änderungen in jede Auslieferung. Je mehr Änderungen, desto größer das Risiko. Ein sich selbst verstärkender Kreislauf, an dessen Ende das halbjährliche Release mit vierzehn Beteiligten und einem Wochenende Bereitschaft steht.

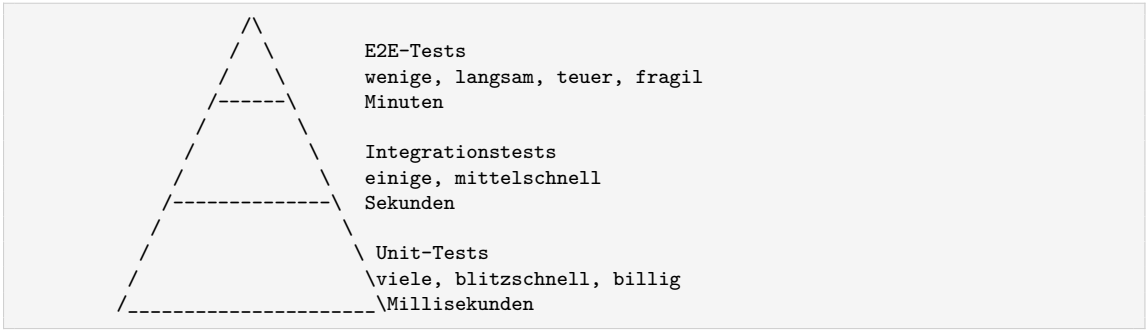
Automatisierte Tests durchbrechen ihn. Sie sind nicht Selbstzweck, sondern **die Erlaubnis, schnell zu sein**.

**Die entscheidende Frage lautet nicht „Haben wir Tests?", sondern:**

*Trauen wir uns, am Freitagnachmittag auszuliefern?*
Wer diese Frage mit Ja beantworten kann, hat gute Tests. Wer sie mit Nein beantwortet, hat unabhängig von jeder Abdeckungszahl ein Testproblem. Das ist übrigens auch ein brauchbarer Reifegradindikator für Bewerbungsgespräche — in beide Richtungen.

## 5.2 Die Testpyramide

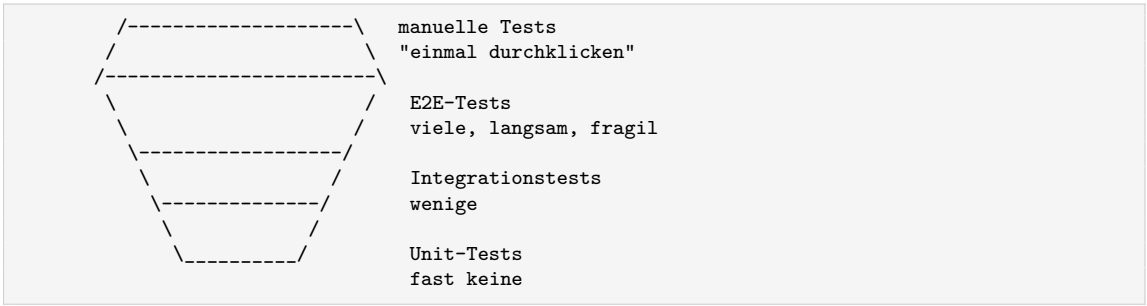
Das Modell geht auf Mike Cohn zurück und ordnet Tests nach Umfang, Geschwindigkeit und Kosten.




| Ebene | Prüft | Anteil | Laufzeit |
|---|---|---|---|
| Unit | eine Funktion isoliert | 70 % | Millisekunden |
| Integration | Zusammenspiel mehrerer Teile, echte Datenbank | 20 % | Sekunden |
| End-to-End | den gesamten Weg wie ein Anwender | 10 % | Minuten |

Die Logik dahinter ist ökonomisch: Ein Unit-Test findet einen Fehler in Millisekunden und zeigt exakt die betroffene Zeile. Ein E2E-Test findet denselben Fehler nach vier Minuten und meldet lediglich, dass irgendetwas nicht stimmt.

### 5.2.1 Das Anti-Pattern: die Eistüte




Die umgedrehte Pyramide — der *Ice Cream Cone* — entsteht nie durch eine Entscheidung, sondern durch Unterlassung: Man testet am Ende über die Oberfläche, weil es dort ohne Vorarbeit möglich ist. Das Ergebnis ist eine Testsuite, die zwanzig Minuten braucht, sporadisch grundlos fehlschlägt und nach jeder Layout-Änderung repariert werden muss.

## 5.3 Testabdeckung — eine gefährliche Kennzahl

Die Abdeckung misst, welcher Anteil des Codes bei einem Testlauf ausgeführt wurde. Sie misst **nicht**, ob dabei etwas Sinnvolles geprüft wurde.

```python
# Dieser Test erzeugt 100 % Abdeckung und prüft nichts
def test_praemie():
    berechne_praemie(5, 3)      # keine Zusicherung, kein assert
```

Nützlich ist die Abdeckung, um **ungetestete Bereiche zu finden** — eine Datei mit 0 Prozent ist ein echtes Signal. Schädlich wird sie als Zielvorgabe: Sobald 80 Prozent verlangt sind, entstehen Tests, die Abdeckung erzeugen statt Fehler zu finden. Das ist ein Lehrbuchbeispiel für Goodharts Gesetz — wird eine Kennzahl zum Ziel, verliert sie ihre Aussagekraft.

**Tipp**

Sinnvoller als eine Abdeckungsquote ist die Frage: **Welche Fehler hat unsere Testsuite in den letzten sechs Monaten tatsächlich verhindert?** Und die Gegenprobe: Welche Störungen sind trotz grüner Tests in Produktion aufgetreten? Jede solche Störung ist eine Einladung, genau dafür einen Test nachzuziehen.

## 5.4 Shift-Left

Shift-Left bedeutet, Qualitätssicherung so früh wie möglich stattfinden zu lassen — im Zeitstrahl also nach links.

```
Klassisch:
Anforderung -- Entwurf -- Coding ---------------- TEST -- Release
                                                    ^
                                              alles am Ende

Shift-Left:
Anforderung -- Entwurf -- Coding -- Release
     ^           ^         ^         ^
   Test        Test      Test      Test
   (Kriterien  (Review)  (Unit, CI) (Canary)
    prüfbar?)
```

**Wichtig**

**Eine verbreitete Fehldarstellung:** Shift-Left bedeutet nicht, dass das Wasserfallmodell abgeschafft wird oder dass am Ende nicht mehr getestet wird. Es bedeutet, dass Qualitätssicherung **an jedem Schritt** stattfindet statt nur an einem.
Auch in einem Wasserfallprojekt kann man Anforderungen auf Prüfbarkeit abklopfen, Entwürfe im Review hinterfragen und Unit-Tests schreiben. Shift-Left ist eine Frage der Verteilung, nicht des Vorgehensmodells.

Konkrete Maßnahmen, von links nach rechts:

| Phase | Qualitätsmaßnahme |
|---|---|
| Anforderung | Akzeptanzkriterien formulieren: „Woran erkennen wir, dass es funktioniert?" |
| Entwurf | Bedrohungsmodellierung, Prüfung auf Nachvollziehbarkeit |
| Coding | Unit-Tests, Linting, Formatierung, Pre-Commit-Hooks |
| Commit | automatischer Testlauf, Secret-Scanning |
| Build | Abhängigkeiten auf Schwachstellen prüfen |
| Vor dem Ausrollen | Integrations- und E2E-Tests, Lasttest |

## 5.5 Shift-Right

Das Gegenstück wird selten mitgelehrt, ist aber ebenso wichtig: Manche Eigenschaften lassen sich **nur in Produktion** feststellen. Echte Last, echte Daten, echtes Nutzerverhalten, echte Netzwerke.

| Verfahren | Was es leistet |
|---|---|
| **Canary Release** | Neue Version zunächst für 5 % der Anfragen, Fehlerrate beobachten |
| **Synthetisches Monitoring** | Ein Roboter durchläuft alle fünf Minuten den Anmeldevorgang |
| **Feature Flags** | Funktion in Produktion vorhanden, aber abgeschaltet — gezielt zuschaltbar |
| **Chaos Engineering** | Kontrolliert Störungen erzeugen, um Annahmen zu prüfen |
| **A/B-Test** | Zwei Varianten parallel, Entscheidung anhand von Daten |

Chaos Engineering klingt nach mutwilliger Zerstörung, ist aber das genaue Gegenteil: Man prüft kontrolliert und zu einer Zeit, in der alle Beteiligten wach und ansprechbar sind, ob die Annahmen über die Ausfallsicherheit zutreffen. Die Alternative besteht darin, dieselbe Erkenntnis unkontrolliert um drei Uhr nachts zu gewinnen.

## 5.6 Testdaten und Datenschutz

Ein Abschnitt, der in Versicherungs- und Bankumgebungen über Wohl und Wehe entscheidet.

**Achtung**

**Produktivdaten gehören nicht in Testsysteme.** Ein Abzug der Vertragsdatenbank auf das Testsystem ist eine Verarbeitung personenbezogener Daten zu einem anderen Zweck als dem der Erhebung — sie braucht eine eigene Rechtsgrundlage, die in aller Regel nicht vorliegt.

Verschärfend kommt hinzu: Testsysteme haben schwächere Zugriffskontrollen, mehr Berechtigte, seltener eingespielte Patches und selten eine Protokollierung, die einer Prüfung standhält. Ein Datenabfluss aus dem Testsystem ist genauso meldepflichtig wie einer aus der Produktion.

Die sauberen Alternativen:

| Verfahren | Beschreibung | Eignung |
|---|---|---|
| **Synthetische Daten** | vollständig erfunden, per Generator erzeugt | erste Wahl |
| **Anonymisierung** | Personenbezug unwiderruflich entfernt | gut, wenn wirklich unwiderruflich |
| **Pseudonymisierung** | Bezug über Schlüssel wiederherstellbar | weiterhin personenbezogene Daten |
| **Maskierung** | Felder überschrieben, Struktur erhalten | brauchbar bei sorgfältiger Umsetzung |

```python
# Synthetische Testdaten mit faker
from faker import Faker

fake = Faker("de_DE")

testkunden = [
    {
        "name": fake.name(),
        "plz": fake.postcode(),
        "sf_klasse": fake.random_int(0, 35),
        "fahrzeugalter": fake.random_int(0, 25),
    }
    for _ in range(1000)
]
```

**Ein Hinweis zur Anonymisierung:** Sie ist schwerer, als sie aussieht. Eine Kombination aus Postleitzahl, Geburtsdatum und Geschlecht identifiziert einen erheblichen Teil der Bevölkerung eindeutig, auch ohne Namen. Wer anonymisiert, sollte das Ergebnis prüfen lassen, nicht nur die Namensspalte überschreiben.

## 5.7 Lab: Tests für den KFZ-Rechner

```bash
cd kfz-rechner
python3 -m venv .venv && source .venv/bin/activate
pip install pytest pytest-cov
```

tests/test_praemie.py

```python
"""Tests für die Prämienberechnung."""
from decimal import Decimal

import pytest

from src.praemie import berechne_praemie
```

```python
def test_grundfall():
    """SF-Klasse 0 ohne Altersnachlass ergibt den vollen Grundbeitrag."""
    assert berechne_praemie(0, 3) == Decimal("480.00")


def test_sf_rabatt():
    """SF-Klasse 5 gewährt 40 Prozent Nachlass."""
    assert berechne_praemie(5, 3) == Decimal("288.00")


def test_altersnachlass():
    """Fahrzeuge über zehn Jahre erhalten zusätzlich zehn Prozent."""
    assert berechne_praemie(5, 12) == Decimal("259.20")


def test_unbekannte_sf_klasse():
    """Nicht hinterlegte SF-Klassen fallen auf den Standardfaktor zurück."""
    assert berechne_praemie(7, 3) == Decimal("240.00")


def test_negative_sf_klasse_wirft_fehler():
    """Negative SF-Klassen sind fachlich unmöglich."""
    with pytest.raises(ValueError, match="negativ"):
        berechne_praemie(-1, 3)


@pytest.mark.parametrize(
    "sf,alter,erwartet",
    [
        (0, 0, "480.00"),
        (1, 0, "408.00"),
        (10, 0, "192.00"),
        (10, 15, "172.80"),
    ],
)
def test_verschiedene_kombinationen(sf, alter, erwartet):
    """Prüft mehrere Kombinationen in einem Durchlauf."""
    assert berechne_praemie(sf, alter) == Decimal(erwartet)


def test_ergebnis_hat_zwei_nachkommastellen():
    """Beträge werden auf Cent gerundet."""
    ergebnis = berechne_praemie(5, 3)
    assert ergebnis.as_tuple().exponent == -2
```

```
pytest -v
pytest --cov=src --cov-report=term-missing
```

**Beobachte dabei zwei Dinge:** Wie lange der gesamte Lauf dauert — bei Unit-Tests sollten es deutlich unter zwei Sekunden sein. Und was die Abdeckungsausgabe unter `Missing` anzeigt: Das sind die Zeilen, die nie ausgeführt wurden.

**Zusatzaufgaben:**

1. Ergänze eine Funktion für einen Wenigfahrer-Rabatt und schreibe den Test **zuerst**. Lass ihn fehlschlagen, dann implementiere. Das ist testgetriebene Entwicklung im Kleinen.
2. Baue absichtlich einen Fehler ein — etwa ein Vorzeichen — und prüfe, ob die Tests ihn finden. Falls nicht: Der Test fehlt.

3. Richte einen Pre-Commit-Hook ein, der `pytest` vor jedem Commit ausführt.

.git/hooks/pre-commit

```
#!/usr/bin/env bash
set -e
echo "Tests laufen ..."
pytest -q || {
    echo "Commit abgebrochen: Tests fehlgeschlagen."
    exit 1
}
```

```
chmod +x .git/hooks/pre-commit
```

## 5.8 Flaky Tests

Ein Test, der bei unverändertem Code mal durchläuft und mal fehlschlägt, ist schlimmer als kein Test — er zerstört das Vertrauen in die gesamte Suite. Sobald ein Team anfängt, fehlgeschlagene Läufe reflexartig zu wiederholen, ist die Testsuite als Sicherheitsnetz erledigt.

| **Häufige Ursache** | **Abhilfe** |
|---|---|
| Zeitabhängigkeit, echte Uhr | Zeit als Parameter übergeben oder einfrieren |
| Feste Wartezeiten (`sleep 2`) | auf Bedingungen warten statt auf Sekunden |
| Reihenfolgeabhängigkeit | Tests unabhängig machen, Zustand zurücksetzen |
| Gemeinsame Testdatenbank | je Lauf eigenes Schema oder eigener Container |
| Zufallswerte ohne festen Startwert | Startwert setzen und protokollieren |
| Echte Netzwerkaufrufe | Doubles oder lokale Testdienste verwenden |

**Regel:** Ein sporadisch fehlschlagender Test wird sofort repariert oder abgeschaltet. Der Mittelweg — ihn stehen lassen und ignorieren — ist der einzige, der garantiert schadet.

## 5.9 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Eistüte statt Pyramide | Testlauf dauert 20 Minuten, ist unzuverlässig | Prüfungen nach unten verlagern |
| Abdeckung als Zielvorgabe | Tests ohne Zusicherungen | Abdeckung beobachten, nicht vorschreiben |
| Produktivdaten im Test | Datenschutzverstoß | synthetische Daten |
| Tests erst am Projektende | findet Fehler, wenn ihre Behebung am teuersten ist | Shift-Left |
| Sporadische Fehlschläge geduldet | Testsuite verliert ihre Funktion | sofort reparieren |
| Manuelle Regressionstests | skaliert nicht, ermüdet, wird übersprungen | automatisieren |
| Nur der Gutfall wird getestet | Fehlerbehandlung bricht in Produktion | Randfälle und Fehlerfälle mitnehmen |

## 5.10 Reflexionsfrage

*Wenn du in deinem Betrieb eine einzelne Zeile Code änderst — wie lange dauert es, bis du weißt, ob du etwas kaputt gemacht hast?*

Minuten sind gut. Stunden sind machbar. Tage bedeuten, dass die Rückmeldeschleife aus Kapitel 2 fehlt. Und „wir merken es, wenn jemand anruftïst die Antwort, die dieses Kapitel verhindern soll.

# Teil III

# Teil C — Wie man ausliefert

# Kapitel 6

# Continuous Integration

**Lernziel**

Nach diesem Kapitel kannst du erklären, was Continuous Integration von „wir haben einen Buildserverünterscheidet, kennst die vier Voraussetzungen und die Zehn-Minuten-Regel. Du hast eine lauffähige CI-Pipeline für den KFZ-Rechner gebaut.

## 6.1 Was Continuous Integration wirklich bedeutet

Der Begriff wird fast immer zu eng verstanden. Viele Teams sagen „wir machen CIünd meinen: Es gibt einen Server, der nach einem Push Tests ausführt.

Das ist die Werkzeugseite. Der eigentliche Inhalt steckt im Wort **Integration**: Alle Entwickler führen ihre Arbeit **fortlaufend** — mindestens täglich — im Hauptzweig zusammen, und jede Zusammenführung wird automatisch überprüft.

**Die Prüffrage lautet nicht „Habt ihr Jenkins?", sondern:**

*Wann hat der letzte Entwickler seine Arbeit zuletzt in* `main` *integriert?*
Lautet die Antwort „vor drei Wochen, er ist noch auf seinem Branch", dann findet unabhängig von jeder installierten Software **keine** Continuous Integration statt. Es gibt dann lediglich einen Server, der jeden Zweig für sich prüft, während die Zweige immer weiter auseinanderlaufen.

## 6.2 Die vier Voraussetzungen

1. **Alles liegt in der Versionsverwaltung.** Code, Tests, Pipeline-Definition, Infrastrukturbeschreibung, Datenbankmigrationen. Alles, was zum Bauen und Betreiben nötig ist — mit Ausnahme der Geheimnisse.
2. **Der Build läuft automatisch.** Ein Befehl, keine Handgriffe, keine Anleitung mit dreißig Schritten.
3. **Die Tests laufen automatisch.** Aus Kapitel 4 — und zwar bei jedem Commit, nicht nachts.
4. **Die Rückmeldung kommt schnell.** Unter zehn Minuten. Sonst arbeitet niemand damit.

## 6.3 Die Zehn-Minuten-Regel

Warum ausgerechnet zehn Minuten? Weil das die Zeitspanne ist, in der ein Mensch bei der Sache bleibt.

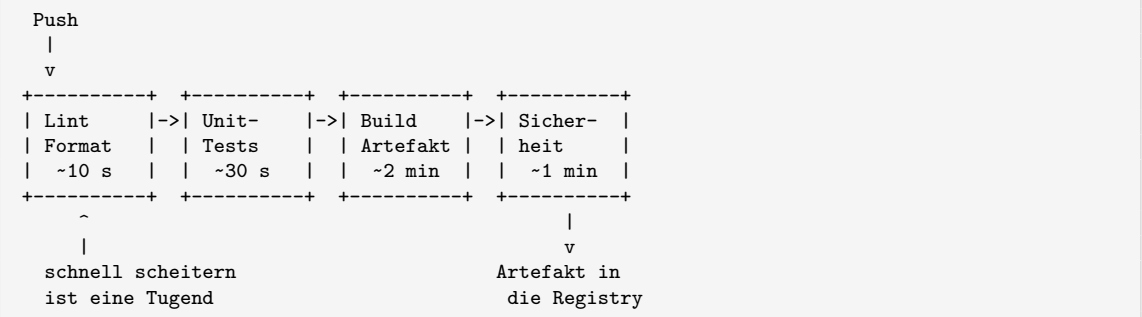
| Dauer der Pipeline | Was der Entwickler tut |
|---|---|
| unter 2 min | wartet und schaut zu |
| 2–10 min | holt sich Kaffee, kommt zurück, arbeitet weiter |
| 10–30 min | fängt etwas Neues an — der Kontext ist weg |
| über 30 min | schaut abends nach, vielleicht |
| über 60 min | die Pipeline ist ein Ritual, kein Werkzeug |

Ab etwa zwanzig Minuten kippt das Verhalten: Man wartet die Rückmeldung nicht mehr ab, sondern arbeitet weiter — und stapelt Änderungen auf einer möglicherweise kaputten Grundlage. Die Pipeline erfüllt ihren Zweck dann nicht mehr.

**Was hilft, wenn es zu lang dauert:**

- Schritte parallelisieren, die voneinander unabhängig sind
- Abhängigkeiten und Container-Schichten zwischenspeichern
- Schnelle Prüfungen zuerst — Linting vor Unit-Tests vor Integrationstests
- Langsame E2E-Tests aus der Commit-Pipeline herausnehmen und nachgelagert ausführen
- bei Monorepos nur bauen, was sich geändert hat

## 6.4 Der Aufbau einer Commit-Pipeline




Das Prinzip heißt **fail fast**: Die billigsten und schnellsten Prüfungen kommen zuerst. Es ergibt keinen Sinn, acht Minuten lang ein Container-Image zu bauen, wenn der Code einen Syntaxfehler enthält, den ein Linter in vier Sekunden findet.

## 6.5 Der gebrochene Hauptzweig

Aus der Fertigungsindustrie stammt das Bild der Reißleine am Fließband: Wer einen Fehler bemerkt, hält das gesamte Band an. Was zunächst absurd teuer klingt, ist billiger als die Alternative — nämlich hundert fehlerhafte Teile weiterlaufen zu lassen.

Übertragen auf die Softwareentwicklung:

**Ein roter Build auf ”main” hat Vorrang vor allem anderen.**

Solange `main` rot ist, kann niemand verlässlich integrieren, niemand ausliefern und niemand wissen, ob der eigene Fehler neu ist oder von der Vorgängerin stammt. Jede weitere Zusammenführung baut auf unsicherem Grund auf.
Praktische Regeln, die sich bewährt haben:

- Wer den Build bricht, repariert ihn — sofort, nicht morgen.
- Gelingt die Reparatur nicht binnen etwa zehn Minuten, wird der auslösende Commit zurückgenommen. Das ist keine Kränkung, sondern Betriebshygiene.
- Niemand geht mit rotem `main` in den Feierabend.

Das ist übrigens der Punkt, an dem sich Kultur und Technik berühren: Die Regel funktioniert nur in einem Umfeld, in dem ein gebrochener Build als normaler Vorgang gilt und nicht als persönliches Versagen. Wo der Verursacher mit spitzen Bemerkungen rechnen muss, wird niemand mehr häufig integrieren — und damit ist die Continuous Integration erledigt, bevor sie begonnen hat.

## 6.6 Lab: CI-Pipeline für den KFZ-Rechner

Beide gängigen Systeme im Vergleich — nimm das, was bei dir verfügbar ist.

### 6.6.1 Variante GitLab CI

.gitlab-ci.yml

```
stages:
  - pruefen
  - testen
  - bauen

default:
  image: python:3.12-slim
  cache:
    key:
      files:
        - requirements.txt
    paths:
      - .venv/

.python_vorbereiten: &python_vorbereiten
  before_script:
    - python -m venv .venv
    - source .venv/bin/activate
```

```
    - pip install --quiet --upgrade pip
    - pip install --quiet -r requirements.txt

# ---------- Stufe 1: schnelle Prüfungen ----------

lint:
  stage: pruefen
  <<: *python_vorbereiten
  script:
    - ruff check src/ tests/
    - ruff format --check src/ tests/

# ---------- Stufe 2: Tests ----------

unit-tests:
  stage: testen
  <<: *python_vorbereiten
  script:
    - pytest --junitxml=report.xml --cov=src --cov-report=xml
  coverage: '/TOTAL.*\s+(\d+%)$/'
  artifacts:
    when: always
    reports:
      junit: report.xml
      coverage_report:
        coverage_format: cobertura
        path: coverage.xml
    expire_in: 1 week

# ---------- Stufe 3: Artefakt ----------

container-bauen:
  stage: bauen
  image: quay.io/podman/stable
  before_script:
    - podman login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
  script:
    - podman build -t "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA" .
    - podman push "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA"
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

### 6.6.2 Variante GitHub Actions

.github/workflows/ci.yml

```
name: CI

on:
  push:
    branches: [main]
  pull_request:

jobs:
  pruefen:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: actions/setup-python@v5
        with:
          python-version: "3.12"
          cache: pip
```

```
      - name: Abhängigkeiten installieren
        run: pip install -r requirements.txt

      - name: Linting
        run: |
          ruff check src/ tests/
          ruff format --check src/ tests/

      - name: Tests
        run: pytest --cov=src --cov-report=term-missing

  bauen:
    needs: pruefen
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - uses: actions/checkout@v4

      - name: An der Registry anmelden
        uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Image bauen und hochladen
        uses: docker/build-push-action@v5
        with:
          push: true
          tags: ghcr.io/${{ github.repository }}:${{ github.sha }}
```

### 6.6.3 Das zugehörige Containerfile

Containerfile

```
FROM registry.access.redhat.com/ubi9/python-312:latest

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY src/ ./src/

USER 1001
EXPOSE 8080

CMD ["python", "-m", "src.server"]
```

### 6.6.4 Aufgaben

1. Bring die Pipeline zum Laufen und beobachte die Gesamtdauer.
2. Baue einen Formatierungsfehler ein. Welche Stufe schlägt fehl, nach wie vielen Sekunden?

3. Baue einen fachlichen Fehler ein — etwa einen falschen Rabattfaktor. Welche Stufe fängt ihn?
4. Miss die Laufzeit mit und ohne Zwischenspeicher der Abhängigkeiten.
5. Aktiviere den Branch-Schutz auf `main`, sodass nur zusammengeführt werden kann, wenn die Pipeline grün ist.

**Tipp**

Der letzte Punkt ist der wichtigste. Eine Pipeline, deren Ergebnis man ignorieren kann, ist Dekoration. Erst der Branch-Schutz macht aus der Prüfung eine Zusage — und nimmt gleichzeitig dem einzelnen Entwickler die unangenehme Rolle, Kollegen auf rote Builds ansprechen zu müssen. Das erledigt jetzt die Maschine, und Maschinen nehmen einem das nicht übel.

## 6.7 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Nächtlicher Build | Rückmeldung nach 14 Stunden | bei jedem Push bauen |
| Roter `main` wird toleriert | niemand weiß, was funktioniert | Reißleine, Revert-Regel |
| Pipeline dauert 45 Minuten | wird nicht mehr abgewartet | parallelisieren, zwischenspeichern, aufteilen |
| Tests werden übersprungen, „ist eilig" | genau dann geht es schief | keine Ausnahmen, technisch erzwingen |
| CI baut, aber niemand schaut hin | Fehler bleiben liegen | Benachrichtigung, Branch-Schutz |
| Pipeline nur auf Feature-Branches | Integration findet nie statt | `main` ist der Prüfstein |
| Pipeline im Web-Formular zusammengeklickt | nicht versioniert, nicht nachvollziehbar | Pipeline as Code |

## 6.8 Reflexionsfrage

*Wie lange dauert bei euch der Weg vom Push bis zur Rückmeldung „alles in Ordnung" — und wie oft wird diese Rückmeldung tatsächlich abgewartet?*

# Kapitel 7

# Continuous Delivery und Deployment

**Lernziel**

Nach diesem Kapitel kannst du CI, Continuous Delivery und Continuous Deployment sauber gegeneinander abgrenzen und erklären, warum der Unterschied zwischen den beiden letzten eine Geschäftsentscheidung ist. Du kennst das Prinzip *einmal bauen, überall ausrollen*, verstehst die Trennung von Deployment und Release und weißt, warum Datenbankmigrationen das eigentliche Hindernis sind.

## 7.1 Die drei Begriffe

```
Continuous Integration
  Code wird laufend zusammengeführt, gebaut und getestet.
  Ergebnis: ein geprüftes Artefakt.
         |
         v
Continuous Delivery
  Das Artefakt ist jederzeit auslieferbar.
  Der Weg in die Produktion ist automatisiert und erprobt.
  Ausgelöst wird er durch einen Menschen.
         |
         v
Continuous Deployment
  Jede Änderung, die alle Prüfungen besteht, geht automatisch
  in Produktion. Ohne Knopfdruck.
```

| | **CI** | **Continuous Delivery** | **Continuous Deployment** |
|---|---|---|---|
| Gebaut und getestet | automatisch | automatisch | automatisch |
| In Testumgebungen | teilweise | automatisch | automatisch |
| In Produktion | nein | **Knopfdruck** | **automatisch** |
| Entscheidung über den Zeitpunkt | — | Mensch | Pipeline |

**Wichtig**

**Der Unterschied zwischen Delivery und Deployment ist technisch minimal — es fehlt genau ein manueller Schritt.** Es ist deshalb keine technische, sondern eine **geschäftliche und regulatorische** Entscheidung.
Für einen internen Prämienrechner kann Continuous Deployment vollkommen angemessen sein. Für ein System, dessen Änderungen der Fachbereich freigeben muss oder bei dem eine Aufsichtsbehörde die Genehmigung sehen will, ist Continuous Delivery das Ziel — der Knopf existiert, jemand muss ihn drücken, und dieses Drücken ist protokolliert. Siehe Kapitel 17.
Wichtig ist die Reihenfolge: **Continuous Delivery ist die anspruchsvolle Leistung.** Wer sie beherrscht, kann jederzeit entscheiden, den letzten Schritt zu automatisieren. Wer sie nicht beherrscht, gewinnt durch das Weglassen des Knopfes gar nichts.

## 7.2 Einmal bauen, überall ausrollen

Das vielleicht wichtigste technische Prinzip dieses Kapitels.

```
FALSCH                              RICHTIG

Code --> Build --> Test-Umgebung    Code --> Build --> Artefakt
Code --> Build --> Abnahme                             |
Code --> Build --> Produktion                          +--> Test
                                                       +--> Abnahme
Drei verschiedene Ergebnisse.                          +--> Produktion
Getestet wurde nie das,
was produktiv läuft.                 Ein Artefakt. Überall dasselbe.
```

Wird für jede Umgebung neu gebaut, hat man drei verschiedene Artefakte. Zwischen dem Bauen für die Abnahme und dem Bauen für die Produktion kann sich eine Abhängigkeit aktualisiert haben, ein Basis-Image geändert, ein Zeitstempel unterschieden. Getestet wurde dann streng genommen etwas anderes als das, was in Produktion läuft — und genau dieser Unterschied ist die Sorte Fehler, die man am Freitagabend sucht.

**Die Konsequenz:** Die Konfiguration darf nicht im Artefakt stecken. Sie kommt von außen.

```
# Dasselbe Image, drei Umgebungen
podman run -e DB_HOST=db-test.intern    kfz-rechner:2.4.0
podman run -e DB_HOST=db-abnahme.intern kfz-rechner:2.4.0
podman run -e DB_HOST=db-prod.intern    kfz-rechner:2.4.0
```

Dieses Prinzip stammt aus der **Zwölf-Faktoren-Methodik**, deren dritter Punkt genau das fordert: Konfiguration gehört in die Umgebung, nicht in den Code.

## 7.3 Deployment ist nicht Release

Zwei Begriffe, die im Alltag synonym gebraucht werden, obwohl ihre Trennung eine der wirkungsvollsten Techniken überhaupt ist.

| Begriff | Bedeutung | Entscheidet |
|---|---|---|
| **Deployment** | neue Version läuft auf den Servern | Technik |
| **Release** | Anwender können die Funktion nutzen | Fachbereich |

Solange beides zusammenfällt, ist jede Auslieferung ein Ereignis mit Publikum. Entkoppelt man sie, wird das Deployment zum Nichtereignis:

```
# Die Funktion ist ausgerollt, aber abgeschaltet
if feature_aktiv("wenigfahrer_rabatt"):
    beitrag = wende_wenigfahrer_rabatt_an(beitrag)
```

Der Code geht am Dienstag in Produktion, ohne dass ihn jemand bemerkt. Am Donnerstag schaltet der Fachbereich die Funktion frei — für zunächst fünf Prozent der Anfragen. Fällt etwas auf, wird sie ausgeschaltet. Das dauert Sekunden und braucht kein Deployment, kein Wartungsfenster und keine Nachtschicht.

Mehr zu Feature Flags in Kapitel 8.

## 7.4 Umgebungen und ihre Tücken

| Umgebung | Zweck | Datenbestand |
|---|---|---|
| Entwicklung | lokales Arbeiten | synthetisch, klein |
| Integration/Test | automatisierte Tests | synthetisch, reproduzierbar |
| Abnahme | fachliche Prüfung, Schulung | synthetisch, realistisch im Umfang |
| Produktion | Echtbetrieb | echt |

**Schneeflockenumgebungen sind das häufigste Hindernis auf dem Weg zu Continuous Delivery.**

Eine Schneeflocke ist ein System, das über Jahre von Hand gepflegt wurde und das niemand mehr identisch nachbauen kann. Typisches Symptom: „Auf der Abnahme läuft es, in Produktion nicht"— und niemand kann sagen, worin sich die beiden unterscheiden. Die Gegenmaßnahme heißt Infrastructure as Code (Kapitel 9) und Containerisierung (Kapitel 10). Der Prüfstein ist unbequem, aber eindeutig: **Könnt ihr eure Abnahmeumgebung löschen und binnen einer Stunde identisch neu erzeugen?**

Vollständige Umgebungsparität ist dabei eine Illusion — die Produktion hat mehr Last, mehr Daten, echte Netzwerke und echte Anwender. Ziel ist nicht Gleichheit, sondern **kontrollierte, bekannte Unterschiede**. Ein Unterschied, den man kennt und dokumentiert hat, ist beherrschbar. Einer, den man erst im Störungsfall entdeckt, nicht.

## 7.5 Datenbankmigrationen — das eigentliche Problem

Anwendungscode lässt sich in Sekunden austauschen und ebenso schnell zurückrollen. Bei Datenbanken gilt das nicht: Eine gelöschte Spalte ist gelöscht, und die Daten darin sind es auch.

Das ist der Punkt, an dem Continuous Delivery in der Praxis am häufigsten scheitert.

### 7.5.1 Das Expand-Contract-Verfahren

Die Lösung besteht darin, Schemaänderungen in mehrere rückwärtskompatible Schritte zu zerlegen.

**Aufgabenstellung:** Die Spalte `sf_klasse` soll in `schadenfreiheitsklasse` umbenannt werden.

```
Schritt 1 -- Erweitern (expand)
  Neue Spalte ergänzen. Die alte bleibt.
  ALTER TABLE vertrag ADD COLUMN schadenfreiheitsklasse INT;
  -> Alte und neue Anwendungsversion funktionieren beide.

Schritt 2 -- Doppelt schreiben
  Anwendung schreibt in beide Spalten, liest aus der alten.
  -> Ausrollbar und zurückrollbar.

Schritt 3 -- Daten übertragen
  UPDATE vertrag SET schadenfreiheitsklasse = sf_klasse
   WHERE schadenfreiheitsklasse IS NULL;

Schritt 4 -- Umschalten
  Anwendung liest aus der neuen Spalte, schreibt weiter beide.
  -> Immer noch zurückrollbar.

Schritt 5 -- Zusammenziehen (contract)
  Erst wenn keine alte Version mehr läuft:
  ALTER TABLE vertrag DROP COLUMN sf_klasse;
```

Fünf Auslieferungen statt einer. Dafür ist jede einzelne davon gefahrlos und jederzeit zurücknehmbar.

**Die Grundregel für Datenbankänderungen bei laufender Auslieferung:**

Jede Migration muss mit der **vorherigen** Anwendungsversion verträglich sein. Sonst ist ein Rollback unmöglich — und ohne Rollback-Möglichkeit ist jede Auslieferung wieder ein Risiko, egal wie gut die Pipeline ist.
Praktisch heißt das: keine Spalten löschen, keine Spalten umbenennen, keine Typen verengen, solange noch eine ältere Version laufen könnte. Alles davon lässt sich als Expand-Contract-Folge abbilden.

Migrationen gehören dabei genauso in die Versionsverwaltung wie der Code — mit Werkzeugen wie **Flyway**, **Liquibase** oder **Alembic**, die durchnummerierte, einmalig ausgeführte Skripte verwalten.

## 7.6 Lab: Die Pipeline bekommt einen Ausrollschritt

.gitlab-ci.yml

```
stages:
  - pruefen
  - testen
  - bauen
  - ausrollen-test
  - ausrollen-prod

variables:
  IMAGE: "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA"

# ... Stufen aus Kapitel 5 ...

ausrollen-test:
  stage: ausrollen-test
  environment:
    name: test
    url: https://kfz-test.intern
  script:
    - ./scripts/deploy.sh test "$IMAGE"
    - ./scripts/smoketest.sh https://kfz-test.intern
  rules:
    - if: $CI_COMMIT_BRANCH == "main"

ausrollen-prod:
  stage: ausrollen-prod
  environment:
    name: produktion
    url: https://kfz.intern
  script:
    - ./scripts/deploy.sh produktion "$IMAGE"
    - ./scripts/smoketest.sh https://kfz.intern
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
      when: manual            # <- genau hier liegt der Unterschied
  allow_failure: false
```

**Beachte die Zeile `when: manual`.** Sie ist der gesamte Unterschied zwischen Continuous Delivery und Continuous Deployment. Nimmt man sie heraus, rollt jede geprüfte Änderung automatisch aus.

Der Rauchtest danach ist keine Formalie, sondern die Zusage, dass die neue Version tatsächlich antwortet:

scripts/smoketest.sh

```
#!/usr/bin/env bash
set -euo pipefail

URL="$1"
VERSUCHE=30

for i in $(seq 1 "$VERSUCHE"); do
    if curl -fsS --max-time 5 "${URL}/health" > /dev/null; then
        echo "Dienst antwortet nach ${i} Versuch(en)."

        # Fachliche Stichprobe: bekannter Eingabewert, bekanntes Ergebnis
        ergebnis=$(curl -fsS "${URL}/api/praemie?sf=5&alter=3" | jq -r '.betrag')
        if [[ "$ergebnis" != "288.00" ]]; then
            echo "FEHLER: Erwartet 288.00, erhalten ${ergebnis}"
```

```
            exit 1
        fi
        echo "Fachliche Prüfung bestanden."
        exit 0
    fi
    sleep 2
done

echo "FEHLER: Dienst antwortet nach ${VERSUCHE} Versuchen nicht."
exit 1
```

**Aufgaben:**

1. Ergänze eine Abnahmeumgebung zwischen Test und Produktion.
2. Sorge dafür, dass in allen Umgebungen dasselbe Image mit unterschiedlicher Konfiguration läuft.
3. Lass den Rauchtest absichtlich fehlschlagen und prüfe, dass die Pipeline rot wird.
4. Formuliere in einem Satz, was passieren muss, damit ihr `when: manual` streichen könnt.

## 7.7 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Pro Umgebung neu bauen | getestet wurde nie das Produktivartefakt | einmal bauen, überall ausrollen |
| Konfiguration im Artefakt | für jede Umgebung ein eigenes Image | Konfiguration von außen |
| Schneeflockenumgebungen | „läuft auf der Abnahme" | Infrastructure as Code |
| Migration ohne Rückwärtskompatibilität | Rollback unmöglich | Expand-Contract |
| Manuelle Schritte in der Anleitung | wird vergessen, wenn es eilt | vollständig automatisieren |
| Kein Rauchtest nach dem Ausrollen | Störung fällt erst Anwendern auf | automatische Prüfung |
| Freigabe als reine Formalie | Genehmigung ohne Kenntnis des Inhalts | Änderungsliste automatisch beilegen |

## 7.8 Reflexionsfrage

*Wenn ihr euer Produktivsystem heute Abend verlieren würdet — wie lange bräuchtet ihr, um es aus dem Repository heraus neu aufzubauen? Und wisst ihr das aus Erfahrung oder aus Vermutung?*

# Kapitel 8

# Pipeline in der Praxis

**Lernziel**

Nach diesem Kapitel weißt du, wie Runner arbeiten, wie Artefakte und Registries zusammenspielen und wie Geheimnisse sicher in eine Pipeline gelangen. Du kannst Pipelines wiederverwendbar aufbauen und kennst die Angriffsfläche, die eine Pipeline selbst darstellt.

## 8.1 Wer führt eigentlich aus?

Eine Pipeline ist eine Beschreibung. Ausgeführt wird sie von einem **Runner** (GitLab), **Agent** (Jenkins) oder **Runner** (GitHub Actions) — einem Prozess, der Aufträge abholt und abarbeitet.

| Betriebsart | Vorteil | Nachteil |
|---|---|---|
| **Gehostet** beim Anbieter | kein Betriebsaufwand, immer aktuell | Code verlässt das Haus, Laufzeitkontingente |
| **Selbst betrieben** | volle Kontrolle, Zugriff auf interne Netze | Pflege, Härtung, Skalierung liegen bei dir |

In regulierten Umgebungen ist die Frage schnell beantwortet: Selbst betriebene Runner im eigenen Netz, weil sie sonst weder an interne Registries noch an Testdatenbanken herankommen — und weil Quellcode das Haus nicht verlässt.

**Ephemere Runner sind der Standard.** Für jeden Auftrag entsteht ein frischer Container, der danach verworfen wird.

**Wichtig**

Ein dauerhaft laufender Runner, der Aufträge nacheinander im selben Arbeitsverzeichnis abarbeitet, ist eine Schneeflocke mit Rechten. Bleibt aus einem Auftrag etwas zurück — eine Datei, eine Umgebungsvariable, ein installiertes Paket — beeinflusst das den nächsten. Das Ergebnis sind Pipelines, die „meistens"durchlaufen, und Fehlersuchen, die niemand mag.

## 8.2 Artefakte und Registries

Ein **Artefakt** ist das Ergebnis des Bauvorgangs: ein Container-Image, ein Python-Rad, ein JAR, ein RPM. Es wird **einmal** erzeugt und in einer Registry abgelegt.

| Art | Beispiele |
|---|---|
| Container-Registry | Quay, Harbor, GitLab Container Registry, ghcr.io |
| Paket-Registry | Nexus, Artifactory, GitLab Package Registry |
| Zwischenartefakte der Pipeline | Testberichte, Abdeckungsberichte, SBOM |

**Warum eine interne Registry auch ohne eigene Images sinnvoll ist:** Sie spiegelt externe Images und macht dich unabhängig von deren Verfügbarkeit, Rate Limits und Löschungen. Wer schon einmal erlebt hat, dass ein Basis-Image aus dem Netz verschwand und damit sämtliche Builds standen, richtet die Spiegelung beim nächsten Mal vorher ein.

### 8.2.1 Beschriftung von Artefakten

```
# Nachvollziehbar: Commit, Version, Zeitpunkt
podman build \
  --label "org.opencontainers.image.revision=$CI_COMMIT_SHA" \
  --label "org.opencontainers.image.version=$CI_COMMIT_TAG" \
  --label "org.opencontainers.image.created=$(date -u +%Y-%m-%dT%H:%M:%SZ)" \
  --label "org.opencontainers.image.source=$CI_PROJECT_URL" \
  -t "$CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA" .
```

Diese Beschriftungen sind kein Selbstzweck. Sie beantworten im Störungsfall in einem einzigen Befehl die Frage, welcher Commit gerade produktiv läuft:

```
podman inspect kfz-rechner --format '{{index .Labels "org.opencontainers.image.revision"}}'
```

**Achtung**

**Der Tag `latest` hat in einer Pipeline nichts zu suchen.** Er ist ein beweglicher Zeiger — was gestern darunter lag, kann heute etwas anderes sein. Damit ist weder reproduzierbar, was getestet wurde, noch nachweisbar, was läuft.
In der Pipeline wird mit dem Commit-Kürzel getaggt, in der Produktion mit dem Digest referenziert.

## 8.3 Geheimnisse in der Pipeline

Die Pipeline braucht Zugangsdaten: für die Registry, für Zielsysteme, für Signaturschlüssel. Sie dürfen niemals im Repository stehen.

| Mechanismus | Einsatz |
|---|---|
| CI-Variablen (maskiert, geschützt) | einfache Fälle, kleine Teams |
| Externer Tresor (HashiCorp Vault, OpenBao) | Unternehmensumgebungen, rotierende Geheimnisse |
| OIDC-Verbund ohne dauerhafte Zugangsdaten | Cloud-Anbieter, heutiger Stand der Technik |
| Signaturschlüssel in einem HSM | Codesignatur, hohe Schutzanforderung |

```
# GitLab: Variable als "masked" und "protected" anlegen
# masked    -> wird in Logs durch [MASKED] ersetzt
# protected -> nur auf geschützten Zweigen verfügbar

deploy:
  script:
    - echo "$DEPLOY_KEY" > /tmp/key && chmod 600 /tmp/key
    - ssh -i /tmp/key conops@server './deploy.sh'
  after_script:
    - shred -u /tmp/key
```

**Achtung**

**Maskierung ist keine Verschlüsselung.** Sie ersetzt bekannte Zeichenketten in der Ausgabe — und versagt zuverlässig, sobald der Wert umkodiert wird. Ein `echo "$GEHEIM base64` erscheint unmaskiert im Protokoll, ebenso ein Wert, der in einer Fehlermeldung als JSON auftaucht.
Deshalb gilt: **Niemals `set -x` in Skripten mit Geheimnissen**, und niemals Umgebungsvariablen zur Fehlersuche vollständig ausgeben. Wer im Zweifel ist, ob ein Geheimnis im Protokoll gelandet ist, wechselt es. Das ist billiger als die Gewissheit.

## 8.4 Pipelines wiederverwenden

Bei zwanzig Projekten will niemand zwanzigmal dieselben achtzig Zeilen pflegen.

ci-vorlagen/python.yml

```
# Zentrale Vorlage, eigenes Repository
.python-tests:
  image: python:3.12-slim
  before_script:
    - pip install --quiet -r requirements.txt
  script:
    - ruff check src/ tests/
    - pytest --cov=src
  cache:
    key:
      files: [requirements.txt]
    paths: [.cache/pip]
```

.gitlab-ci.yml

```
include:
  - project: 'betrieb/ci-vorlagen'
    ref: v2.1.0                  # feste Version, nicht "main"
    file: '/python.yml'
```

```
unit-tests:
  extends: .python-tests
  stage: testen
```

Bei GitHub Actions leisten *Reusable Workflows* und *Composite Actions* dasselbe.

**Wichtig ist die feste Versionsangabe.** Eine eingebundene Vorlage ist fremder Code, der in deiner Pipeline mit deinen Zugangsdaten läuft. Wer `ref: main` schreibt, übernimmt jede Änderung ungeprüft — auch die, die jemand anders versehentlich hineingeschrieben hat.

## 8.5 Die Pipeline als Angriffsfläche

Ein Aspekt, der in Schulungen fast immer fehlt: Die Pipeline hat Zugriff auf Quellcode, Registry und Produktivsysteme. Wer sie kontrolliert, kontrolliert die Auslieferung.

| Risiko | Gegenmaßnahme |
|---|---|
| Fremde Actions und Vorlagen | auf Commit-Hash festnageln, nicht auf bewegliche Tags |
| Pipeline-Definition änderbar durch jeden | Änderungen an CI-Dateien gesondert prüfen lassen |
| Geheimnisse auf Feature-Branches | „protectedVariablen nur auf geschützten Zweigen |
| Fremde Pull Requests mit Zugriff | für externe Beiträge keine Geheimnisse bereitstellen |
| Runner mit Zugriff auf alles | Netzsegmentierung, minimale Rechte je Auftrag |
| Manipuliertes Basis-Image | eigene Spiegel-Registry, Signaturprüfung |

```
# Schlecht: beweglicher Tag
- uses: actions/checkout@v4

# Besser: auf den Commit festgenagelt
- uses: actions/checkout@b4ffde65f46336ab88eb53be808477a3936bae11 # v4.1.1
```

Der Angriff auf die Lieferkette ist kein theoretisches Szenario mehr — mehrere große Vorfälle der letzten Jahre liefen genau über kompromittierte Build-Prozesse. Mehr dazu in Kapitel 14.

## 8.6 Laufzeit im Griff behalten

```
# Parallelisieren, was unabhängig ist
stages: [pruefen, testen]

lint:
  stage: pruefen
unit-tests:
  stage: testen
integration-tests:
  stage: testen        # läuft parallel zu unit-tests
```

```
# Matrix: mehrere Varianten gleichzeitig
test:
  parallel:
    matrix:
      - PYTHON: ["3.11", "3.12", "3.13"]
  image: python:$PYTHON

# Nur bauen, was sich geändert hat
container-bauen:
  rules:
    - changes:
        - src/**/*
        - Containerfile
```

**Was am meisten bringt, in dieser Reihenfolge:** Abhängigkeiten zwischenspeichern, unabhängige Schritte parallelisieren, Container-Schichten so anordnen, dass der Cache greift (Kapitel 10), langsame E2E-Tests nachgelagert ausführen.

## 8.7 Lab: Die vollständige Pipeline

.gitlab-ci.yml

```
stages:
  - pruefen
  - testen
  - bauen
  - scannen
  - ausrollen

variables:
  IMAGE: "$CI_REGISTRY_IMAGE"
  TAG: "$CI_COMMIT_SHORT_SHA"

default:
  interruptible: true          # alter Lauf wird bei neuem Push abgebrochen

# ---------------------------------------------------------------
lint:
  stage: pruefen
  image: python:3.12-slim
  script:
    - pip install --quiet ruff
    - ruff check src/ tests/
    - ruff format --check src/ tests/

geheimnisse-suchen:
  stage: pruefen
  image: zricethezav/gitleaks:latest
  script:
    - gitleaks detect --source . --verbose --no-git

# ---------------------------------------------------------------
unit-tests:
  stage: testen
  image: python:3.12-slim
  script:
    - pip install --quiet -r requirements.txt
    - pytest --junitxml=report.xml --cov=src --cov-report=xml
  artifacts:
    when: always
```

```
    reports:
      junit: report.xml

integration-tests:
  stage: testen
  image: python:3.12-slim
  services:
    - name: postgres:16
      alias: db
  variables:
    POSTGRES_PASSWORD: test
    DB_HOST: db
  script:
    - pip install --quiet -r requirements.txt
    - pytest tests/integration/ -v

# ---------------------------------------------------------------
container-bauen:
  stage: bauen
  image: quay.io/podman/stable
  script:
    - podman login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
    - |
      podman build \
        --label "org.opencontainers.image.revision=$CI_COMMIT_SHA" \
        --label "org.opencontainers.image.source=$CI_PROJECT_URL" \
        -t "$IMAGE:$TAG" .
    - podman push "$IMAGE:$TAG"
    - podman inspect "$IMAGE:$TAG" --format '{{.Digest}}' > digest.txt
  artifacts:
    paths: [digest.txt]
  rules:
    - if: $CI_COMMIT_BRANCH == "main"

# ---------------------------------------------------------------
schwachstellen:
  stage: scannen
  image: aquasec/trivy:latest
  script:
    - trivy image --exit-code 0 --severity LOW,MEDIUM "$IMAGE:$TAG"
    - trivy image --exit-code 1 --severity HIGH,CRITICAL "$IMAGE:$TAG"
  rules:
    - if: $CI_COMMIT_BRANCH == "main"

# ---------------------------------------------------------------
ausrollen-test:
  stage: ausrollen
  environment:
    name: test
  script:
    - ./scripts/deploy.sh test "$IMAGE@$(cat digest.txt)"
    - ./scripts/smoketest.sh https://kfz-test.intern
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

**Aufgaben:**

1. Bring die Pipeline vollständig zum Laufen und miss die Gesamtdauer.
2. Verschiebe `geheimnisse-suchen` testweise in die letzte Stufe. Was ändert sich an der Rückmeldezeit?
3. Lass `trivy` eine kritische Schwachstelle finden — etwa mit einem absichtlich veralteten Basis-Image.

4. Baue die Vorlage aus 7.4 und lagere die Python-Stufen dorthin aus.
5. Prüfe im Protokoll, ob irgendwo ein Geheimnis unmaskiert auftaucht.

## 8.8 Anti-Pattern

| **Anti-Pattern** | **Folge** | **Abhilfe** |
|---|---|---|
| Pipeline zusammengeklickt | nicht versioniert, nicht reproduzierbar | Pipeline as Code |
| `latest` als Tag | unklar, was läuft | Commit-Kürzel und Digest |
| Geheimnisse im Protokoll | Kompromittierung | Maskierung, kein `set -x` |
| Fremde Actions auf beweglichen Tags | Lieferkettenrisiko | auf Commit-Hash festnageln |
| Kein Zwischenspeicher | jede Pipeline lädt alles neu | Cache je Abhängigkeitsdatei |
| Dauerhafte Runner mit Rückständen | „läuft mal, läuft mal nicht" | ephemere Runner |
| Copy-Paste über zwanzig Projekte | Änderungen erreichen nie alle | zentrale Vorlagen mit Version |
| Scan ohne Konsequenz | Bericht wird erzeugt und ignoriert | Schwellwert setzt die Pipeline rot |

## 8.9 Reflexionsfrage

*Wer könnte in deiner Umgebung eine Änderung an der Pipeline-Definition vornehmen, die beim nächsten Lauf mit Produktionsrechten ausgeführt wird — und würde das jemandem auffallen?*

# Kapitel 9

# Deployment-Strategien

**Lernziel**

Nach diesem Kapitel kennst du die fünf gängigen Ausrollverfahren mit ihren Stärken und Kosten, kannst begründet eines auswählen und weißt, wie Feature Flags Deployment und Release trennen. Du hast ein Blue/Green-Deployment mit Podman selbst aufgebaut.

## 9.1 Warum die Strategie zählt

Die Pipeline aus den letzten Kapiteln kann jetzt zuverlässig ausliefern. Offen ist die Frage: **Wie kommt die neue Version in Betrieb, ohne dass jemand etwas davon merkt?**

Das ist keine akademische Frage. Sie entscheidet darüber, ob eine Auslieferung ein Wartungsfenster, eine Nachtschicht und eine Vorabinformation an den Fachbereich braucht — oder ob sie am Dienstagvormittag zwischen zwei Besprechungen stattfindet.

## 9.2 Recreate — die einfachste Variante

```
alt  ############........
neu  ............########
                ^
           Ausfallzeit
```

Alte Version stoppen, neue starten. Ehrlich, verständlich, und mit einer Lücke dazwischen.

**Passt zu:** Batch-Verarbeitung, internen Werkzeugen, allem mit vereinbartem Wartungsfenster. **Passt nicht zu:** allem, was ständig erreichbar sein muss.

```
podman stop kfz-rechner && podman rm kfz-rechner
podman run -d --name kfz-rechner registry.intern/kfz-rechner:2.5.0
```

## 9.3 Rolling Update

```
Instanz 1  ######..########
Instanz 2  ########..######
Instanz 3  ##########..####
           ----------------->
           immer mindestens zwei erreichbar
```

Die Instanzen werden nacheinander ausgetauscht. Der Standard in Kubernetes und OpenShift.

**Vorteil:** kein Ausfall, kein zusätzlicher Ressourcenbedarf. **Nachteil:** Für einen Zeitraum laufen **beide Versionen gleichzeitig**. Das muss die Anwendung aushalten — insbesondere die Datenbank, siehe Expand-Contract in Kapitel 6.

**Wichtig**

Das gleichzeitige Laufen zweier Versionen wird regelmäßig unterschätzt. Wenn Version 2.4 ein Feld erwartet, das Version 2.5 nicht mehr schreibt, entstehen für einige Minuten Fehler, die anschließend spurlos verschwinden — und sich nicht reproduzieren lassen. Solche Fehler kosten mehr Ermittlungszeit als das gesamte Ausrollverfahren einspart.

## 9.4 Blue/Green

```
            +-------------+
Anfragen ->| Lastverteiler|
            +------+------+
                   |  Umschalten in einer Sekunde
         +---------+----------+
         v                    v
 +-----------+          +-----------+
 |  BLAU     |          |  GRÜN     |
 |  v2.4     |          |  v2.5     |
 |  aktiv    |          | Bereitsch|
 +-----------+          +-----------+
```

Zwei vollständige Umgebungen. Die neue Version wird in der inaktiven aufgebaut und in Ruhe geprüft. Dann schaltet der Lastverteiler um. Geht etwas schief, schaltet man zurück.

**Vorteil:** Rollback in Sekunden, Prüfung unter realen Bedingungen vor der Umschaltung. **Nachteil:** doppelte Ressourcen; die gemeinsame Datenbank bleibt der Knackpunkt.

## 9.5 Canary

```
100 % -+
       | v2.4  #############.............
       | v2.5  .............====#########
       +-----------------------------------
                 5 %   25 %   50 %  100 %
                   ^
         Fehlerrate beobachten,
```

```
            bei Auffälligkeit abbrechen
```

Die neue Version bekommt zunächst einen kleinen Anteil des Verkehrs. Bleiben Fehlerrate und Antwortzeiten unauffällig, wird der Anteil schrittweise erhöht.

**Vorteil:** Ein Fehler trifft fünf Prozent der Anwender statt aller. Man findet Probleme, die nur unter echter Last auftreten. **Nachteil:** braucht belastbare Messwerte — sonst schaltet man blind weiter. Ohne Observability (Kapitel 12) ist Canary lediglich ein langsameres Rolling Update.

Der Name stammt vom Kanarienvogel im Bergbau. Das Bild ist zutreffend, aber unhöflich gegenüber den fünf Prozent.

## 9.6 Shadow Traffic

Echte Anfragen werden zusätzlich an die neue Version geschickt, deren Antworten aber verworfen. Man vergleicht Laufzeiten und Ergebnisse, ohne dass ein Anwender betroffen ist.

**Passt zu:** Umbauten mit unverändertem fachlichem Verhalten — etwa dem Austausch einer Berechnungsbibliothek. **Achtung:** Schreibende Vorgänge müssen zuverlässig unterdrückt werden. Sonst verschickt der Schattenbetrieb Policen zweimal, und die Erklärung gegenüber dem Fachbereich wird unerfreulich.

## 9.7 Gegenüberstellung

| | **Recreate** | **Rolling** | **Blue/Green** | **Canary** | **Shadow** |
|---|---|---|---|---|---|
| Ausfallzeit | ja | nein | nein | nein | nein |
| Zusätzliche Ressourcen | keine | gering | **doppelt** | gering | doppelt |
| Rollback-Dauer | Minuten | Minuten | **Sekunden** | Sekunden | entfällt |
| Zwei Versionen gleichzeitig | nein | **ja** | nein | **ja** | ja |
| Braucht Messwerte | nein | nein | wenig | **ja** | ja |
| Aufwand der Einrichtung | sehr gering | gering | mittel | hoch | hoch |
| Risiko bei Fehlern | alle Anwender | alle Anwender | alle Anwender | **wenige** | keine |

**Empfehlung für den Einstieg:** Blue/Green. Es lässt sich mit einem Reverse Proxy und zwei Containern auf einem einzelnen Server umsetzen, erfordert keine Messinfrastruktur und liefert den größten Einzelgewinn — ein Rollback in Sekunden.

## 9.8 Feature Flags

Die Technik, die Deployment und Release endgültig entkoppelt.

src/features.py

```python
"""Sehr einfache Feature-Flag-Verwaltung über Umgebungsvariablen."""
import os


def feature_aktiv(name: str, standard: bool = False) -> bool:
    wert = os.environ.get(f"FEATURE_{name.upper()}")
    if wert is None:
        return standard
    return wert.lower() in ("1", "true", "ja", "an")
```

```python
from src.features import feature_aktiv

def berechne_praemie(sf_klasse, fahrzeugalter, jahreskilometer=None):
    beitrag = grundberechnung(sf_klasse, fahrzeugalter)

    if feature_aktiv("wenigfahrer_rabatt") and jahreskilometer:
        if jahreskilometer < 6000:
            beitrag *= Decimal("0.92")

    return beitrag.quantize(Decimal("0.01"))
```

### 9.8.1 Vier Arten von Flags

| Art | Zweck | Lebensdauer |
|---|---|---|
| Release-Flag | unfertige Funktion verbergen | Tage bis Wochen — **danach entfernen** |
| Betriebs-Flag | Funktion bei Überlast abschalten | dauerhaft |
| Berechtigungs-Flag | Funktion nur für bestimmte Gruppen | dauerhaft |
| Experiment-Flag | A/B-Test | Dauer des Versuchs |

**Feature Flags sind technische Schulden mit Verfallsdatum.**

Jedes Flag verdoppelt rechnerisch die Zahl der möglichen Codepfade. Zehn Flags ergeben theoretisch 1.024 Kombinationen, von denen sicher niemand mehr als drei testet. Nach zwei Jahren weiß niemand, ob `FEATURE_ALTE_TARIFLOGIK` noch irgendwo aktiv ist — und ob man den Zweig löschen darf.

**Regel:** Jedes Release-Flag bekommt beim Anlegen ein Ablaufdatum und ein Ticket zum Entfernen. Wer das nicht diszipliniert durchhält, tauscht das Problem langlebiger Branches gegen ein schlechter sichtbares ein.

## 9.9 Rollback oder Roll-forward

| | Rollback | Roll-forward |
|---|---|---|
| Vorgehen | zurück zur alten Version | Fehler beheben, neu ausliefern |
| Dauer | Sekunden bis Minuten | abhängig von Analyse und Pipeline |
| Voraussetzung | altes Artefakt verfügbar, DB verträglich | schnelle, verlässliche Pipeline |
| Passt zu | akuter Störung | kleinen, klaren Fehlern |

**Die pragmatische Regel:** Bei einer laufenden Störung wird zurückgerollt, nicht analysiert. Die Ursachensuche findet statt, wenn das System wieder läuft — nicht, während Anwender warten. Wer bei einem Ausfall zuerst die Ursache verstehen will, verlängert die Ausfallzeit um die Dauer seiner Neugier.

Roll-forward ist die richtige Wahl, wenn der Fehler offensichtlich und die Korrektur einzeilig ist — oder wenn eine Datenbankmigration das Zurückrollen ohnehin ausschließt.

## 9.10 Lab: Blue/Green mit Podman

Umsetzbar auf einem einzelnen Server, auch auf einem Raspberry Pi.

scripts/bluegreen.sh

```
#!/usr/bin/env bash
#
# Blue/Green-Deployment mit Podman und nginx.
# Aufruf:  ./bluegreen.sh <image:tag>
#
set -euo pipefail

NEUES_IMAGE="$1"
PROXY_KONF="/etc/nginx/conf.d/kfz-upstream.conf"

# --- Welche Farbe ist gerade aktiv? ---
if grep -q "8081" "$PROXY_KONF"; then
    AKTIV="blau";  AKTIV_PORT=8081
    NEU="gruen";   NEU_PORT=8082
else
    AKTIV="gruen"; AKTIV_PORT=8082
    NEU="blau";    NEU_PORT=8081
fi

echo "Aktiv: ${AKTIV} (${AKTIV_PORT}). Neue Version geht nach ${NEU} (${NEU_PORT})."

# --- Neue Version in der inaktiven Farbe starten ---
podman rm -f "kfz-${NEU}" 2>/dev/null || true
podman run -d --name "kfz-${NEU}" \
    -p "127.0.0.1:${NEU_PORT}:8080" \
    -e "UMGEBUNG=produktion" \
    --health-cmd 'curl -f http://localhost:8080/health || exit 1' \
    --health-interval 5s \
    "$NEUES_IMAGE"

# --- Warten, bis sie gesund ist ---
echo "Warte auf Gesundheitsstatus ..."
```

```
for i in {1..30}; do
    status=$(podman inspect "kfz-${NEU}" --format '{{.State.Health.Status}}')
    [[ "$status" == "healthy" ]] && break
    sleep 2
done

if [[ "$status" != "healthy" ]]; then
    echo "FEHLER: Neue Version wird nicht gesund. Umschaltung unterbleibt."
    podman logs --tail 50 "kfz-${NEU}"
    podman rm -f "kfz-${NEU}"
    exit 1
fi

# --- Fachliche Stichprobe vor der Umschaltung ---
ergebnis=$(curl -fsS "http://127.0.0.1:${NEU_PORT}/api/praemie?sf=5&alter=3" | jq -r '.betrag')
if [[ "$ergebnis" != "288.00" ]]; then
    echo "FEHLER: Fachliche Prüfung fehlgeschlagen (${ergebnis}). Umschaltung unterbleibt."
    podman rm -f "kfz-${NEU}"
    exit 1
fi

# --- Umschalten ---
echo "Schalte um auf ${NEU} ..."
sed -i "s/127.0.0.1:${AKTIV_PORT}/127.0.0.1:${NEU_PORT}/" "$PROXY_KONF"
nginx -t && systemctl reload nginx

echo "Umschaltung erfolgt. ${AKTIV} bleibt als Rückfallebene stehen."
echo "Zurückschalten:  sed -i 's/${NEU_PORT}/${AKTIV_PORT}/' ${PROXY_KONF} && systemctl reload
    ↪ nginx"
```

/etc/nginx/conf.d/kfz-upstream.conf

```
upstream kfz_backend {
    server 127.0.0.1:8081;
}

server {
    listen 443 ssl;
    server_name kfz.intern;

    ssl_certificate     /etc/pki/tls/certs/kfz.intern.crt;
    ssl_certificate_key /etc/pki/tls/private/kfz.intern.key;

    location / {
        proxy_pass http://kfz_backend;
        proxy_set_header Host            $host;
        proxy_set_header X-Real-IP       $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

**Aufgaben:**

1. Führe das Skript zweimal aus und beobachte, wie die Farben wechseln.
2. Lass die fachliche Stichprobe absichtlich fehlschlagen. Prüfe, dass die alte Version weiterläuft.
3. Miss, wie lange die Umschaltung tatsächlich dauert.
4. Schalte von Hand zurück und stoppe die Zeit. Das ist eure reale Rollback-Dauer — eine Zahl, die man kennen sollte, bevor man sie braucht.

5. Überlege: Was müsste am Skript geändert werden, damit es Canary statt Blue/Green umsetzt?

**Tipp**

Die vorletzte Aufgabe ist die wichtigste. In Kapitel 15 taucht diese Zahl als **Time to Restore Service** wieder auf — eine der vier zentralen DevOps-Kennzahlen. Wer sie gemessen hat, kann sie verbessern. Wer sie schätzt, schätzt meist zu optimistisch.

## 9.11 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Ausrollen ohne Rückweg | jede Auslieferung ist ein Wagnis | Rollback vorher üben, nicht im Ernstfall |
| Canary ohne Messung | man merkt nichts und schaltet trotzdem weiter | erst Observability, dann Canary |
| Flags ohne Ablaufdatum | Codepfad-Wildwuchs | Ablaufdatum und Ticket beim Anlegen |
| Blue/Green mit gemeinsamer Datenbank ohne Expand-Contract | Rollback bricht die Daten | rückwärtskompatible Migrationen |
| Rollback erst nach der Ursachenanalyse | Ausfallzeit verlängert sich | erst wiederherstellen, dann verstehen |
| Ausrollen freitags um 16 Uhr | niemand da, wenn es klemmt | entweder früher — oder gute Automatisierung |

Der letzte Punkt verdient eine Anmerkung: Das Freitagsverbot ist ein **Symptom**, keine Lösung. Es bedeutet, dass dem eigenen Ausrollverfahren nicht getraut wird. Reife Organisationen liefern freitags aus, weil sie es können — nicht weil sie mutig sind.

## 9.12 Reflexionsfrage

*Wann habt ihr zuletzt ein Rollback durchgeführt — und war das eine geübte Routine oder eine Improvisation?*

# Teil IV

# Teil D — Wie man betreibt

# Kapitel 10

# Infrastructure as Code

**Lernziel**

Nach diesem Kapitel kannst du deklarative von imperativer Automatisierung unterscheiden, Provisionierung von Konfigurationsverwaltung abgrenzen und erklären, was Idempotenz und Konfigurationsdrift bedeuten. Du hast den Server für den KFZ-Rechner vollständig aus Code erzeugt.

## 10.1 Das Problem: Schneeflocken und Drift

In Kapitel 6 kam der Begriff schon vor: Eine **Schneeflocke** ist ein über Jahre von Hand gepflegtes System, das niemand mehr identisch nachbauen kann. Jede ist einzigartig, keine ist reproduzierbar.

Wie sie entsteht, kennt jeder:

```
2021-03  Server aufgesetzt, nach Anleitung im Wiki
2021-07  Paket nachinstalliert, weil etwas fehlte   (nicht dokumentiert)
2022-01  Kernel-Parameter angepasst, wegen Performance
2022-09  Zertifikat von Hand ersetzt                (Kollege war im Urlaub)
2023-04  Firewall-Regel ergänzt, "nur mal kurz zum Testen"
2024-11  Niemand traut sich mehr, das System neu zu starten
```

Die schleichende Abweichung zwischen dokumentiertem Sollzustand und tatsächlichem Ist-Zustand heißt **Konfigurationsdrift**. Sie ist die Ursache hinter dem Satz „auf der Abnahme läuft es aber"— und sie wächst monoton, solange niemand eingreift.

**Infrastructure as Code** dreht das Verhältnis um: Nicht das System ist die Wahrheit, sondern das Repository. Weicht das System ab, wird es korrigiert — nicht die Dokumentation.

## 10.2 Deklarativ statt imperativ

| | Imperativ | Deklarativ |
|---|---|---|
| Beschreibt | den Weg | das Ziel |
| Beispiel | „installiere Paket X" | „Paket X soll vorhanden sein" |
| Zweiter Durchlauf | kann Schaden anrichten | ändert nichts |
| Typisch | Shell-Skript | Ansible, Terraform, Kubernetes |

```
# Imperativ – der zweite Durchlauf hängt die Zeile erneut an
useradd conops
echo "conops ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers.d/conops
```

```
# Deklarativ – beliebig oft ausführbar
- name: Dienstbenutzer sicherstellen
  ansible.builtin.user:
    name: conops
    shell: /bin/bash
    state: present

- name: sudo-Regel sicherstellen
  ansible.builtin.copy:
    dest: /etc/sudoers.d/conops
    content: "conops ALL=(ALL) NOPASSWD:ALL\n"
    mode: "0440"
    validate: /usr/sbin/visudo -cf %s
```

Die Eigenschaft, dass ein wiederholter Durchlauf nichts verändert, heißt **Idempotenz**. Sie ist der Kern der Sache: Nur so kann man den Automatismus regelmäßig laufen lassen und Drift damit aktiv beseitigen, statt ihn nur zu dokumentieren.

**Wichtig**

**Der Prüfstein für jede IaC-Umsetzung:** Lasse den Automatismus zweimal hintereinander laufen. Meldet der zweite Lauf Änderungen, ist die Beschreibung fehlerhaft — meist wegen eines Shell-Aufrufs ohne Zustandsprüfung.
Ein Automatismus, der bei jedem Lauf „geändert"meldet, ist wertlos als Kontrollinstrument: Man kann dann nicht mehr erkennen, ob eine echte Abweichung vorlag oder nur das übliche Rauschen.

## 10.3 Zwei Ebenen: Provisionierung und Konfiguration

| Ebene | Frage | Werkzeuge |
|---|---|---|
| **Provisionierung** | Welche Systeme gibt es überhaupt? | Terraform, OpenTofu, Pulumi, Cloud-Vorlagen |
| **Konfiguration** | Wie sind sie eingerichtet? | Ansible, Puppet, Chef, Salt |
| **Anwendungsebene** | Was läuft darauf? | Container, Kubernetes-Objekte |

In der Praxis kombiniert man: Terraform erzeugt die virtuellen Maschinen, Netze und Speicher; Ansible richtet die Betriebssysteme ein; Container bringen die Anwendungen. In einer reinen Rechenzentrumsumgebung ohne Cloud-Anteil fällt die erste Ebene oft weg oder wird von der Virtualisierungsplattform übernommen.

Für den Rest dieses Kapitels bleiben wir bei **Ansible**, weil es ohne Agenten, ohne Server und ohne Zustandsdatei auskommt — und weil es in RHEL-Umgebungen ohnehin der Standard ist. Die ausführliche Behandlung findest du in der Ansible-Unterlage dieses Wikis; hier geht es nur um die Einordnung in den DevOps-Zusammenhang.

## 10.4 Zustand und Drift-Erkennung

Werkzeuge wie Terraform führen eine **Zustandsdatei**, die den zuletzt hergestellten Stand festhält. Daraus ergibt sich ein mächtiges Werkzeug: der Abgleich von Soll und Ist.

```
terraform plan            # Was würde sich ändern? Ändert nichts.
ansible-playbook site.yml --check --diff
```

**Tipp**

**Ein sehr wirkungsvoller und selten genutzter Kniff:** Lass die Beschreibung nächtlich im Prüfmodus über alle Systeme laufen und melde jede gefundene Abweichung. Damit weißt du am nächsten Morgen, wo jemand von Hand eingegriffen hat — und zwar bevor es zum Problem wird. In regulierten Umgebungen ist das zugleich ein belastbarer Nachweis, dass die Konfiguration der Vorgabe entspricht. Der Aufwand liegt bei einem Nachmittag; der Nutzen bei jeder Prüfung.

Die Zustandsdatei bei Terraform ist gleichzeitig ein empfindlicher Punkt: Sie enthält oft Geheimnisse im Klartext und darf niemals ins Git-Repository. Sie gehört in einen Objektspeicher mit Verschlüsselung und Sperrmechanismus.

## 10.5 Veränderlich oder unveränderlich

| | **Mutable** | **Immutable** |
|---|---|---|
| Vorgehen bei Änderungen | System wird angepasst | System wird ersetzt |
| Drift | möglich | ausgeschlossen |
| Rollback | Rückkonfiguration | altes Abbild starten |
| Passt zu | klassischen Servern | Containern, Cloud-Instanzen |

Der unveränderliche Ansatz ist der konsequentere: Statt einen Server zu patchen, baut man ein neues Abbild und tauscht ihn aus. Genau das machen Container ohnehin, weshalb sie und IaC so gut zusammenpassen.

In gewachsenen Umgebungen mit Datenbankservern und Altanwendungen bleibt der veränderliche Ansatz aber die Realität — und dort ist regelmäßige, idempotente Konfigurationsverwaltung das nächstbeste Mittel.

## 10.6 Geheimnisse in IaC

Zugangsdaten gehören nicht ins Repository — auch nicht in eine Ansible-Variablendatei.

| Verfahren | Eignung |
|---|---|
| `ansible-vault` | einfache Fälle, alles in einem Repository |
| **SOPS** mit age oder GPG | verschlüsselte Werte, Git-freundliche Diffs |
| **HashiCorp Vault / OpenBao** | Unternehmen, rotierende und kurzlebige Geheimnisse |
| Geheimnisverwaltung des Cloud-Anbieters | wenn ohnehin dort betrieben |

```
ansible-vault encrypt_string 'GeheimesPasswort' --name 'db_passwort'
```

Bewährt hat sich die Trennung in eine unverschlüsselte Datei mit sprechenden Variablennamen und eine verschlüsselte mit den Werten. So bleibt im Klartext sichtbar, *welche* Geheimnisse existieren, ohne ihren Inhalt preiszugeben.

## 10.7 IaC testen

Beschreibungen von Infrastruktur sind Code — also gelten die Regeln aus Kapitel 4.

| Stufe | Werkzeug | Prüft |
|---|---|---|
| Syntax | `ansible-playbook -syntax-check`, `terraform validate` | formale Korrektheit |
| Stil | `ansible-lint`, `tflint` | gute Praxis, typische Fehler |
| Sicherheit | `checkov`, `tfsec`, `kics` | offene Ports, fehlende Verschlüsselung |
| Verhalten | `molecule` | Rolle läuft in einem Container und ist idempotent |
| Probelauf | `-check`, `terraform plan` | was würde sich ändern |

```
# Molecule: Rolle in einem Container prüfen, inklusive Idempotenz
molecule test
```

Der Idempotenztest von Molecule ist dabei der wertvollste Teil: Er führt die Rolle zweimal aus und schlägt fehl, wenn der zweite Lauf Änderungen meldet.

## 10.8 Lab: Der Server für den KFZ-Rechner aus Code

roles/kfz_host/tasks/main.yml

```
---
- name: Distributionsspezifische Variablen laden
  ansible.builtin.include_vars: "{{ ansible_facts['os_family'] }}.yml"

- name: Erforderliche Pakete installieren
  ansible.builtin.package:
    name: "{{ kfz_pakete }}"
    state: present

- name: Dienstbenutzer anlegen
  ansible.builtin.user:
    name: "{{ kfz_benutzer }}"
    shell: /bin/bash
    create_home: true
    state: present

- name: Lingering aktivieren, damit Container ohne Anmeldung laufen
  ansible.builtin.command: "loginctl enable-linger {{ kfz_benutzer }}"
  args:
    creates: "/var/lib/systemd/linger/{{ kfz_benutzer }}"

- name: Verzeichnis für Quadlet-Units anlegen
  ansible.builtin.file:
    path: "/home/{{ kfz_benutzer }}/.config/containers/systemd"
    state: directory
    owner: "{{ kfz_benutzer }}"
    group: "{{ kfz_benutzer }}"
    mode: "0755"

- name: Unprivilegierte Ports ab 80 freigeben
  ansible.posix.sysctl:
    name: net.ipv4.ip_unprivileged_port_start
    value: "80"
    sysctl_file: /etc/sysctl.d/99-rootless.conf
    state: present
    reload: true

- name: Datenverzeichnis anlegen
  ansible.builtin.file:
    path: "{{ kfz_datenverzeichnis }}"
    state: directory
    owner: "{{ kfz_benutzer }}"
    group: "{{ kfz_benutzer }}"
    mode: "0750"

- name: SELinux-Kontext für Containerdaten setzen
  community.general.sefcontext:
    target: "{{ kfz_datenverzeichnis }}(/.*)?"
    setype: container_file_t
    state: present
  when: ansible_facts['os_family'] == 'RedHat'
  notify: SELinux-Kontext anwenden

- name: Firewall für HTTPS öffnen
  ansible.posix.firewalld:
    service: https
    permanent: true
    immediate: true
    state: enabled
  when: ansible_facts['os_family'] == 'RedHat'
```

```
- name: Quadlet-Unit ausbringen
  ansible.builtin.template:
    src: kfz-rechner.container.j2
    dest: "/home/{{ kfz_benutzer }}/.config/containers/systemd/kfz-rechner.container"
    owner: "{{ kfz_benutzer }}"
    mode: "0644"
  notify: Container neu starten
```

roles/kfz_host/handlers/main.yml

```
---
- name: SELinux-Kontext anwenden
  ansible.builtin.command: "restorecon -Rv {{ kfz_datenverzeichnis }}"

- name: Container neu starten
  ansible.builtin.systemd_service:
    name: kfz-rechner.service
    state: restarted
    daemon_reload: true
    scope: user
  become: true
  become_user: "{{ kfz_benutzer }}"
```

**Aufgaben:**

1. Führe das Playbook zweimal aus. Der zweite Lauf muss `changed=0` melden.
2. Verändere von Hand etwas auf dem Server — etwa die Firewall-Regel. Lass das Playbook mit `-check -diff` laufen. Es muss die Abweichung finden.
3. Lösche die virtuelle Maschine und baue sie aus dem Playbook neu auf. Stoppe die Zeit.
4. Ergänze eine Prüfung, die sicherstellt, dass der Dienst nach dem Durchlauf tatsächlich antwortet.
5. Baue das Playbook in die Pipeline aus Kapitel 7 ein — zunächst nur mit `-check`.

**Tipp**

Die dritte Aufgabe ist die eigentliche Prüfung. Solange niemand den Server einmal absichtlich gelöscht und neu erzeugt hat, ist unbewiesen, dass die Beschreibung vollständig ist. Fast immer fehlt beim ersten Versuch etwas — ein Zertifikat, eine Datei, eine Berechtigung, die vor Jahren jemand von Hand gesetzt hat.
Diese Übung einmal jährlich durchzuführen, ersetzt einen erheblichen Teil einer Notfallübung. Und sie beantwortet nebenbei die Reflexionsfrage aus Kapitel 6 mit einer Zahl statt mit einer Hoffnung.

## 10.9 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Shell-Skripte statt deklarativer Beschreibung | nicht wiederholbar | idempotente Module verwenden |
| Nach dem Automatismus von Hand nachbessern | Drift kehrt sofort zurück | Änderung in den Code, dann erneut ausführen |
| IaC nicht versioniert | keine Historie, kein Rollback | Repository wie für Anwendungscode |
| Geheimnisse im Klartext | Kompromittierung | Vault, SOPS, `ansible-vault` |
| Einmal ausgeführt und nie wieder | Drift wächst unbemerkt | regelmäßiger Prüflauf |
| Terraform-Zustandsdatei im Git | Geheimnisse öffentlich, Konflikte | Objektspeicher mit Sperre |
| Kein Idempotenztest | Beschreibung meldet ständig Änderungen | Molecule, zweiter Durchlauf |
| Produktion als Übungsfeld | Ausfall beim Lernen | erst Test, dann Abnahme, dann Produktion |

## 10.10 Reflexionsfrage

*Welcher Server in deiner Umgebung würde dir die meisten Sorgen bereiten, wenn er heute Nacht ausfiele — und warum ausgerechnet dieser?*

Die Antwort benennt fast immer eine Schneeflocke. Und damit auch das erste sinnvolle Ziel für Infrastructure as Code.

# Kapitel 11

# Container und Orchestrierung

**Lernziel**

Nach diesem Kapitel kannst du erklären, warum Container das natürliche Artefakt einer Auslieferungskette sind, kennst die für DevOps wesentlichen Zwölf-Faktoren-Regeln und kannst begründen, wann Orchestrierung nötig wird und wann nicht. Der KFZ-Rechner läuft als Container unter systemd.

## 11.1 Warum Container hier auftauchen

Die technischen Grundlagen — Namespaces, cgroups, Image-Schichten, Podman gegen Docker — behandelt die Container-Unterlage dieses Wikis ausführlich. Dieses Kapitel beschränkt sich auf die Frage: **Was leisten Container für die Auslieferungskette?**

Die Antwort besteht aus vier Punkten, die alle an früher Gesagtes anknüpfen:

| Anforderung | Kapitel | Was der Container beiträgt |
|---|---|---|
| Einmal bauen, überall ausrollen | 6 | Das Image *ist* das Artefakt — bitgleich in jeder Umgebung |
| Umgebungsparität | 6 | Laufzeitumgebung steckt im Image, nicht auf dem Server |
| Schnelles Rollback | 8 | Altes Image starten, Sekunden statt Minuten |
| Reproduzierbarkeit | 9 | Containerfile ist Code, das Image ist unveränderlich |

**Wichtig**

**Der Satz, auf den es ankommt:** Ohne Container ist das Artefakt einer Pipeline ein Archiv, das auf einem passend vorbereiteten Server ausgepackt werden muss — und die Frage, ob dieser Server passend vorbereitet ist, bleibt offen.
Mit Container ist das Artefakt selbst der vorbereitete Server. Das ist der eigentliche Grund, warum Container und Continuous Delivery gemeinsam populär wurden: Das eine löst ein Problem, das das andere aufwirft.

## 11.2 Die Zwölf Faktoren — die relevanten sechs

Die *Twelve-Factor App* ist eine Sammlung von Regeln für Anwendungen, die in solchen Umgebungen betrieben werden sollen. Sechs davon sind für diese Unterlage unmittelbar relevant.

| Faktor | Regel | Warum |
|---|---|---|
| **III Konfiguration** | Konfiguration in Umgebungsvariablen, nicht im Code | ein Image für alle Umgebungen |
| **VI Prozesse** | zustandslos, nichts im lokalen Dateisystem behalten | Container jederzeit ersetzbar |
| **IX Einweg** | schneller Start, sauberes Herunterfahren auf SIGTERM | Rolling Update ohne Fehler |
| **XI Logs** | nach `stdout` schreiben, nicht in Dateien | die Umgebung sammelt ein |
| **X Parität** | Entwicklung, Test und Produktion ähnlich halten | weniger Überraschungen |
| **XII Verwaltungsaufgaben** | Migrationen als einmalige Prozesse | reproduzierbar, nachvollziehbar |

```python
# Faktor III und XI in der Praxis
import logging
import os
import sys

logging.basicConfig(
    stream=sys.stdout,                      # XI: nach stdout, nicht in Dateien
    format='{"zeit":"%(asctime)s","stufe":"%(levelname)s","text":"%(message)s"}',
)

DB_HOST = os.environ["DB_HOST"]             # III: aus der Umgebung
DB_PASS = os.environ["DB_PASSWORD"]         # kein Standardwert bei Geheimnissen
LOG_LEVEL = os.environ.get("LOG_LEVEL", "INFO")
```

**Tipp**

**Faktor IX wird am häufigsten übersehen.** Beim Rolling Update sendet die Umgebung `SIGTERM` und wartet einige Sekunden. Eine Anwendung, die dieses Signal ignoriert, wird anschließend hart abgeschossen — mitten in laufenden Anfragen.
Praktisch heißt sauberes Herunterfahren: keine neuen Anfragen mehr annehmen, laufende zu Ende bearbeiten, Verbindungen schließen, beenden. Das sind zwanzig Zeilen Code und der Unterschied zwischen einem unauffälligen Update und sporadischen Fehlern, die niemand reproduzieren kann.

## 11.3 Wann Orchestrierung nötig wird

Die ehrliche Antwort lautet: seltener, als angenommen wird.

| Stufe | Werkzeug | Taugt für | Grenze |
|---|---|---|---|
| 1 | `podman run` | Ausprobieren | überlebt keinen Neustart |
| 2 | **Quadlet / systemd** | 1–30 Dienste auf einem Server | ein Host |
| 3 | Compose | Entwicklungsumgebungen | nicht für Produktion gedacht |
| 4 | k3s | kleiner Cluster, Ausfallsicherheit | Betriebsaufwand steigt |
| 5 | Kubernetes / OpenShift | Unternehmensumgebungen | halbe bis ganze Stelle für die Plattform |

Der Sprung von Stufe 2 auf Stufe 4 lohnt sich, wenn **einer** dieser Punkte zutrifft:

- Der Ausfall eines einzelnen Servers ist nicht hinnehmbar.
- Die Last schwankt so stark, dass automatisch skaliert werden muss.
- Mehrere Teams brauchen abgestufte Rechte auf derselben Plattform.
- Die Anzahl der Dienste übersteigt das, was eine Person überblicken kann.

Trifft nichts davon zu, ist Stufe 2 die wirtschaftlichere Wahl. Ein einzelner Server mit Quadlet-Units, Git-Anbindung und einem Deployment-Skript liefert Reproduzierbarkeit, Versionierung, Rollback und saubere Protokollierung — mit Werkzeugen, die jeder Linux-Administrator bereits beherrscht.

## 11.4 Kurzüberblick Kubernetes und OpenShift

Nur so viel, wie zur Einordnung nötig ist.

| Objekt | Aufgabe |
|---|---|
| **Pod** | kleinste Einheit: ein oder mehrere Container mit gemeinsamer IP |
| **Deployment** | sorgt für die gewünschte Anzahl Pods, ermöglicht Rolling Update |
| **Service** | stabile interne Adresse und Lastverteilung |
| **Ingress / Route** | Zugang von außen |
| **ConfigMap / Secret** | Konfiguration und Geheimnisse getrennt vom Image |

Das Grundprinzip ist eine **Regelschleife**: Man beschreibt den Sollzustand, ein Controller vergleicht laufend mit dem Ist-Zustand und korrigiert. Genau dasselbe Denkmuster wie bei Ansible aus Kapitel 9 — nur dauerhaft statt einmalig.

**OpenShift** ist Kubernetes mit integrierter Registry, Weboberfläche, Pipelines, Monitoring und deutlich strengeren Sicherheitsvorgaben. Der wichtigste praktische Unterschied: Container laufen dort mit einer zufällig zugewiesenen Benutzerkennung und niemals als Root. Ein Image, das unter OpenShift läuft, läuft überall — umgekehrt gilt das nicht.

## 11.5 Lab: Der KFZ-Rechner wird zum Dienst

Containerfile

```
# ---------- Stufe 1: Abhängigkeiten ----------
FROM registry.access.redhat.com/ubi9/python-312:latest AS builder

WORKDIR /build
COPY requirements.txt .
RUN pip install --no-cache-dir --target=/build/deps -r requirements.txt

# ---------- Stufe 2: Laufzeit ----------
FROM registry.access.redhat.com/ubi9/python-312-minimal:latest

LABEL org.opencontainers.image.title="KFZ-Rechner" \
      org.opencontainers.image.vendor="Beispiel AG"

WORKDIR /app
COPY --from=builder /build/deps /app/deps
COPY src/ ./src/

ENV PYTHONPATH=/app/deps \
    PYTHONUNBUFFERED=1 \
    LOG_LEVEL=INFO

# Gruppe 0 statt fester UID -> läuft auch unter OpenShift
RUN chgrp -R 0 /app && chmod -R g=u /app
USER 1001

EXPOSE 8080

HEALTHCHECK --interval=30s --timeout=3s --start-period=10s --retries=3 \
  CMD python -c "import urllib.request,sys; \
      sys.exit(0 if urllib.request.urlopen('http://localhost:8080/health').status==200 else 1)"

CMD ["python", "-m", "src.server"]
```

/.config/containers/systemd/kfz-rechner.container

```
[Unit]
Description=KFZ-Prämienrechner
After=network-online.target

[Container]
Image=registry.intern/kfz-rechner:2.5.0
ContainerName=kfz-rechner
PublishPort=127.0.0.1:8081:8080
Environment=UMGEBUNG=produktion
Environment=LOG_LEVEL=INFO
Secret=kfz_db_passwort,type=env,target=DB_PASSWORD
Volume=/srv/kfz/daten:/app/daten:Z
ReadOnly=true
Tmpfs=/tmp:rw,size=32m
NoNewPrivileges=true
DropCapability=ALL
Memory=512m
AutoUpdate=registry

[Service]
Restart=always
TimeoutStartSec=90

[Install]
WantedBy=default.target
```

```
systemctl --user daemon-reload
systemctl --user start kfz-rechner.service
systemctl --user status kfz-rechner.service
journalctl --user -u kfz-rechner.service -f
```

**Aufgaben:**

1. Bring den Dienst zum Laufen und prüfe nach einem Neustart des Servers, dass er von selbst wiederkommt.
2. Ergänze sauberes Herunterfahren auf `SIGTERM` im Python-Code und beobachte den Unterschied bei `systemctl restart`.
3. Vergleiche die Größe des Images mit und ohne Multi-Stage-Build.
4. Ergänze den Bau des Images in der Pipeline aus Kapitel 7 und rolle das Ergebnis über das Blue/Green-Skript aus Kapitel 8 aus.
5. Erzeuge mit `podman kube generate kfz-rechner` eine Kubernetes-Beschreibung und sieh sie dir an. Was fehlt darin für den Produktivbetrieb?

## 11.6 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Mehrere Dienste in einem Container | schlecht wartbare VM im Container-Gewand | ein Prozess je Container |
| SSH-Server im Container | Änderungen von Hand, die beim Neustart verschwinden | `podman exec` zur Diagnose, sonst Image ändern |
| Als Root laufen lassen | unnötige Angriffsfläche, läuft nicht unter OpenShift | `USER 1001`, Gruppe 0 |
| Konfiguration im Image | ein Image je Umgebung, Prinzip verletzt | Umgebungsvariablen |
| Logs in Dateien im Container | verschwinden mit dem Container | nach `stdout` schreiben |
| Daten im Container ablegen | Datenverlust beim Austausch | Volumes |
| `latest` in Produktion | unklar, was läuft | feste Version, besser Digest |
| `SIGTERM` ignoriert | Fehler bei jedem Update | sauberes Herunterfahren umsetzen |
| Kubernetes für drei Container | Aufwand ohne Gegenwert | Quadlet auf einem Server |

## 11.7 Reflexionsfrage

*Wie viele Dienste betreibt ihr, und wie oft ist im letzten Jahr ein Server so ausgefallen, dass ein Cluster geholfen hätte?*

Die zweite Zahl ist meist kleiner, als die Diskussion über Kubernetes vermuten lässt.

# Kapitel 12

# GitOps

**Lernziel**

Nach diesem Kapitel kennst du die vier GitOps-Prinzipien, kannst Push- von Pull-Verfahren unterscheiden und weißt, wie Geheimnisse in einem öffentlich lesbaren Repository sicher abgelegt werden. Du hast einen einfachen Pull-Abgleich für den KFZ-Rechner gebaut.

## 12.1 Die Idee

GitOps ist die konsequente Fortsetzung von Infrastructure as Code: **Git ist nicht die Dokumentation des Systems, Git ist das System.**

Vier Prinzipien:

1. **Deklarativ.** Der Sollzustand ist vollständig beschrieben, nicht als Abfolge von Schritten.
2. **Versioniert und unveränderlich.** Git ist die einzige Quelle der Wahrheit, jede Änderung hat Autor, Zeitpunkt und Begründung.
3. **Automatisch gezogen.** Ein Automatismus holt sich den Sollzustand — statt dass eine Pipeline ihn hineinschiebt.
4. **Fortlaufend abgeglichen.** Weicht der Ist-Zustand ab, wird korrigiert. Dauerhaft, nicht nur beim Ausrollen.

Der vierte Punkt ist der eigentliche Unterschied zu allem Vorherigen. Eine Pipeline stellt einen Zustand **einmal** her. Ein GitOps-Automatismus hält ihn **aufrecht**.

## 12.2 Push oder Pull

```
PUSH (klassische Pipeline)

Git --> Pipeline --> Zugangsdaten --> Zielsystem
                     ^
           Die Pipeline braucht Schreibrechte
           auf der Produktion.
```

```
PULL (GitOps)

Git <------ Agent im Zielsystem --> wendet an
            zieht regelmäßig
            ^
   Keine Zugangsdaten außerhalb des Zielsystems.
   Die Pipeline endet an der Registry.
```

| | **Push** | **Pull** |
|---|---|---|
| Wer verbindet sich zu wem | Pipeline zum Ziel | Ziel zum Repository |
| Zugangsdaten für Produktion | in der Pipeline | im Zielsystem |
| Firewall | eingehend zum Ziel | nur ausgehend |
| Drift-Korrektur | nein | ja, laufend |
| Einrichtung | einfach | Agent nötig |

Der sicherheitstechnische Vorteil des Pull-Verfahrens ist erheblich: Die Pipeline braucht keine Produktionszugangsdaten mehr. Sie baut, prüft und legt in der Registry ab — mehr nicht. Wer die Pipeline übernimmt, hat damit noch keinen Zugriff auf die Produktion. Angesichts der Angriffsfläche aus Kapitel 7 ist das ein starkes Argument.

## 12.3 Zwei Repositories

Bewährt hat sich die Trennung:

```
kfz-rechner/                 (Anwendungsrepository)
+-- src/
+-- tests/
+-- Containerfile
+-- .gitlab-ci.yml           baut das Image, legt es in der Registry ab

kfz-betrieb/                 (Konfigurationsrepository)
+-- test/
|   +-- kfz-rechner.yaml     Image-Tag: 2.5.1
+-- abnahme/
|   +-- kfz-rechner.yaml     Image-Tag: 2.5.0
+-- produktion/
    +-- kfz-rechner.yaml     Image-Tag: 2.4.8
```

**Warum getrennt?** Der Ist-Zustand jeder Umgebung ist damit eine Zeile in einer Datei, und ein Deployment ist ein Commit. Der Git-Verlauf des Konfigurationsrepositories beantwortet lückenlos, welche Version wann in welcher Umgebung lief und wer sie freigegeben hat. Ein Rollback ist ein `git revert`.

**Tipp**

**Für regulierte Umgebungen ist das der entscheidende Punkt.** Die Freigabe einer Produktivänderung wird zu einem Pull Request im Konfigurationsrepository — mit Prüfer, Zeitstempel und Begründung. Das Vier-Augen-Prinzip ist damit technisch erzwungen statt organisatorisch vereinbart, und der Nachweis fällt als Nebenprodukt

an. Mehr dazu in Kapitel 17.

## 12.4 Werkzeuge

| Werkzeug | Umfeld | Anmerkung |
|---|---|---|
| **Argo CD** | Kubernetes, OpenShift | Weboberfläche, sehr verbreitet |
| **Flux** | Kubernetes | schlanker, stärker CLI-orientiert |
| **OpenShift GitOps** | OpenShift | Argo CD als Operator, integriert |
| **systemd-Timer plus Skript** | einzelner Server | reicht überraschend weit |

Die letzte Zeile ist ernst gemeint. GitOps ist ein Prinzip, kein Produkt — und auf einem einzelnen Server lässt es sich mit einem Zeitgeber und dreißig Zeilen Shell umsetzen.

## 12.5 Geheimnisse in einem Git-Repository

Das offensichtliche Problem: Wenn alles in Git steht, stehen dort auch die Passwörter. Das darf nicht sein.

| Verfahren | Funktionsweise | Eignung |
|---|---|---|
| **SOPS** | Werte einzeln verschlüsselt, Struktur bleibt lesbar | gut, auch außerhalb von Kubernetes |
| **Sealed Secrets** | verschlüsselt für genau einen Cluster | Kubernetes |
| **External Secrets Operator** | verweist auf einen externen Tresor | Unternehmensumgebungen |
| **Podman Secrets** | Geheimnis lokal, Unit verweist darauf | einzelner Server |

```
# SOPS: Struktur lesbar, Werte verschlüsselt - gute Diffs im Git
db:
    host: db-prod.intern
    benutzer: kfz_app
    passwort: ENC[AES256_GCM,data:8Kj2mQ==,iv:...,tag:...,type:str]
```

**Achtung**

**Ein häufiger Denkfehler:** „Das Repository ist ja privat, da kann das Passwort ruhig im Klartext stehen."
Ein privates Repository wird geklont — auf Notebooks, in CI-Runner, in Sicherungen. Es hat Leseberechtigte, die man nicht mehr überblickt, und es überlebt Personalwechsel. Der Klartext bleibt zudem für immer in der Historie, auch nach dem Löschen. Verschlüsselung kostet einen Nachmittag Einrichtung; der Alternativpfad kostet im Ernstfall den Wechsel

sämtlicher Zugangsdaten.

## 12.6 Lab: GitOps für einen einzelnen Server

/usr/local/bin/gitops-abgleich.sh

```
#!/usr/bin/env bash
#
# Minimaler GitOps-Abgleich: holt den Sollzustand und stellt ihn her.
#
set -euo pipefail

REPO="${HOME}/kfz-betrieb"
UMGEBUNG="produktion"
LOG_TAG="gitops"

log() { logger -t "$LOG_TAG" "$*"; echo "$*"; }

cd "$REPO"

# --- 1. Sollzustand holen ---
ALT="$(git rev-parse HEAD)"
git fetch --quiet origin
git reset --quiet --hard origin/main
NEU="$(git rev-parse HEAD)"

# --- 2. Gewünschtes Image aus der Konfiguration lesen ---
SOLL_IMAGE="$(grep '^image:' "${UMGEBUNG}/kfz-rechner.yaml" | awk '{print $2}')"

# --- 3. Ist-Zustand ermitteln ---
IST_IMAGE="$(podman inspect kfz-rechner --format '{{.ImageName}}' 2>/dev/null || echo "keiner")"

if [[ "$SOLL_IMAGE" == "$IST_IMAGE" ]]; then
    exit 0                       # Soll = Ist, nichts zu tun
fi

log "Abweichung erkannt: ist=${IST_IMAGE} soll=${SOLL_IMAGE} (commit ${NEU:0:8})"

# --- 4. Sollzustand herstellen ---
podman pull "$SOLL_IMAGE"
sed -i "s|^Image=.*|Image=${SOLL_IMAGE}|" \
    "${HOME}/.config/containers/systemd/kfz-rechner.container"

systemctl --user daemon-reload
systemctl --user restart kfz-rechner.service

# --- 5. Ergebnis prüfen, sonst zurückrollen ---
sleep 5
if ! curl -fsS --max-time 5 http://127.0.0.1:8081/health > /dev/null; then
    log "FEHLER: Dienst antwortet nicht. Rolle zurück auf ${IST_IMAGE}."
    sed -i "s|^Image=.*|Image=${IST_IMAGE}|" \
        "${HOME}/.config/containers/systemd/kfz-rechner.container"
    systemctl --user daemon-reload
    systemctl --user restart kfz-rechner.service
    exit 1
fi

log "Abgleich erfolgreich: ${SOLL_IMAGE} (commit ${NEU:0:8})"
```

/.config/systemd/user/gitops-abgleich.timer

```
[Unit]
Description=GitOps-Abgleich alle zwei Minuten

[Timer]
OnBootSec=2min
OnUnitActiveSec=2min
AccuracySec=10s

[Install]
WantedBy=timers.target
```

/.config/systemd/user/gitops-abgleich.service

```
[Unit]
Description=GitOps-Abgleich

[Service]
Type=oneshot
ExecStart=/usr/local/bin/gitops-abgleich.sh
```

```
systemctl --user enable --now gitops-abgleich.timer
systemctl --user list-timers
journalctl --user -u gitops-abgleich.service -f
```

**Aufgaben:**

1. Ändere im Konfigurationsrepository den Image-Tag und beobachte, wie der Server binnen zwei Minuten nachzieht — ohne dass jemand etwas ausrollt.
2. Ändere den laufenden Container von Hand. Was passiert beim nächsten Abgleich?
3. Trage einen nicht existierenden Tag ein. Prüfe, dass der Dienst weiterläuft.
4. Führe das Rollback über `git revert` im Konfigurationsrepository durch und stoppe die Zeit.
5. Überlege: Welche Zugangsdaten braucht die Pipeline in diesem Aufbau noch für die Produktion?

**Tipp**

Die letzte Frage ist die Pointe des Kapitels. Die Antwort lautet: **keine.** Die Pipeline endet an der Registry. Wer sie kompromittiert, kann ein Image ablegen — aber nichts ausrollen, solange niemand den entsprechenden Commit im Konfigurationsrepository freigibt.

## 12.7 Anti-Pattern

| **Anti-Pattern** | **Folge** | **Abhilfe** |
|---|---|---|
| Änderungen von Hand am Zielsystem | werden beim nächsten Abgleich überschrieben — oder verhindern ihn | ausschließlich über Git |
| Geheimnisse im Klartext | Kompromittierung | SOPS, Sealed Secrets, externer Tresor |
| Anwendungs- und Konfigurationsrepository vermischt | Bauvorgang löst Deployment aus, keine saubere Freigabe | trennen |
| Abgleich ohne Prüfung | fehlerhafter Zustand wird stur hergestellt | Gesundheitsprüfung mit Rückfall |
| Kein Rückweg | Fehler wird endlos neu ausgerollt | Rollback im Skript vorsehen |
| `latest` im Konfigurationsrepository | Git-Verlauf sagt nichts mehr aus | feste Version oder Digest |

## 12.8 Reflexionsfrage

*Wenn jemand heute Nacht auf einem eurer Produktivsysteme von Hand eine Konfigurationsdatei ändert — wann und wie würdet ihr das erfahren?*

# Kapitel 13

# Observability

**Lernziel**

Nach diesem Kapitel kannst du Monitoring von Observability unterscheiden, kennst die drei Säulen und die vier goldenen Signale und kannst begründen, warum auf Symptome und nicht auf Ursachen alarmiert wird. Der KFZ-Rechner liefert Metriken, wird überwacht und meldet sich bei Problemen.

## 13.1 Monitoring oder Observability

Die Begriffe werden oft synonym verwendet, meinen aber Unterschiedliches.

| | **Monitoring** | **Observability** |
|---|---|---|
| Beantwortet | bekannte Fragen | unvorhergesehene Fragen |
| Beispiel | „Ist die CPU über 80 Prozent?" | „Warum sind ausgerechnet Anfragen aus Filiale 12 langsam?" |
| Vorgehen | vorher festgelegte Prüfungen | Daten reichhaltig genug, um nachträglich zu forschen |
| Findet | die Probleme, die man erwartet hat | auch die anderen |

Monitoring ist nicht überholt — Observability ist die Erweiterung. Man braucht beides: feste Prüfungen für Bekanntes und genug Kontext, um Unbekanntes untersuchen zu können.

Aus Kapitel 2 wieder aufgegriffen: Das ist der **Zweite Weg**, die Rückmeldung. Ohne Messung bleibt er eine Absichtserklärung.

## 13.2 Die drei Säulen

| Säule | Beantwortet | Beispiel |
|---|---|---|
| **Metriken** | Was passiert? Wie viel? Wie schnell? | 340 Anfragen/min, 1,2 % Fehler, p95 bei 180 ms |
| **Logs** | Was genau ist passiert? | „Berechnung für Vertrag 4711 abgebrochen: Division durch null" |
| **Traces** | Wo genau hat es gehakt? | Anfrage 78 ms in der Anwendung, 1.240 ms in der Datenbank |

Metriken sind billig, aggregiert und gut für Alarme. Logs sind teuer, detailliert und gut für die Ursachensuche. Traces zeigen den Weg einer einzelnen Anfrage durch mehrere Systeme — und sind bei verteilten Anwendungen unverzichtbar.

**Der übliche Ablauf einer Fehlersuche:** Ein Alarm auf einer Metrik meldet, *dass* etwas nicht stimmt. Ein Trace zeigt, *wo*. Die Logs sagen, *warum*.

## 13.3 Die vier goldenen Signale

Aus dem Betriebshandbuch von Google, und als Einstieg schwer zu schlagen. Wer nur vier Dinge misst, misst diese:

| Signal | Frage | Beim KFZ-Rechner |
|---|---|---|
| **Latenz** | Wie lange dauert eine Anfrage? | Antwortzeit der Berechnung |
| **Verkehr** | Wie viel Last liegt an? | Anfragen pro Minute |
| **Fehler** | Wie viel geht schief? | Anteil der Antworten mit Status 5xx |
| **Sättigung** | Wie voll ist das System? | Speicher, Verbindungspool, Warteschlange |

**Latenz getrennt nach erfolgreichen und fehlgeschlagenen Anfragen messen.**

Ein Systemfehler wird oft sehr schnell zurückgegeben. Mischt man beides, sinkt die Durchschnittslatenz genau dann, wenn das System kaputtgeht — und die Kurve sieht aus, als hätte sich etwas verbessert.
Überhaupt ist der **Durchschnitt bei Latenzen die falsche Kennzahl**. Bei 1.000 Anfragen mit 50 ms und 10 Anfragen mit 8 Sekunden liegt der Mittelwert bei rund 130 ms — unauffällig. Zehn Anwender haben trotzdem acht Sekunden gewartet. Deshalb misst man **Perzentile**: p50, p95, p99. Das p99 sagt: Ein Prozent der Anwender hat es mindestens so schlecht erwischt.

## 13.4 Brauchbare Logs

```
Schlecht:
  ERROR: something went wrong

Besser:
  2026-07-29T14:32:07Z ERROR Prämienberechnung fehlgeschlagen
  vertrag=4711 sf_klasse=-3 fehler=ValueError anfrage_id=a7f3c9

Am besten - strukturiert:
  {"zeit":"2026-07-29T14:32:07Z","stufe":"ERROR",
   "text":"Prämienberechnung fehlgeschlagen","vertrag":"4711",
   "sf_klasse":-3,"fehler":"ValueError","anfrage_id":"a7f3c9"}
```

Strukturierte Logs sind maschinell auswertbar. Statt in Textmengen zu suchen, filtert man nach Feldern: alle Fehler des Typs `ValueError` der letzten Stunde, gruppiert nach Vertrag.

### 13.4.1 Die Korrelations-ID

Der wichtigste Einzelkniff: Jede Anfrage bekommt beim Eintritt eine eindeutige Kennung, die durch alle Systeme mitwandert und in jeder Logzeile steht.

```python
import uuid
from contextvars import ContextVar

anfrage_id: ContextVar[str] = ContextVar("anfrage_id", default="-")


def middleware(request):
    # Vorhandene ID übernehmen oder neue erzeugen
    kennung = request.headers.get("X-Request-ID") or str(uuid.uuid4())[:8]
    anfrage_id.set(kennung)
    antwort = verarbeite(request)
    antwort.headers["X-Request-ID"] = kennung
    return antwort
```

Damit lässt sich der Weg einer einzelnen Anfrage über Reverse Proxy, Anwendung und Datenbank hinweg vollständig zusammensetzen. Ohne diese Kennung ist die Fehlersuche in verteilten Systemen Ratearbeit.

**Keine personenbezogenen Daten in Logs.**

Namen, Adressen, Geburtsdaten, Vertragsinhalte, vollständige Anfragekörper — all das gehört nicht in ein Protokoll. Logs werden zentral gesammelt, breit lesbar gemacht, lange aufbewahrt und selten gelöscht. Damit entsteht faktisch eine zweite, schlecht geschützte Datenhaltung.
Statt der Daten protokolliert man **Verweise**: eine Vertragsnummer statt der Vertragsdaten, eine Kundennummer statt des Namens. Wer den Zusammenhang wirklich braucht, kann ihn über das Fachsystem herstellen — protokolliert und berechtigt.
Das gilt auch für Fehlermeldungen mit vollständigem Stacktrace: Dort landen erschreckend häufig ganze Eingabedaten.

## 13.5 Alarmierung

Die schwierigste Disziplin. Zu wenige Alarme, und Störungen fallen Anwendern zuerst auf. Zu viele, und niemand schaut mehr hin.

### 13.5.1 Auf Symptome alarmieren, nicht auf Ursachen

```
Schlecht (Ursache):        Besser (Symptom):
  CPU über 80 %              p95 der Antwortzeit über 2 s
  Speicher über 90 %         Fehlerrate über 1 %
  Festplatte zu 85 % voll    Anfragen schlagen fehl
```

Eine CPU-Auslastung von 90 Prozent ist kein Problem, solange die Anwender nichts merken — es kann auch schlicht bedeuten, dass die Maschine gut ausgelastet ist. Umgekehrt kann alles im grünen Bereich sein, während die Anwendung wegen eines Sperrproblems in der Datenbank steht.

**Die Regel:** Alarmiert wird auf das, was Anwender spüren. Ursachenmetriken bleiben in den Diagrammen, damit man sie bei der Analyse zur Hand hat.

Eine Ausnahme bestätigt die Regel: Vorlaufende Größen mit sicherem Ausgang. Ein Zertifikat, das in vierzehn Tagen abläuft, oder ein Dateisystem, das bei aktueller Wachstumsrate in drei Tagen voll ist — dort ist die Ursache selbst der Alarm, weil das Symptom zu spät käme.

### 13.5.2 Jeder Alarm braucht eine Handlung

**Wichtig**

**Prüffrage für jeden Alarm:** *Was genau soll die Person tun, die um drei Uhr nachts geweckt wird?*
Gibt es keine sinnvolle Antwort, ist es kein Alarm, sondern ein Diagramm. Diagramme wecken niemanden.
Alarmmüdigkeit ist eine ernste Betriebsgefahr: Wer in der Woche vierzig Meldungen bekommt, von denen achtunddreißig folgenlos sind, entwickelt zwangsläufig die Gewohnheit, sie wegzuklicken. Und klickt irgendwann auch die zwei echten weg.

## 13.6 Lab: Der KFZ-Rechner wird messbar

src/metriken.py

```python
"""Prometheus-Metriken für den KFZ-Rechner."""
from prometheus_client import Counter, Histogram, Gauge

ANFRAGEN = Counter(
    "kfz_anfragen_gesamt",
    "Anzahl der Berechnungsanfragen",
    ["ergebnis"],                      # erfolg | fehler
```

```
)

DAUER = Histogram(
    "kfz_berechnungsdauer_sekunden",
    "Dauer einer Prämienberechnung",
    buckets=(0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5),
)

DB_VERBINDUNGEN = Gauge(
    "kfz_db_verbindungen_aktiv",
    "Aktuell belegte Datenbankverbindungen",
)
```

src/server.py

```
from prometheus_client import generate_latest, CONTENT_TYPE_LATEST

from src.metriken import ANFRAGEN, DAUER
from src.praemie import berechne_praemie


@app.route("/api/praemie")
def api_praemie():
    with DAUER.time():
        try:
            betrag = berechne_praemie(
                int(request.args["sf"]),
                int(request.args["alter"]),
            )
            ANFRAGEN.labels(ergebnis="erfolg").inc()
            return {"betrag": str(betrag)}
        except (ValueError, KeyError) as fehler:
            ANFRAGEN.labels(ergebnis="fehler").inc()
            app.logger.warning(
                "Berechnung fehlgeschlagen",
                extra={"fehler": type(fehler).__name__},
            )
            return {"fehler": "ungültige Eingabe"}, 400


@app.route("/metrics")
def metrics():
    return generate_latest(), 200, {"Content-Type": CONTENT_TYPE_LATEST}


@app.route("/health")
def health():
    """Lebt der Prozess?"""
    return {"status": "ok"}


@app.route("/ready")
def ready():
    """Kann der Dienst Anfragen bearbeiten – inklusive Abhängigkeiten?"""
    if not datenbank_erreichbar():
        return {"status": "nicht bereit"}, 503
    return {"status": "bereit"}
```

**Tipp**

Die Trennung von `/health` und `/ready` ist wichtiger, als sie aussieht. **Health** beantwortet: Lebt der Prozess? Ist die Antwort nein, muss neu gestartet werden. **Readiness**

beantwortet: Kann der Dienst gerade arbeiten? Ist die Antwort nein — etwa weil die Datenbank hängt — soll er **keinen** Verkehr bekommen, aber auch **nicht** neu gestartet werden.
Wer beides zusammenlegt, bekommt eine hübsche Fehlerkaskade: Die Datenbank ist kurz nicht erreichbar, sämtliche Instanzen werden für ungesund erklärt und neu gestartet, und der Neustart erzeugt einen Ansturm auf die ohnehin überlastete Datenbank.

### 13.6.1 Prometheus und Alarmregeln

prometheus.yml

```yaml
scrape_configs:
  - job_name: kfz-rechner
    scrape_interval: 15s
    static_configs:
      - targets: ["kfz-rechner:8080"]
```

alarme.yml

```yaml
groups:
  - name: kfz-rechner
    rules:
      - alert: HoheFehlerrate
        expr: |
          sum(rate(kfz_anfragen_gesamt{ergebnis="fehler"}[5m]))
          / sum(rate(kfz_anfragen_gesamt[5m])) > 0.05
        for: 5m
        labels:
          severity: kritisch
        annotations:
          summary: "Fehlerrate über 5 Prozent"
          beschreibung: "Seit 5 Minuten schlagen mehr als 5 % der Berechnungen fehl."
          runbook: "https://creutz.spdns.de/doku.php/edv:devops:13_incident_management"

      - alert: LangsameAntworten
        expr: |
          histogram_quantile(0.95,
            sum(rate(kfz_berechnungsdauer_sekunden_bucket[5m])) by (le)
          ) > 1
        for: 10m
        labels:
          severity: warnung
        annotations:
          summary: "p95 der Berechnungsdauer über 1 Sekunde"

      - alert: DienstNichtErreichbar
        expr: up{job="kfz-rechner"} == 0
        for: 2m
        labels:
          severity: kritisch
        annotations:
          summary: "KFZ-Rechner antwortet nicht"
```

**Aufgaben:**

1. Bring Prometheus und Grafana als Container zum Laufen und lege ein Dashboard mit den vier goldenen Signalen an.

2. Erzeuge künstlich Fehler und beobachte, wann der Alarm auslöst.
3. Verändere `for: 5m` auf `for: 30s`. Was passiert bei kurzen Lastspitzen?
4. Formuliere für jeden der drei Alarme in einem Satz, was die geweckte Person tun soll. Fällt dir zu einem nichts ein, lösche ihn.
5. Ergänze eine Messung der Sättigung — etwa der belegten Datenbankverbindungen.

## 13.7 Anti-Pattern

| **Anti-Pattern** | **Folge** | **Abhilfe** |
|---|---|---|
| Alarm auf Ursachen | viele Meldungen ohne Anwenderwirkung | auf Symptome alarmieren |
| Durchschnittslatenz | Ausreißer verschwinden | Perzentile |
| Alarme ohne Handlungsanweisung | Ratlosigkeit um drei Uhr nachts | Runbook verlinken |
| Zu viele Alarme | Ermüdung, echte Meldungen gehen unter | ausdünnen, `for`-Zeiten nutzen |
| Dashboard mit vierzig Diagrammen | niemand liest es | ein Übersichtsbild, Details dahinter |
| Personenbezogene Daten in Logs | Datenschutzverstoß | Verweise statt Inhalte |
| Unstrukturierte Logs | nicht auswertbar | JSON, feste Felder |
| `/health` prüft auch die Datenbank | Neustartkaskade | Health und Readiness trennen |
| Erst überwachen, wenn es brennt | Störung ohne Datenlage | Instrumentierung von Anfang an |

## 13.8 Reflexionsfrage

*Wie viele Alarme habt ihr in der vergangenen Woche bekommen, und bei wie vielen davon musste tatsächlich jemand handeln?*

Liegt das Verhältnis unter einem Drittel, ist die Alarmierung ein Problem für sich — unabhängig davon, wie gut die Überwachung funktioniert.

# Kapitel 14

# Incident-Management

**Lernziel**

Nach diesem Kapitel kennst du die Rollen und den Ablauf einer Störungsbearbeitung, kannst ein schuldfreies Postmortem schreiben und erklären, warum die Frage nach dem Schuldigen die Sicherheit eines Systems verringert. Du hast ein Runbook und ein Postmortem selbst verfasst.

## 14.1 Keine Panik

Der Reiseführer im Anhalter durch die Galaxis trägt bekanntlich zwei beruhigende Worte auf dem Umschlag, und zwar aus einem guten Grund: In einer unübersichtlichen Lage ist Panik die einzige Reaktion, die die Lage zuverlässig verschlimmert.

Für Störungen im IT-Betrieb gilt dasselbe, und es ist keine Charakterfrage. Wer unter Druck improvisiert, trifft schlechtere Entscheidungen — das ist gut untersucht. Die Gegenmaßnahme heißt nicht Gelassenheit, sondern **Vorbereitung**: klare Rollen, festgelegte Abläufe, geschriebene Runbooks. Alles, was vorher entschieden wurde, muss während der Störung nicht mehr entschieden werden.

## 14.2 Rollen

Auch bei kleinen Teams lohnt sich die Trennung — notfalls übernimmt eine Person mehrere Rollen, aber bewusst.

| Rolle | Aufgabe | Tut ausdrücklich nicht |
|---|---|---|
| **Incident Commander** | koordiniert, entscheidet, behält den Überblick | selbst am System arbeiten |
| **Operations** | arbeitet technisch an der Behebung | mit Stakeholdern kommunizieren |
| **Communications** | informiert Fachbereich, Leitung, Anwender | technische Entscheidungen treffen |
| **Scribe** | protokolliert Zeitpunkte, Maßnahmen, Erkenntnisse | mitdiskutieren |

**Die wichtigste Regel: Der Incident Commander tippt nicht.**

Der Reflex ist verständlich — die erfahrenste Person will helfen und legt selbst Hand an. Damit verliert die Störung ihre Koordination. Niemand hat mehr den Überblick, niemand spricht mit dem Fachbereich, und drei Personen probieren gleichzeitig widersprüchliche Dinge aus.
Der Incident Commander ist die Person mit dem besten Überblick, nicht mit dem meisten Fachwissen. Bei kleinen Störungen darf das derselbe Mensch sein — dann aber mit bewusstem Rollenwechsel und nicht nebenbei.

## 14.3 Schweregrade

| Stufe | Bedeutung | Reaktion |
|---|---|---|
| **Sev 1** | Totalausfall, Geschäftsbetrieb steht | sofort, auch nachts, alle Hebel |
| **Sev 2** | wesentliche Funktion gestört, Umgehung möglich | sofort während der Geschäftszeit |
| **Sev 3** | eingeschränkt, einzelne betroffen | am nächsten Arbeitstag |
| **Sev 4** | Schönheitsfehler | im normalen Arbeitsvorrat |

Die Einstufung erfolgt **nach Auswirkung, nicht nach technischer Dramatik**. Ein ausgefallener Datenbankknoten, den der Cluster abgefangen hat, ist Sev 3. Ein Rechtschreibfehler im Prämienbescheid, der an 40.000 Kunden ging, kann Sev 1 sein.

Wichtig ist außerdem: **Im Zweifel höher einstufen.** Eine Herabstufung nach zehn Minuten ist billig; eine zu spät erkannte Sev 1 ist es nicht.

## 14.4 Der Ablauf

```
1. ERKENNEN      Alarm, Anruf, Beobachtung
      |          -> Störung ausrufen, Rollen besetzen, Kanal öffnen
      v
2. EINDÄMMEN     Auswirkung begrenzen
      |          -> Rollback, Feature-Flag aus, Verkehr umlenken
      v
3. WIEDERHER-    Normalbetrieb zurückholen
   STELLEN       -> nicht die Ursache beheben - den Betrieb!
      |
      v
4. AUFARBEITEN   Postmortem, Maßnahmen, Umsetzung
                 -> innerhalb weniger Tage, solange Erinnerung frisch
```

**Schritt 2 und 3 kommen vor der Ursachenanalyse.**

Der häufigste und teuerste Fehler bei Störungen ist der Versuch, erst zu verstehen und dann zu handeln. Neugier ist eine Berufstugend — während einer laufenden Störung ist sie eine Ausfallzeitverlängerung.

Erst wird zurückgerollt, umgeleitet oder abgeschaltet. Verstanden wird danach, in Ruhe, mit vollständigen Logs und ohne wartende Anwender. Die Logs laufen nicht weg; die Geduld des Fachbereichs schon.
Die einzige Ausnahme: Wenn völlig unklar ist, ob eine Maßnahme die Lage verbessert oder verschlimmert, ist kurzes Innehalten richtig. Das ist aber Abwägung, nicht Ursachenforschung.

## 14.5 Runbooks

Ein Runbook ist eine Handlungsanweisung für einen bekannten Störungsfall — geschrieben für den ungünstigsten Fall: Es ist drei Uhr nachts, die Person hat Bereitschaft, kennt dieses System nur oberflächlich und ist gerade aufgewacht.

docs/runbooks/kfz-hohe-fehlerrate.md

```
# Runbook: KFZ-Rechner, hohe Fehlerrate

**Alarm:** HoheFehlerrate
**Schweregrad:** Sev 2 (Sev 1, wenn über 50 % und länger als 15 Minuten)
**Zuständig:** Team Bestandssysteme

## 1. Lage feststellen

    curl -s https://kfz.intern/health
    journalctl --user -u kfz-rechner.service --since "15 min ago" | tail -50
    podman inspect kfz-rechner --format '{{.ImageName}}'

Grafana-Dashboard: https://grafana.intern/d/kfz

## 2. Häufigste Ursachen, nach Wahrscheinlichkeit

| Beobachtung im Log             | Ursache                 | Maßnahme           |
|--------------------------------|-------------------------|--------------------|
| ‘connection refused‘ zur DB    | Datenbank nicht da      | Abschnitt 3.1      |
| ‘ValueError‘ gehäuft           | fehlerhaftes Deployment | Abschnitt 3.2      |
| ‘pool exhausted‘               | Verbindungen erschöpft  | Abschnitt 3.3      |
| keine Auffälligkeit            | vorgelagertes System    | Abschnitt 3.4      |

## 3. Maßnahmen

### 3.1 Datenbank nicht erreichbar
    systemctl status postgresql   # auf db-prod01
Ist die DB tatsächlich aus: Eskalation an Team Datenbanken, Sev 1.

### 3.2 Fehlerhaftes Deployment
Rollback (Dauer etwa 30 Sekunden):
    cd ~/kfz-betrieb && git revert HEAD && git push
Der GitOps-Abgleich zieht binnen zwei Minuten nach.
Beschleunigen: ‘systemctl --user start gitops-abgleich.service‘

### 3.3 Verbindungspool erschöpft
    systemctl --user restart kfz-rechner.service
Tritt das mehr als einmal pro Woche auf: Ticket für Poolgröße anlegen.

### 3.4 Keine Auffälligkeit
Vorgelagerten Dienst prüfen, dann Eskalation an den Incident Commander.

## 4. Eskalation
```

```
- Rufbereitschaft Bestandssysteme: siehe Dienstplan
- Team Datenbanken: siehe Dienstplan
- Ab Sev 1: Leitung Anwendungsbetrieb informieren

## 5. Nach der Störung

Postmortem anlegen, wenn die Störung länger als 30 Minuten dauerte
oder Kunden betroffen waren.
```

**Tipp**

Runbooks veralten schneller als jede andere Dokumentation. Der zuverlässigste Weg, sie aktuell zu halten: **Jedes Postmortem prüft das verwendete Runbook und korrigiert es.** War keines vorhanden, entsteht eines. Damit wächst die Sammlung genau dort, wo tatsächlich Störungen auftreten — und nicht dort, wo jemand einmal Zeit hatte.

## 14.6 Das schuldfreie Postmortem

Nach der Störung folgt die Aufarbeitung. Ihr Zweck ist ausschließlich, **Wiederholung zu verhindern** — nicht, Verantwortung zuzuweisen.

### 14.6.1 Warum Schuldsuche schadet

Das ist kein Wohlfühlargument, sondern ein sicherheitstechnisches. Wo Fehler Konsequenzen für Einzelne haben, geschieht Folgendes:

- Beinahe-Vorfälle werden nicht mehr gemeldet — genau die Informationen, die am meisten wert sind.
- Menschen sichern sich ab, statt Ursachen zu benennen.
- Die Analyse endet beim menschlichen Fehler und nicht bei der Frage, warum das System diesen Fehler zuließ.

**Wichtig**

**Der zentrale Perspektivwechsel:** „Ein Kollege hat das falsche Kommando ausgeführt"ist kein Ergebnis, sondern der Anfang der eigentlichen Untersuchung.
Die Anschlussfragen lauten: Warum war dieses Kommando so leicht zu verwechseln? Warum gab es keine Rückfrage vor einem zerstörenden Vorgang? Warum hatte diese Sitzung Produktionsrechte? Warum fiel es erst nach vierzig Minuten auf?
**Menschliches Versagen ist nie die Ursache. Es ist das Symptom eines Systems, das Versagen zugelassen hat.** Denn der nächste Mensch an dieser Stelle wird denselben Fehler machen — und Personalwechsel ist keine Sicherheitsmaßnahme.
Hinzu kommt der Rückschaufehler: Im Nachhinein wirkt die richtige Entscheidung offensichtlich. Während der Störung war sie es nicht. Ein brauchbares Postmortem rekonstruiert, welche Informationen den Beteiligten **zum jeweiligen Zeitpunkt** vorla-

gen.

### 14.6.2 Vorlage

docs/postmortems/2026-07-29-kfz-ausfall.md

```
# Postmortem: Ausfall KFZ-Rechner am 29.07.2026

**Schweregrad:** Sev 2
**Dauer:** 47 Minuten (14:32 - 15:19)
**Betroffen:** etwa 2.400 Berechnungsanfragen, davon 340 abgebrochen
**Verfasser:** [Name]     **Status:** abgeschlossen

## Zusammenfassung

Nach dem Ausrollen von Version 2.5.0 schlugen alle Berechnungen mit
SF-Klasse 0 fehl. Ursache war eine Division durch null in der neuen
Rabattlogik. Behoben durch Rollback auf 2.4.8.

## Zeitlicher Ablauf

| Zeit  | Ereignis                                                    |
|-------|-------------------------------------------------------------|
| 14:28 | Version 2.5.0 ausgerollt, Pipeline grün                     |
| 14:32 | Alarm "HoheFehlerrate" ausgelöst                            |
| 14:35 | Bereitschaft übernimmt, Störung ausgerufen                  |
| 14:41 | Zusammenhang mit dem Deployment erkannt                     |
| 14:44 | Entscheidung: Rollback                                      |
| 15:02 | Rollback abgeschlossen, Rückfrage im Team hatte verzögert   |
| 15:19 | Fehlerrate wieder normal, Störung geschlossen               |

## Auswirkung

340 Anwender erhielten eine Fehlermeldung. Keine falschen Prämien
berechnet, keine Datenverluste.

## Was ist passiert

Die neue Wenigfahrer-Logik teilt durch den SF-Faktor. Bei SF-Klasse 0
ist dieser Faktor 0. Der Fall war in den Tests nicht abgedeckt, weil
die Testdaten nur SF-Klassen ab 1 enthielten.

## Was gut lief

- Der Alarm griff nach vier Minuten.
- Das Runbook führte direkt zum Rollback-Abschnitt.
- Der Fachbereich war nach acht Minuten informiert.

## Was schlecht lief

- Das Rollback dauerte 18 statt der erwarteten 2 Minuten, weil
  unklar war, wer es freigeben muss.
- Die Testdaten deckten den Randfall nicht ab.
- Der Canary-Schritt war für diesen Dienst nicht eingerichtet.

## Wo wir Glück hatten

Der Fehler trat sofort auf. Bei einem Fehler, der nur jede tausendste
Berechnung falsch macht, hätten wir ihn womöglich wochenlang nicht
bemerkt - mit falsch berechneten Prämien als Folge.

## Maßnahmen
```

```
| # | Maßnahme                                          | Wer    | Bis        |
|---|---------------------------------------------------|--------|------------|
| 1 | Testdaten um SF-Klasse 0 und Randfälle ergänzen | A. K.  | 05.08.2026 |
| 2 | Rollback-Freigabe im Runbook eindeutig regeln   | M. R.  | 02.08.2026 |
| 3 | Canary-Schritt für den KFZ-Rechner einrichten   | Team   | 31.08.2026 |
| 4 | Grenzwerttests als Pflicht in die Definition of |        |            |
|   | Done aufnehmen                                    | Team   | 15.08.2026 |
```

Die beiden Abschnitte **„Was gut lief"** und **„Wo wir Glück hatten"** werden gern weggelassen. Beide sind wertvoll: Der erste bewahrt funktionierende Praktiken davor, im Zuge von Verbesserungen versehentlich abgeschafft zu werden. Der zweite deckt Risiken auf, die diesmal folgenlos blieben — und beim nächsten Mal nicht.

## 14.7 Bereitschaft

| Regel | Begründung |
|---|---|
| Bereitschaft wird vergütet oder ausgeglichen | sonst wird sie stillschweigend abgebaut |
| Wer entwickelt, ist auch erreichbar | schärft das Interesse an guter Alarmierung |
| Höchstens ein bis zwei Weckrufe pro Bereitschaft | mehr ist ein Zeichen fehlender Stabilität |
| Nach einer Nachtstörung: Ausgleich am Folgetag | Übermüdete verursachen Folgestörungen |
| Jeder Weckruf wird ausgewertet | Ursache beheben statt Symptom ertragen |

Die dritte Zeile ist die wichtigste Kennzahl der Bereitschaft. Wer regelmäßig fünfmal pro Nacht geweckt wird, hat kein Bereitschaftsproblem, sondern ein Qualitätsproblem — und Bereitschaft ist dann lediglich das Verfahren, mit dem es verwaltet statt gelöst wird.

## 14.8 Lab

**Aufgabe 1 — Runbook.** Schreibe ein Runbook für einen Alarm aus Kapitel 12. Gib es jemandem, der das System nicht kennt, und lass es durcharbeiten. Jede Rückfrage ist eine Lücke.

**Aufgabe 2 — Störungsübung.** Verabredet euch zu einer festen Zeit. Eine Person baut heimlich einen Fehler ein — falsches Image, gestoppte Datenbank, gefülltes Dateisystem. Der Rest bearbeitet die Störung mit besetzten Rollen. Stoppt die Zeit bis zur Erkennung und bis zur Wiederherstellung.

**Aufgabe 3 — Postmortem.** Schreibe zur Übungsstörung ein vollständiges Postmortem nach der Vorlage. Achte darauf, dass in der Ursachenbeschreibung kein Name vorkommt.

**Aufgabe 4 — Prüfung.** Lies ein echtes Postmortem aus deinem Betrieb, falls vorhanden. Endet es bei einem menschlichen Fehler oder bei einer Systemeigenschaft?

## 14.9 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Ursachensuche vor Wiederherstellung | Ausfallzeit verlängert sich | erst eindämmen |
| Incident Commander tippt mit | niemand koordiniert | Rollen trennen |
| Postmortem endet beim Menschen | dieselbe Störung kommt wieder | nach Systemursachen fragen |
| Maßnahmen ohne Termin und Verantwortliche | werden nie umgesetzt | Tabelle mit Wer und Bis |
| Kein Postmortem bei „nur kurzenSStörungen | Beinahe-Vorfälle gehen verloren | feste Schwelle vereinbaren |
| Runbooks veralten | Bereitschaft steht ohne Anleitung da | im Postmortem mitpflegen |
| Bereitschaft ohne Ausgleich | Erschöpfung, Kündigungen | vergüten und begrenzen |
| Alle telefonieren durcheinander | Erkenntnisse gehen verloren | ein Kanal, ein Protokoll |

## 14.10 Reflexionsfrage

*Denk an die letzte größere Störung in deinem Umfeld. Gibt es dazu ein schriftliches Ergebnis — und wurde daraus tatsächlich etwas umgesetzt?*

Ein Postmortem ohne umgesetzte Maßnahmen ist ein Aufsatz. Die Maßnahmentabelle ist der einzige Teil, der Wirkung entfaltet.

# Teil V

# Teil E — Wie man absichert

# Kapitel 15

# DevSecOps

**Lernziel**

Nach diesem Kapitel kennst du die Prüfverfahren SAST, DAST, SCA und IaC-Scanning mit ihren jeweiligen Stärken, kannst eine Lieferkette absichern und weißt, warum ein Scanner ohne Bearbeitungsverfahren wertlos ist. Der KFZ-Rechner hat eine abgesicherte Pipeline mit Stückliste und Signatur.

## 15.1 Sicherheit als Flaschenhals

Der klassische Ablauf: Die Anwendung ist fertig, dann kommt die Sicherheitsprüfung. Sie dauert drei Wochen, findet vierzig Befunde, und die Hälfte davon lässt sich nur mit erheblichem Umbau beheben. Der Termin steht aber. Also werden Ausnahmen genehmigt, und man nimmt sich vor, das später zu beheben — wobei „späterëin Fachbegriff für „nieïst.

DevSecOps überträgt das Shift-Left-Prinzip aus Kapitel 4 auf die Sicherheit: Prüfungen finden **fortlaufend und automatisiert** statt, nicht als Torwächter am Ende.

**Wichtig**

**Der wichtigste Satz dieses Kapitels:** Sicherheit, die den Fluss blockiert, wird umgangen. Nicht aus Böswilligkeit, sondern weil der Liefertermin ebenfalls eine Vorgabe ist.
Ein Sicherheitsprozess, der drei Wochen dauert, erzeugt Ausnahmegenehmigungen. Einer, der drei Minuten in der Pipeline dauert, erzeugt behobene Befunde. Die zweite Variante findet weniger Einzelfälle, verhindert aber mehr.

## 15.2 Die vier Prüfverfahren

| Verfahren | Prüft | Zeitpunkt | Findet | Findet nicht |
|---|---|---|---|---|
| **SAST** | Quellcode | beim Commit | Injektionen, unsichere Aufrufe | Konfigurationsfehler, Laufzeitprobleme |
| **SCA** | Abhängigkeiten | beim Bauen | bekannte Schwachstellen in Bibliotheken | eigene Fehler |
| **DAST** | laufende Anwendung | in der Testumgebung | Fehlkonfiguration, Authentifizierungslücken | Fehler in selten genutzten Pfaden |
| **IaC-Scan** | Infrastrukturcode | beim Commit | offene Ports, fehlende Verschlüsselung | Anwendungsfehler |

Dazu kommen **Secret Scanning** (Zugangsdaten im Repository) und **Container-Scanning** (Schwachstellen im Image).

Keines dieser Verfahren ersetzt die anderen. SAST kennt den Code, aber nicht die Umgebung. DAST kennt die Umgebung, aber nicht den Code. SCA kennt beides nicht, dafür aber die Schwachstellendatenbanken — und der überwiegende Teil der ausnutzbaren Lücken in typischen Anwendungen steckt nicht im Eigenanteil, sondern in den mitgebrachten Bibliotheken.

## 15.3 Lieferkette und Stückliste

Eine moderne Anwendung besteht zu einem großen Teil aus fremdem Code. Eine mittelgroße Python- oder JavaScript-Anwendung bringt schnell mehrere hundert Pakete mit, die überwiegend niemand je angesehen hat.

**Ein SBOM** (Software Bill of Materials) ist die Stückliste dazu: eine maschinenlesbare Aufstellung aller enthaltenen Komponenten mit Version und Herkunft.

```
# SBOM erzeugen
syft packages dir:. -o cyclonedx-json > sbom.json

# Gegen bekannte Schwachstellen prüfen
grype sbom:sbom.json

# Image signieren
cosign sign --key cosign.key registry.intern/kfz-rechner:2.5.0

# Signatur prüfen - vor dem Ausrollen
cosign verify --key cosign.pub registry.intern/kfz-rechner:2.5.0
```

**Der praktische Wert eines SBOM zeigt sich am Tag einer neuen kritischen Schwachstelle.**

Die Frage lautet dann: „Sind wir betroffen?Öhne Stückliste beginnt eine Suchaktion über alle Systeme, die Tage dauert und deren Ergebnis niemand garantieren kann. Mit Stückliste ist es eine Abfrage von Sekunden — und die Antwort ist belastbar.
Wer die Log4j-Woche im Dezember 2021 miterlebt hat, braucht keine weitere Begründung. Wer nicht, dem sei gesagt: Der überwiegende Teil des Aufwands entfiel damals nicht auf das Patchen, sondern auf die Frage, wo das Ding überhaupt überall steckte.

## 15.4 Abhängigkeiten pflegen

```
# Versionen festnageln - reproduzierbare Builds
pip-compile requirements.in --generate-hashes -o requirements.txt
```

Automatische Aktualisierungswerkzeuge wie **Renovate** oder **Dependabot** legen Pull Requests für neue Versionen an. Zusammen mit einer belastbaren Testsuite aus Kapitel 4 wird daraus ein funktionierendes Verfahren: Der Pull Request kommt automatisch, die Pipeline prüft, ein Mensch führt zusammen.

**Achtung**

**Die Gegenrichtung ist eine Falle.** Automatische Aktualisierungen ohne Tests bedeuten, dass fremder Code ungeprüft in die eigene Anwendung wandert. Angriffe über kompromittierte Pakete sind real und nehmen zu — von der übernommenen Bibliothek bis zum Tippfehler-Paket, das dem echten zum Verwechseln ähnlich sieht.
Merke: Automatische Aktualisierung braucht automatische Prüfung. Sonst tauscht man ein bekanntes Risiko gegen ein unbekanntes.

## 15.5 Umgang mit Befunden

Der häufigste Fehler beim Einführen von Scannern: Man schaltet sie ein, bekommt 1.400 Befunde und schaltet sie wieder aus.

**Ein tragfähiges Vorgehen:**

1. **Basislinie ziehen.** Alle bestehenden Befunde werden zunächst dokumentiert, aber nicht blockierend gestellt.
2. **Ab jetzt keine neuen.** Die Pipeline wird rot bei **neuen** Befunden ab „hoch". Das ist die entscheidende Regel — sie verhindert Verschlechterung ohne den Betrieb lahmzulegen.
3. **Bestand abarbeiten.** Ein festes Kontingent pro Sprint, nach Risiko sortiert.
4. **Ausnahmen befristen.** Jede Ausnahme bekommt Begründung, Verantwortlichen und Ablaufdatum.

```
schwachstellen:
  stage: scannen
  script:
    # Bestand: nur berichten
    - trivy image --exit-code 0 --severity LOW,MEDIUM "$IMAGE:$TAG"
    # Neue kritische Befunde: Pipeline rot
    - trivy image --exit-code 1 --severity HIGH,CRITICAL
        --ignorefile .trivyignore "$IMAGE:$TAG"
```

.trivyignore

```
# Befristete Ausnahmen - jede mit Begründung und Ablaufdatum!
# CVE-2024-12345  bis 2026-09-30, Fix erfordert Umstieg auf lib 4.x,
#                 Ticket SEC-231, nicht erreichbar (kein Netzwerkpfad)
CVE-2024-12345
```

Nicht jeder Befund ist ein Risiko. Eine Schwachstelle in einer Bibliothek, deren betroffene Funktion die Anwendung gar nicht aufruft, ist real vorhanden und praktisch bedeutungslos. Umgekehrt kann ein mittelschwerer Befund an einer exponierten Stelle dringlicher sein als ein kritischer im Innern. **Bewertung braucht Kontext**, und der Scanner hat ihn nicht.

## 15.6 Bedrohungsmodellierung

Werkzeuge finden bekannte Muster. Konstruktionsfehler finden sie nicht — dafür braucht es ein Gespräch, und zwar vor dem Bauen.

Vier Fragen, die für den Anfang genügen:

1. Woran arbeiten wir? (Ein Bild reicht.)
2. Was kann schiefgehen?
3. Was tun wir dagegen?
4. Haben wir gute Arbeit geleistet?

Als Gedankenstütze für die zweite Frage dient **STRIDE**:

| Buchstabe | Bedrohung | Beim KFZ-Rechner |
|---|---|---|
| **S**poofing | Identitätsvortäuschung | Kann jemand fremde Vertragsdaten abrufen? |
| **T**ampering | Manipulation | Kann die berechnete Prämie verändert werden? |
| **R**epudiation | Abstreitbarkeit | Ist nachweisbar, wer welche Berechnung veranlasst hat? |
| **I**nformation Disclosure | Informationsabfluss | Landen Vertragsdaten in Logs? |
| **D**enial of Service | Verfügbarkeit | Was passiert bei 10.000 Anfragen pro Sekunde? |
| **E**levation of Privilege | Rechteausweitung | Kann ein Anwender Verwaltungsfunktionen erreichen? |

Eine Stunde mit einem Whiteboard zu Beginn ersetzt keine Werkzeuge, findet aber die Sorte Fehler, die kein Werkzeug je findet.

## 15.7 Lab: Die abgesicherte Pipeline

.gitlab-ci.yml

```
stages: [pruefen, testen, bauen, scannen, ausrollen]

# ---------- schnelle Prüfungen ----------
geheimnisse:
  stage: pruefen
  image: zricethezav/gitleaks:latest
  script:
    - gitleaks detect --source . --redact --verbose

sast:
  stage: pruefen
  image: python:3.12-slim
  script:
    - pip install --quiet bandit
    - bandit -r src/ -f json -o bandit.json || true
    - bandit -r src/ -ll          # blockiert ab Schweregrad "medium"
  artifacts:
    paths: [bandit.json]
    when: always

iac-scan:
  stage: pruefen
  image: bridgecrew/checkov:latest
  script:
    - checkov -d ansible/ --compact --quiet
    - checkov -f Containerfile --framework dockerfile

# ---------- Abhängigkeiten ----------
sca:
  stage: testen
  image: python:3.12-slim
  script:
    - pip install --quiet pip-audit
    - pip-audit -r requirements.txt --strict

# ---------- Stückliste und Signatur ----------
sbom:
  stage: bauen
  image: anchore/syft:latest
  script:
    - syft "$IMAGE:$TAG" -o cyclonedx-json > sbom.json
  artifacts:
    paths: [sbom.json]
    expire_in: 1 year          # für die Nachweisführung aufbewahren

signieren:
  stage: bauen
  image: gcr.io/projectsigstore/cosign:latest
  script:
    - echo "$COSIGN_KEY" > /tmp/cosign.key
    - cosign sign --key /tmp/cosign.key "$IMAGE:$TAG"
  after_script:
    - shred -u /tmp/cosign.key
```

```
# ---------- Container prüfen ----------
container-scan:
  stage: scannen
  image: aquasec/trivy:latest
  script:
    - trivy image --exit-code 0 --severity LOW,MEDIUM "$IMAGE:$TAG"
    - trivy image --exit-code 1 --severity HIGH,CRITICAL "$IMAGE:$TAG"
```

**Aufgaben:**

1. Bring alle Stufen zum Laufen. Wie lange dauert die Pipeline jetzt insgesamt?
2. Baue absichtlich einen SQL-String per Zeichenkettenverkettung. Findet `bandit` ihn?
3. Trage ein Passwort in eine Datei ein und committe. Findet `gitleaks` es? Nimm den Commit zurück — und wechsle das Passwort trotzdem.
4. Erzeuge die Stückliste und suche darin eine Bibliothek, von deren Existenz du nichts wusstest.
5. Setze eine Ausnahme in `.trivyignore` mit Begründung und Ablaufdatum.
6. Ergänze in der Ausrollstufe eine Signaturprüfung, die verhindert, dass unsignierte Images starten.

## 15.8 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Sicherheitsprüfung erst am Ende | Befunde kommen, wenn ihre Behebung teuer ist | Shift-Left |
| Scanner ohne Bearbeitungsverfahren | 1.400 Befunde, die niemand ansieht | Basislinie, dann keine neuen |
| Alles blockierend stellen | Pipeline dauerhaft rot, wird umgangen | nach Schweregrad staffeln |
| Unbefristete Ausnahmen | „temporär"wird dauerhaft | Ablaufdatum verpflichtend |
| Automatische Updates ohne Tests | fremder Code ungeprüft in Produktion | Tests als Voraussetzung |
| Geheimnisse im Repository | Kompromittierung | Secret Scanning, Tresor |
| Kein SBOM | „Sind wir von X betroffen?"bleibt tagelang offen | Stückliste je Build |
| Sicherheit als eigene Abteilung mit Vetorecht | Umgehung, Schattenprozesse | Security Champions in den Teams |

## 15.9 Reflexionsfrage

*Wenn morgen früh eine kritische Schwachstelle in einer weit verbreiteten Bibliothek bekannt wird — wie lange braucht ihr für die Antwort auf die Frage, ob und wo ihr betroffen seid?*

# Teil VI

# Teil F — Wie man misst

# Kapitel 16

# DORA-Metriken, SLI, SLO und Error Budget

**Lernziel**

Nach diesem Kapitel kennst du die vier DORA-Kennzahlen, kannst sie in deinem Umfeld erheben und weißt, warum Geschwindigkeit und Stabilität kein Gegensatz sind. Du kannst SLI, SLO und SLA unterscheiden und erklären, wozu ein Fehlerbudget dient.

**Wichtig**

**Achtung, Namensgleichheit.** In diesem Kapitel geht es um die **DORA-Metriken** aus der Forschungsgruppe *DevOps Research and Assessment*. Damit hat die **DORA-Verordnung** der EU — der Digital Operational Resilience Act aus Kapitel 17 — nichts zu tun. Zwei völlig verschiedene Dinge mit demselben Kürzel. In einem Haus, in dem beide vorkommen, lohnt es sich, das im Gespräch klarzustellen.

## 16.1 Die vier Kennzahlen

Die Forschungsgruppe um Nicole Forsgren, Jez Humble und Gene Kim hat über Jahre tausende Organisationen befragt und dabei vier Kennzahlen identifiziert, die leistungsfähige von weniger leistungsfähigen IT-Organisationen unterscheiden. Nachzulesen in *Accelerate* und den jährlichen *State of DevOps*-Berichten.

| Kennzahl | Frage | Misst |
|---|---|---|
| **Deployment-Frequenz** | Wie oft liefert ihr aus? | Durchsatz |
| **Lead Time for Changes** | Wie lange von Commit bis Produktion? | Durchsatz |
| **Change Failure Rate** | Welcher Anteil der Auslieferungen verursacht eine Störung? | Stabilität |
| **Time to Restore Service** | Wie lange bis zur Wiederherstellung? | Stabilität |

Später kam als fünfte Größe die **Zuverlässigkeit** hinzu — im Sinne der Einhaltung der eigenen Zusagen, was zum SLO-Teil dieses Kapitels führt.

### 16.1.1 Größenordnungen

| Kennzahl | Hoch | Mittel | Niedrig |
|---|---|---|---|
| Deployment-Frequenz | mehrmals täglich | wöchentlich bis monatlich | seltener als monatlich |
| Lead Time | unter einem Tag | Woche bis Monat | mehr als ein Monat |
| Change Failure Rate | unter 15 % | 15–30 % | über 30 % |
| Time to Restore | unter einer Stunde | unter einem Tag | Tage bis Wochen |

Die genauen Schwellen verschieben sich von Bericht zu Bericht — als Orientierung taugen sie trotzdem. Wichtiger als der Vergleich mit anderen ist ohnehin der Vergleich mit dem eigenen Wert vom letzten Quartal.

## 16.2 Die zentrale Erkenntnis

**Geschwindigkeit und Stabilität sind kein Gegensatz.**

Das ist das überraschendste und wichtigste Ergebnis der DORA-Forschung. Die intuitive Annahme lautet: Wer schneller ausliefert, geht mehr Risiko ein. Die Daten zeigen das Gegenteil — dieselben Organisationen, die häufig und schnell ausliefern, haben **auch** die niedrigeren Fehlerraten und die kürzeren Wiederherstellungszeiten.

Der Zusammenhang ist plausibel, wenn man ihn einmal gesehen hat: Häufige Auslieferung bedeutet **kleine** Auslieferung. Kleine Änderungen sind leichter zu prüfen, brechen seltener, und wenn doch, ist die Ursache offensichtlich. Die dafür nötigen Fähigkeiten — Testautomatisierung, Rollback, Überwachung — verbessern gleichzeitig die Stabilität.

Die Umkehrung gilt ebenso: Wer aus Vorsicht seltener ausliefert, sammelt Änderungen an und macht jede einzelne Auslieferung riskanter. Die Vorsicht erzeugt genau das Risiko, das sie vermeiden soll.

Daraus folgt die Regel für den Umgang mit den Kennzahlen: **Immer paarweise betrachten.** Eine gestiegene Deployment-Frequenz allein sagt nichts. Zusammen mit einer gleichbleibenden Fehlerrate sagt sie sehr viel.

## 16.3 Messen statt schätzen

Alle vier Kennzahlen lassen sich aus vorhandenen Daten gewinnen.

scripts/dora.sh

```
#!/usr/bin/env bash
#
# Grobe Erhebung der DORA-Kennzahlen aus Git und der Pipeline.
#
set -euo pipefail

ZEITRAUM="${1:-90}"   # Tage
SEIT="$(date -d "${ZEITRAUM} days ago" +%Y-%m-%d)"

echo "=== Zeitraum: letzte ${ZEITRAUM} Tage ==="

# --- 1. Deployment-Frequenz: Tags auf produktiven Releases ---
ANZAHL=$(git tag --list 'v*' --sort=-creatordate \
         --format='%(creatordate:short) %(refname:short)' \
         | awk -v seit="$SEIT" '$1 >= seit' | wc -l)
echo "Deployment-Frequenz: ${ANZAHL} in ${ZEITRAUM} Tagen"
echo "  = $(echo "scale=2; ${ANZAHL}/${ZEITRAUM}" | bc) pro Tag"

# --- 2. Lead Time: Commit-Zeit bis Tag-Zeit ---
echo
echo "Lead Time je Release (Stunden):"
git tag --list 'v*' --sort=-creatordate --format='%(refname:short)' \
  | head -20 | while read -r tag; do
    tag_zeit=$(git log -1 --format=%ct "$tag")
    erster=$(git log --format=%ct "${tag}" --not "${tag}^{}~20" 2>/dev/null | tail -1)
    [[ -z "$erster" ]] && continue
    echo "  ${tag}: $(( (tag_zeit - erster) / 3600 ))"
done

# --- 3. Change Failure Rate: Anteil der Releases mit Revert/Hotfix ---
FEHLER=$(git log --since="$SEIT" --oneline \
         --grep='^revert' --grep='^hotfix' --grep='^fix!' -i -E | wc -l)
echo
echo "Change Failure Rate: ${FEHLER} von ${ANZAHL}"

# --- 4. Time to Restore: aus dem Störungssystem, hier nicht automatisierbar ---
echo
echo "Time to Restore: aus Postmortems/Ticketsystem ermitteln"
```

**Tipp**

Die Erhebung muss nicht perfekt sein. Eine grobe, konsistent erhobene Zahl ist unendlich viel wertvoller als eine exakte, die niemand erhebt. Wichtig ist nur, dass das Verfahren über die Zeit gleich bleibt — sonst misst man die Änderung des Messverfahrens statt die Änderung der Leistung.

**Kennzahlen dienen der Selbststeuerung, nicht dem Teamvergleich.**

Sobald Teams anhand ihrer Deployment-Frequenz verglichen oder gar bewertet werden, steigt die Deployment-Frequenz. Nicht weil besser gearbeitet wird, sondern weil man Änderungen aufteilt, leere Auslieferungen erzeugt oder Störungen nicht mehr als solche verbucht.
Auch der Vergleich zwischen Teams ist selten sinnvoll: Ein Team, das an einer Weboberfläche arbeitet, hat andere Rahmenbedingungen als eines, das ein Kernsystem mit Nachtverarbeitung betreut. Der einzige belastbare Vergleich ist der mit dem eigenen Vorquartal.

## 16.4 SLI, SLO und SLA

| Begriff | Bedeutung | Beispiel |
|---|---|---|
| **SLI** — Indicator | die gemessene Größe | Anteil der Anfragen unter 500 ms |
| **SLO** — Objective | das interne Ziel | 99,5 % der Anfragen unter 500 ms im Monat |
| **SLA** — Agreement | die vertragliche Zusage mit Folgen | 99,0 %, sonst Gutschrift |

Die Reihenfolge ist wichtig: **Ein SLO liegt immer strenger als das SLA.** Der Abstand dazwischen ist der Sicherheitspuffer, der zum Handeln zwingt, bevor eine vertragliche Zusage verletzt wird.

### 16.4.1 Was Verfügbarkeitszahlen bedeuten

| Verfügbarkeit | Ausfall pro Monat | Ausfall pro Jahr |
|---|---|---|
| 99 % | 7,2 Stunden | 3,65 Tage |
| 99,5 % | 3,6 Stunden | 1,83 Tage |
| 99,9 % | 43 Minuten | 8,76 Stunden |
| 99,95 % | 22 Minuten | 4,38 Stunden |
| 99,99 % | 4,3 Minuten | 52,6 Minuten |
| 99,999 % | 26 Sekunden | 5,26 Minuten |

**Jede weitere Neun kostet ungefähr das Zehnfache.**

Bei 99,99 Prozent bleiben im gesamten Monat 4,3 Minuten Ausfall. Das schließt geplante Wartung, Netzwerkstörungen, Fehler bei Ausrollvorgängen und die Reaktionszeit eines Menschen mit ein. Praktisch bedeutet es: keine manuellen Eingriffe mehr, redundante Auslegung über Standorte hinweg, automatische Umschaltung.
Die Frage an den Fachbereich lautet deshalb nie „Wollen Sie hohe Verfügbarkeit?"— die Antwort ist immer ja. Sie lautet: **„Was kostet Sie eine Stunde Ausfall, und wie viel darf die Vermeidung kosten?"** Erst diese Frage erzeugt eine belastbare Anforderung.

## 16.5 Das Fehlerbudget

Ein eleganter Gedanke: Wenn das Ziel bei 99,9 Prozent liegt, sind 0,1 Prozent Ausfall **erlaubt**. Das sind rund 43 Minuten im Monat — und dieses Budget ist eine Ressource, die man ausgeben darf.

```
Fehlerbudget im Monat (SLO 99,9 %) = 43 Minuten

Budget noch offen        ->  Neues ausrollen, experimentieren,
                             Risiko eingehen
Budget aufgebraucht      ->  Ausrollstopp für Funktionen,
```

```
                                nur noch Stabilisierung
```

Der Nutzen liegt darin, dass ein alter Konflikt zu einer Rechenaufgabe wird. Statt „die Entwicklung will zu schnell"gegen „der Betrieb bremst"gibt es eine gemeinsame Zahl. Ist das Budget offen, hat die Entwicklung freie Fahrt — der Betrieb kann nicht ohne Begründung bremsen. Ist es aufgebraucht, wird stabilisiert — auch wenn der Fachbereich drängt.

Bemerkenswert ist die andere Richtung: **Ein dauerhaft ungenutztes Fehlerbudget ist ebenfalls ein Befund.** Es bedeutet, dass das SLO zu locker gewählt ist oder dass zu vorsichtig gearbeitet wird. Beides kostet Geld, nur eben unsichtbar.

Die zugehörige Regel — was genau passiert, wenn das Budget aufgebraucht ist — muss **vorher** vereinbart und von der Leitung getragen sein. Eine Regel, die im Ernstfall verhandelt wird, ist keine.

## 16.6 Lab

**Aufgabe 1.** Erhebe die vier DORA-Kennzahlen für ein Projekt aus deinem Umfeld, notfalls durch Zählen von Hand für die letzten drei Monate.

**Aufgabe 2.** Formuliere für den KFZ-Rechner ein SLO. Beispiel:

```
SLI:  Anteil der Anfragen an /api/praemie mit HTTP 2xx und
      Antwortzeit unter 500 ms
SLO:  99,5 % über 30 Tage rollierend
Fehlerbudget: 0,5 %  ->  bei 500.000 Anfragen im Monat
      sind das 2.500 Anfragen, die schiefgehen dürfen
Regel: Bei über 75 % verbrauchtem Budget keine neuen Funktionen
      mehr, nur noch Stabilisierung.
```

**Aufgabe 3.** Baue das SLO als Prometheus-Regel auf Basis von Kapitel 12 und stelle den Budgetverbrauch in Grafana dar.

```
- record: kfz:sli_erfolgsrate
  expr: |
    sum(rate(kfz_anfragen_gesamt{ergebnis="erfolg"}[30d]))
    / sum(rate(kfz_anfragen_gesamt[30d]))

- record: kfz:fehlerbudget_verbraucht
  expr: (1 - kfz:sli_erfolgsrate) / 0.005

- alert: FehlerbudgetFastAufgebraucht
  expr: kfz:fehlerbudget_verbraucht > 0.75
  labels:
    severity: warnung
  annotations:
    summary: "Über 75 Prozent des Fehlerbudgets verbraucht"
```

**Aufgabe 4.** Diskutiert im Team: Was wäre bei euch die Konsequenz eines aufgebrauchten Budgets — und wer müsste dem zustimmen?

## 16.7 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| Kennzahlen zum Teamvergleich | Zahlen werden optimiert, nicht die Arbeit | zur Selbststeuerung nutzen |
| Nur Durchsatz messen | Tempo auf Kosten der Stabilität | immer paarweise |
| SLO ohne Fehlerbudgetregel | schöne Zahl ohne Konsequenz | Regel vorher vereinbaren |
| SLO strenger als nötig | teuer, ohne dass es jemand merkt | am Bedarf ausrichten |
| SLO = SLA | kein Puffer bis zur Vertragsverletzung | SLO strenger ansetzen |
| Verfügbarkeit als einzige Kennzahl | langsame, aber erreichbare Systeme gelten als gut | Latenz und Fehlerrate ergänzen |
| Messen ohne Handeln | Bericht wird erstellt und abgelegt | feste Auswertung im Quartal |
| Perfekte Messung anstreben | man beginnt nie | grob anfangen, verfeinern |

## 16.8 Reflexionsfrage

*Welche der vier Kennzahlen könntest du für deinen Bereich heute Nachmittag erheben — und welche Zahl erwartest du?*

Der Vergleich zwischen Erwartung und Ergebnis ist oft der lehrreichste Teil der Übung.

# Kapitel 17

# Wertstromanalyse und DMAIC

**Lernziel**

Nach diesem Kapitel kannst du eine Wertstromanalyse durchführen, Engpässe erkennen und den DMAIC-Zyklus als eines von mehreren Verbesserungsverfahren einordnen. Du weißt, warum es sinnlos ist, an einer Stelle zu optimieren, die nicht der Engpass ist.

## 17.1 Warum an der richtigen Stelle optimiert werden muss

In Kapitel 1 stand die Rechnung: zwei Arbeitstage Bearbeitung, 43 Tage Warten. Dieses Kapitel liefert das Handwerkszeug, um solche Zahlen systematisch zu erheben und daraus Maßnahmen abzuleiten.

Die zugrundeliegende Einsicht stammt aus der **Engpasstheorie**: In jedem Ablauf gibt es genau eine Stelle, die den Durchsatz begrenzt. Jede Verbesserung an einer anderen Stelle ist wirkungslos — sie erzeugt nur größere Warteschlangen vor dem Engpass.

```
Anforderung --> Entwicklung --> TEST --> Freigabe --> Ausrollen
                  3 Tage       14 Tage    2 Tage      1 Tag
                                 ^
                             der Engpass

Entwicklung wird doppelt so schnell:  3 -> 1,5 Tage
Gesamtdauer:  20 -> 18,5 Tage.  Gewonnen: 7 Prozent.

Testumgebung automatisiert:  14 -> 1 Tag
Gesamtdauer:  20 -> 7 Tage.  Gewonnen: 65 Prozent.
```

Beide Verbesserungen kosten ähnlich viel Aufwand. Nur eine wirkt.

## 17.2 Die Wertstromanalyse

Das Verfahren stammt aus der schlanken Produktion und ist auf IT-Abläufe unmittelbar übertragbar.

### 17.2.1 Vorgehen

1. **Rahmen festlegen.** Wo beginnt und endet die Betrachtung? Üblich: von der Anforderung bis zur produktiven Nutzung.
2. **Schritte aufnehmen.** Jeden Arbeitsschritt in der tatsächlichen Reihenfolge, nicht in der beschriebenen.
3. **Zeiten erheben.** Je Schritt Bearbeitungszeit und Wartezeit davor.
4. **Übergaben markieren.** Jeder Wechsel der zuständigen Person oder Abteilung.
5. **Qualität erfassen.** Wie oft muss ein Schritt wiederholt werden?
6. **Auswerten.** Flow-Effizienz, Engpass, größter Einzelhebel.

### 17.2.2 Die Kennzahlen

```
Bearbeitungszeit  = Summe der reinen Arbeitszeit
Durchlaufzeit     = Anfang bis Ende, inklusive Warten
Flow-Effizienz    = Bearbeitungszeit / Durchlaufzeit

Erstdurchlaufquote = Anteil der Schritte ohne Nacharbeit
```

| Flow-Effizienz | Einordnung |
|---|---|
| unter 5 % | üblich, keineswegs ungewöhnlich |
| 5–15 % | bereits verbessert |
| 15–25 % | gut |
| über 25 % | sehr gut |

### 17.2.3 Beispiel: KFZ-Rechner, vorher und nachher

| Schritt | Vorher: Bearbeitung / Warten | Nachher |
|---|---|---|
| Anforderung aufnehmen | 2 h / 5 Tage | 2 h / 1 Tag |
| Entwickeln | 6 h / 3 Tage | 6 h / 0 |
| Code Review | 1 h / 4 Tage | 1 h / 3 h |
| Testumgebung bereitstellen | 15 min / 9 Tage | automatisch / 4 min |
| Testen | 4 h / 2 Tage | 12 min automatisch / 0 |
| Änderungsantrag und Freigabe | 1 h / 14 Tage | 15 min / 2 h (Standardänderung) |
| Ausrollen | 3 h / 6 Tage | 4 min automatisch / 0 |
| **Durchlaufzeit** | **etwa 45 Tage** | **etwa 1,5 Tage** |
| **Flow-Effizienz** | **etwa 3,5 %** | **etwa 25 %** |

Die Verbesserungen stammen ausschließlich aus den vorangegangenen Kapiteln — Automatisierung, Testpipeline, GitOps, Standardänderungen. Nichts davon ist besonders raffiniert; die Wirkung entsteht durch die Kombination.

**Führe die Analyse mit den Beteiligten durch, nicht über sie hinweg.**

Ein Wertstrom, den die Prozessabteilung im stillen Kämmerlein zeichnet, bildet den *beschriebenen* Ablauf ab. Der tatsächliche sieht regelmäßig anders aus — mit Umgehungen, Absprachen auf dem Flur und Schritten, die nur einer bestimmten Person bekannt sind.
Zwei Stunden mit allen Beteiligten an einer Wand mit Haftnotizen liefern ein ehrlicheres Bild als vier Wochen Interviews. Und sie erzeugen nebenbei etwas, das noch wertvoller ist: das gemeinsame Verständnis, dass niemand einzeln schuld ist.

## 17.3 Angefangene Arbeit begrenzen

Eine der wirkungsvollsten und billigsten Maßnahmen überhaupt — und eine der unbeliebtesten.

**Das Gesetz von Little** beschreibt den Zusammenhang:

```
Durchlaufzeit = angefangene Arbeit / Durchsatz

20 begonnene Aufgaben, 4 werden pro Woche fertig  -> 5 Wochen
8 begonnene Aufgaben,  4 werden pro Woche fertig  -> 2 Wochen
```

Wer weniger gleichzeitig anfängt, wird schneller fertig. Das widerspricht dem Gefühl, weil eine begonnene Aufgabe wie Fortschritt aussieht. Sie ist aber erst dann Wert, wenn sie fertig ist — vorher ist sie Bestand, der Aufmerksamkeit bindet.

Dazu kommen die Umschaltkosten: Wer an drei Dingen gleichzeitig arbeitet, verliert einen erheblichen Teil seiner Zeit an das Wiedereinfinden. Bei fünf parallelen Vorgängen ist der produktive Anteil je Vorgang erschreckend gering.

## 17.4 DMAIC

DMAIC ist der Kernzyklus des Qualitätsmanagementansatzes **Six Sigma**. Er ist ein strukturiertes, datengetriebenes Verbesserungsverfahren — und einer von mehreren möglichen Rahmen.

| Phase | Frage | Beim KFZ-Rechner |
|---|---|---|
| **D**efine | Was ist das Problem? | Durchlaufzeit 45 Tage, Fachbereich unzufrieden |
| **M**easure | Wie groß ist es? | Wertstromanalyse, DORA-Kennzahlen erhoben |
| **A**nalyse | Woran liegt es? | Engpass: Bereitstellung der Testumgebung, 9 Tage Wartezeit |
| **I**mprove | Was ändern wir? | Umgebung per IaC automatisch bereitstellen |
| **C**ontrol | Bleibt es so? | Durchlaufzeit monatlich messen, Abweichung meldet sich |

**Die Phase, die am häufigsten übersprungen wird, ist Control.**

Eine Verbesserung, die nicht überwacht wird, verfällt. Der automatisierte Ablauf bekommt eine Ausnahme, dann eine zweite, und nach einem Jahr ist der alte Zustand wiederhergestellt — nur mit zusätzlicher Werkzeugschicht.
Control heißt konkret: Die Kennzahl aus der Measure-Phase wird weiterhin erhoben, und es gibt einen vereinbarten Schwellwert, bei dessen Unterschreitung jemand handelt. Ohne diesen Teil ist DMAIC eine Projektmethode statt eines Verbesserungszyklus.

### 17.4.1 Wann DMAIC passt und wann nicht

| Passt zu | Passt nicht zu |
|---|---|
| wiederkehrenden, messbaren Abläufen | einmaligen Vorhaben |
| Problemen mit unklarer Ursache | Problemen mit offensichtlicher Ursache |
| Umgebungen, in denen Daten vorliegen | Situationen ohne Messgrundlage |
| Verbesserungen mit Formalisierungsbedarf | schnellen Experimenten |

Der bürokratische Aufwand von DMAIC ist nicht gering. Für die Frage „warum dauert die Testumgebung neun Tage"reicht oft ein Nachmittag mit den Beteiligten und ein Ishikawa-Diagramm. Für ein Problem, das seit zwei Jahren wiederkehrt und dessen Ursache niemand kennt, ist der strukturierte Zyklus dagegen die richtige Wahl.

## 17.5 Verwandte Verfahren im Überblick

| Verfahren | Kern | Passt zu |
|---|---|---|
| **PDCA** | Plan-Do-Check-Act, kleiner Zyklus | kontinuierliche kleine Verbesserungen |
| **DMAIC** | datengetrieben, fünf Phasen | wiederkehrende, messbare Probleme |
| **A3** | ein Problem auf einer Seite | Verdichtung, Führungskommunikation |
| **Kaizen** | Haltung der laufenden Verbesserung | Kultur, nicht Methode |
| **Retrospektive** | regelmäßige Teamreflexion | agile Teams, kurzer Takt |
| **5 Warum** | fünfmal nachfragen | schnelle Ursachensuche |
| **Ishikawa** | Ursachen nach Kategorien | Sammlung möglicher Ursachen |

Die Wahl ist weniger wichtig als die Regelmäßigkeit. Ein Team, das jede zweite Woche eine halbe Stunde ehrlich zurückblickt und **eine** Maßnahme daraus umsetzt, verbessert sich zuverlässiger als eines mit einem ausgefeilten Verfahren, das zweimal im Jahr angewendet wird.

## 17.6 Lab: Wertstrom-Workshop

**Vorbereitung:** Haftnotizen in drei Farben, eine freie Wand, alle Beteiligten des Ablaufs, zwei Stunden Zeit.

1. **Schritte sammeln** (Farbe 1). Jeder schreibt die Schritte auf, die er selbst ausführt. Anschließend gemeinsam in Reihenfolge bringen.
2. **Zeiten ergänzen** (Farbe 2). Je Schritt Bearbeitungs- und Wartezeit. Schätzen ist erlaubt — grob und ehrlich schlägt genau und geschönt.
3. **Probleme markieren** (Farbe 3). Wo gibt es Nacharbeit, Rückfragen, Blockaden?
4. **Rechnen.** Durchlaufzeit, Bearbeitungszeit, Flow-Effizienz.
5. **Engpass bestimmen.** Wo steckt die längste Wartezeit?
6. **Eine Maßnahme wählen.** Genau eine, mit Verantwortlichem und Termin.
7. **Nachmessen.** Nach vier Wochen dieselbe Analyse für einen neuen Vorgang.

**Tipp**

Der sechste Punkt ist der schwierigste, und zwar aus einem menschlichen Grund: Nach zwei Stunden Analyse hat man zwölf Verbesserungsideen und will alle umsetzen. Genau das führt dazu, dass keine davon fertig wird — und bestätigt nebenbei das Prinzip aus Abschnitt 16.3 an der Verbesserungsarbeit selbst.
Eine Maßnahme, die in vier Wochen nachweislich gewirkt hat, ist mehr wert als zwölf angefangene.

## 17.7 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| An der falschen Stelle optimieren | Aufwand ohne Wirkung | Engpass zuerst bestimmen |
| Wertstrom ohne die Beteiligten | bildet den Wunsch ab, nicht die Wirklichkeit | gemeinsam an der Wand |
| Zwölf Maßnahmen gleichzeitig | keine wird fertig | eine, mit Termin |
| Control-Phase weglassen | Verbesserung verfällt | Kennzahl weiter erheben |
| Nur Bearbeitungszeit betrachten | die Wartezeit bleibt unsichtbar | beides erheben |
| Methodenstreit | Diskussion statt Verbesserung | irgendein Verfahren, aber regelmäßig |
| Verbesserung ohne Zeitbudget | „wenn mal Luft ist"— also nie | feste Kapazität einplanen |

## 17.8 Reflexionsfrage

*Wo steckt in deinem Ablauf die längste Wartezeit — und was wäre nötig, um sie zu halbieren?*

Bemerkenswert oft lautet die Antwort auf den zweiten Teil nicht „mehr Personal", sondern „eine Entscheidung, die niemand trifft".

# Teil VII

# Teil G — Im regulierten Umfeld

# Kapitel 18

# DevOps im regulierten Umfeld

**Lernziel**

Nach diesem Kapitel kannst du erklären, warum Continuous Delivery und aufsichtsrechtliche Anforderungen kein Widerspruch sind, kennst die Rolle der Standardänderung als zentralen Hebel und weißt, wie sich Funktionstrennung und Vier-Augen-Prinzip technisch statt organisatorisch erzwingen lassen.

**Achtung**

**Keine Rechtsberatung.** Dieses Kapitel ordnet technische Verfahren in einen regulatorischen Zusammenhang ein. Es ersetzt weder die Abstimmung mit der eigenen Compliance-Abteilung noch die Prüfung durch die Innenrevision. Regulatorische Anforderungen ändern sich; der hier beschriebene Stand ist zu verifizieren.

## 18.1 Der vermeintliche Widerspruch

Die verbreitete Annahme lautet: Automatisierte Auslieferung und Aufsichtsrecht schließen einander aus. Wer mehrmals täglich ausliefert, kann kein Vier-Augen-Prinzip einhalten, keine Funktionstrennung gewährleisten und keine Nachweise führen.

Das Gegenteil ist der Fall — und zwar nicht als Behauptung, sondern aus der Natur der Sache:

| Anforderung | Manueller Betrieb | Automatisierte Kette |
|---|---|---|
| Wer hat die Änderung veranlasst? | Ticket, sofern gepflegt | Git-Commit mit Signatur |
| Wer hat freigegeben? | Unterschrift oder Mailfreigabe | Pull-Request-Genehmigung, unveränderlich |
| Was läuft in Produktion? | Vermutung, Inventarliste | Image-Digest, kryptografisch eindeutig |
| Wurde getestet? | Testprotokoll, händisch | Pipeline-Protokoll, automatisch |
| Ist die Konfiguration wie vorgesehen? | letzte Prüfung vor acht Monaten | laufender Soll-Ist-Abgleich |
| Kann zurückgerollt werden? | Sicherung, Dauer unbekannt | gemessen, geübt, dokumentiert |

**Wichtig**

**Der entscheidende Punkt:** In einer automatisierten Kette fallen die Nachweise als **Nebenprodukt** an. Niemand muss sie erzeugen, niemand kann sie vergessen, und niemand kann sie nachträglich verändern.
Ein manueller Prozess mit sorgfältig geführten Formularen ist nachweislich schlechter dokumentiert als eine Pipeline, die nichts dokumentiert, aber alles protokolliert. Der Unterschied liegt in der Fälschungssicherheit und in der Vollständigkeit — ein Formular kann man vergessen, einen Commit nicht.

## 18.2 Der regulatorische Rahmen

### 18.2.1 DORA — die Verordnung

Die **Verordnung (EU) 2022/2554**, der *Digital Operational Resilience Act*, gilt seit dem <cite index="17-1»17. Januar 2025</cite> unmittelbar. Sie regelt für Finanzunternehmen — Banken, Versicherungen, Kapitalverwaltungsgesellschaften, Zahlungsdienstleister und weitere — den Umgang mit IKT-Risiken.

Fünf Themenbereiche:

1. IKT-Risikomanagement
2. Behandlung und Meldung IKT-bezogener Vorfälle
3. Testen der digitalen operationalen Widerstandsfähigkeit
4. Management des Risikos durch IKT-Drittdienstleister
5. Informationsaustausch

**Wichtig**

**Wichtig für die Einordnung:** Um Doppelregulierung zu vermeiden, hat die BaFin die nationalen Rundschreiben <cite index="20-1»KAIT, VAIT und ZAIT mit Ablauf des 16. Januar 2025 aufgehoben</cite>. Für Versicherungsunternehmen im Anwendungsbereich ist damit **DORA** das maßgebliche Regelwerk, nicht mehr die VAIT.
Die **BAIT** werden <cite index="20-1»schrittweise aufgehoben</cite>; für Institute, die ein IKT-Risikomanagement nach DORA betreiben müssen, gelten sie bereits seit dem 17. Januar 2025 nicht mehr.
Praktisch bleiben viele Inhalte der aufgehobenen Rundschreiben relevant: Was dort gefordert wurde, findet sich überwiegend in DORA wieder, und etabliertes Vorgehen aus der VAIT-Zeit gilt weiterhin als gute Praxis. Wer in älteren Unterlagen auf VAIT-Verweise stößt, sollte sie aber gegen die entsprechenden DORA-Artikel abgleichen.

### 18.2.2 Weitere Rahmenwerke

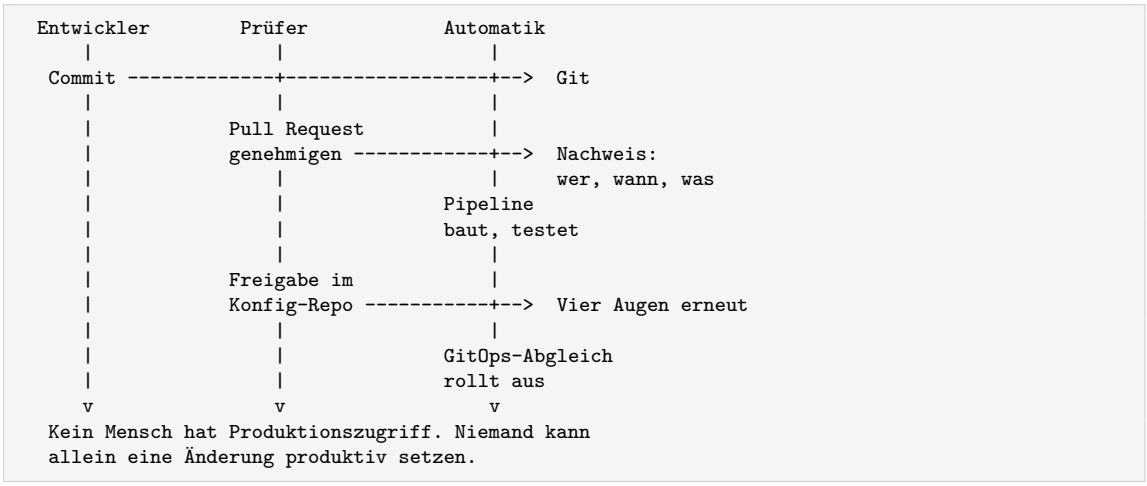
| Rahmen | Betrifft | Bezug zu DevOps |
|---|---|---|
| **DSGVO** | personenbezogene Daten | Testdaten, Logs, Aufbewahrung |
| **ISO 27001** | Informationssicherheit allgemein | Änderungsmanagement, Zugriffskontrolle |
| **BSI IT-Grundschutz** | öffentlicher Sektor, oft freiwillig | Härtung, Dokumentation |
| **ITIL 4** | Betriebsprozesse | Änderungs- und Störungsmanagement |
| **KRITIS / NIS-2** | kritische Infrastrukturen | Meldepflichten, Nachweise |

## 18.3 Funktionstrennung und Vier-Augen-Prinzip

Die Kernanforderung: **Wer eine Änderung entwickelt, darf sie nicht allein in Produktion bringen.**

Das klingt nach einem Ausschlusskriterium für Continuous Deployment — ist aber nur eines für **unbeaufsichtigtes** Continuous Deployment. Continuous **Delivery** aus Kapitel 6 erfüllt die Anforderung vollständig.

### 18.3.1 Technische Umsetzung



| Anforderung | Technische Entsprechung |
|---|---|
| Vier-Augen-Prinzip | Branch-Schutz mit erforderlicher Genehmigung |
| Entwickler ≠ Freigeber | Pull Request kann nicht selbst genehmigt werden |
| Funktionstrennung | Kein Personenkonto hat Produktionszugriff — nur die Automatik |
| Nachvollziehbarkeit | Git-Historie, signierte Commits, Pipeline-Protokolle |
| Unveränderlichkeit | geschützte Zweige, keine Historienänderung möglich |

**Tipp**

**Der stärkste Einzelhebel:** Wenn kein Mensch mehr auf Produktion schreiben kann, ist die Funktionstrennung nicht mehr eine Regel, die eingehalten werden soll, sondern eine Eigenschaft des Systems.
Das ist qualitativ etwas anderes. Eine organisatorische Regel kann umgangen werden — im Notfall, unter Zeitdruck, mit den besten Absichten. Eine technische Eigenschaft kann es nicht. Und für die Prüfung ist der Unterschied erheblich: Statt Stichproben in Protokollen zu ziehen, lässt sich die Berechtigungsstruktur einmal belegen.

## 18.4 Die Standardänderung — der zentrale Hebel

Hier liegt der praktische Schlüssel zum gesamten Kapitel. ITIL kennt drei Änderungsarten:

| Art | Merkmal | Freigabe |
|---|---|---|
| **Normal Change** | Einzelfall, Risikobewertung nötig | Änderungsgremium |
| **Standard Change** | wiederkehrend, vorab genehmigt, geringes Risiko | **vorab genehmigt** |
| **Emergency Change** | dringend, nachträgliche Aufarbeitung | Notfallverfahren |

Der Engpass in der Wertstromanalyse aus Kapitel 16 — vierzehn Tage Wartezeit bis zur nächsten Gremiumssitzung — betrifft ausschließlich den **Normal Change**.

Eine **Standardänderung** ist einmal bewertet und dauerhaft genehmigt. Sie wird nicht erneut vorgelegt; sie wird protokolliert.

### 18.4.1 Was ein Antrag auf Standardänderung enthalten muss

- eine genaue Beschreibung des Änderungstyps und seiner Abgrenzung
- die Risikobewertung, warum das Risiko gering ist
- den vollständigen automatisierten Ablauf inklusive aller Prüfschritte
- das Rückrollverfahren mit **gemessener** Dauer
- die anfallenden Nachweise und ihre Aufbewahrung
- die Abbruchbedingungen — wann greift das Verfahren nicht?

**Ein Formulierungsbeispiel, das sich bewährt hat:**

*„Änderungen an der Anwendung KFZ-Rechner, die über die definierte Pipeline ausgeliefert werden, gelten als Standardänderung, sofern: die Pipeline vollständig grün ist, ein Vier-Augen-Review im Pull Request dokumentiert ist, keine Datenbankmigration mit Datenverlustpotenzial enthalten ist, keine Schnittstelle zu Drittsystemen verändert wird und das Rückrollverfahren verfügbar ist. Trifft eine dieser Bedingungen nicht zu, gilt die Änderung als Normal Change."*
Der letzte Satz ist der wichtigste. Er macht aus der Standardänderung keine Generalermächtigung, sondern eine klar abgegrenzte Kategorie — und genau das ist es, was ein Prüfer sehen will.

Das ist der Punkt, an dem sich die Durchlaufzeit tatsächlich ändert. Alles Vorherige in dieser Unterlage — Tests, Pipeline, Rollback, Überwachung — dient letztlich der Begründung, warum das Risiko gering genug für diese Einstufung ist.

## 18.5 Nachweisführung

Was bei einer Prüfung tatsächlich gefragt wird, und woher die Antwort kommt:

| Frage | Quelle | Aufbewahrung |
|---|---|---|
| Welche Version läuft seit wann? | Git-Historie des Konfigurationsrepositories | dauerhaft |
| Wer hat entwickelt, wer freigegeben? | Commit und Pull-Request-Genehmigung | dauerhaft |
| Wurde getestet, mit welchem Ergebnis? | Pipeline-Protokoll, Testberichte | nach Vorgabe, oft mehrere Jahre |
| Welche Komponenten sind enthalten? | SBOM aus Kapitel 14 | je Release |
| Wurde auf Schwachstellen geprüft? | Scanner-Berichte | je Release |
| Wer hatte Zugriff auf Produktion? | Rechteverwaltung, Protokolle | laufend |
| Wie wurde eine Störung bearbeitet? | Postmortem aus Kapitel 13 | je Vorfall |
| Entspricht die Konfiguration der Vorgabe? | Ergebnis des Soll-Ist-Abgleichs | laufend |

**Tipp**

**Der Nachweis, der am meisten Eindruck macht**, ist der nächtliche Soll-Ist-Abgleich aus Kapitel 9. Er belegt nicht, dass die Konfiguration bei der letzten Prüfung stimmte, sondern dass sie **jeden Tag** geprüft wird und Abweichungen gemeldet werden.
Das ist die Antwort auf die unangenehmste Prüfungsfrage überhaupt: „Woher wissen Sie, dass sich seit der letzten Prüfung nichts geändert hat?"

Zwei technische Anforderungen an Protokolle: Sie müssen **unveränderlich** sein — Anhängen erlaubt, Ändern nicht — und sie brauchen eine **verlässliche Zeitquelle**. Ein Protokoll auf einem System, auf dem die Betroffenen Schreibrechte haben, ist als Nachweis wertlos.

## 18.6 Notfalländerungen

Der Fall, in dem alle Verfahren unter Druck geraten: Produktion steht, die Korrektur ist bekannt, das Gremium tagt am Donnerstag.

Ein tragfähiges Notfallverfahren regelt vorher:

1. **Wer darf auslösen?** Namentlich benannter Personenkreis, keine Improvisation.
2. **Was ist erlaubt?** Der reguläre Weg mit verkürzter Freigabe — nicht das Umgehen aller Prüfungen.
3. **Wie wird protokolliert?** Automatisch, mit Begründung im Commit.
4. **Was passiert danach?** Nachträgliche Vorlage, Postmortem, Prüfung binnen einer festen Frist.

**Die Häufigkeit von Notfalländerungen ist eine Kennzahl für sich.**

Ein einstelliger Anteil ist normal. Liegt der Anteil bei einem Drittel, ist der reguläre Weg zu langsam — und das Notfallverfahren dient nicht mehr dem Notfall, sondern der Umgehung. Das fällt bei jeder ernsthaften Prüfung auf, und es ist deutlich unangenehmer zu erklären als eine hohe Auslieferungsfrequenz.
Der richtige Schluss aus vielen Notfalländerungen ist nicht, das Notfallverfahren zu verschärfen, sondern den regulären Weg zu beschleunigen.

## 18.7 Datenschutz in der Auslieferungskette

Zusammengefasst, was in den vorherigen Kapiteln verstreut steht:

| Stelle | Risiko | Maßnahme | Kapitel |
|---|---|---|---|
| Testdaten | Produktivdaten im Testsystem | synthetische Daten | 4 |
| Logs | personenbezogene Daten in der zentralen Sammlung | Verweise statt Inhalte | 12 |
| Fehlermeldungen | Eingabedaten im Stacktrace | Ausgabe begrenzen | 12 |
| Sicherungen | Aufbewahrungsfristen | Löschkonzept, auch für Sicherungen | 6 |
| Pipeline-Protokolle | Daten in Testausgaben | keine echten Daten in Tests | 7 |
| Container-Images | Konfiguration mit Daten eingebacken | Konfiguration von außen | 10 |

## 18.8 Lab

**Aufgabe 1 — Antrag auf Standardänderung.** Formuliere für den KFZ-Rechner einen vollständigen Antrag nach der Gliederung aus 17.4. Achte besonders auf die Abbruchbedingungen: Welche Änderungen sollen ausdrücklich **nicht** als Standardänderung gelten?

**Aufgabe 2 — Prüfungssimulation.** Lass dir von jemandem die acht Fragen aus 17.5 stellen und beantworte sie ausschließlich mit Belegen aus deinen Systemen. Wo du improvisieren musst, fehlt ein Nachweis.

**Aufgabe 3 — Berechtigungen.** Prüfe für ein Produktivsystem: Welche Personenkonten haben Schreibzugriff? Ist jedes davon begründet? Was wäre nötig, um die Liste auf null zu bringen?

**Aufgabe 4 — Gegenprobe.** Nimm eine tatsächliche Produktivänderung der letzten Wochen und versuche, den vollständigen Weg von der Anforderung bis zum laufenden Prozess lückenlos zu belegen. Notiere jede Lücke.

**Tipp**

Aufgabe 4 ist die aufschlussreichste. In den meisten Umgebungen bricht die Kette an einer bestimmten Stelle — typischerweise zwischen „das Ticket wurde geschlossenünd „diese Binärdatei läuft auf diesem Server". Genau diese Lücke schließt der Image-Digest.

## 18.9 Anti-Pattern

| Anti-Pattern | Folge | Abhilfe |
|---|---|---|
| „Regulierung verbietet Automatisierung" | Stillstand mit falscher Begründung | Anforderungen lesen, nicht vermuten |
| Freigabe als Formalie ohne Kenntnis | Vier-Augen-Prinzip nur auf dem Papier | Änderungsliste automatisch beilegen |
| Notfalländerung als Regelweg | fällt bei jeder Prüfung auf | regulären Weg beschleunigen |
| Nachweise nachträglich erzeugen | lückenhaft, angreifbar | als Nebenprodukt anfallen lassen |
| Personenkonten mit Produktionszugriff | Funktionstrennung nicht belegbar | ausschließlich über die Automatik |
| Compliance erst am Projektende | teure Umbauten | Anforderungen im Entwurf klären |
| Protokolle auf dem geprüften System | als Nachweis wertlos | getrennte, unveränderliche Ablage |
| Papierprozess parallel zur Automatik | doppelter Aufwand, widersprüchliche Stände | den Papierweg abschaffen, nicht ergänzen |

## 18.10 Reflexionsfrage

*Welche einzelne aufsichtsrechtliche Anforderung nennt man in deinem Haus als Grund gegen häufigere Auslieferungen — und steht sie tatsächlich so in der Vorschrift, oder ist sie eine gewachsene Auslegung?*

Die Antwort ist häufiger das Zweite als das Erste. Und das ist eine gute Nachricht, denn Auslegungen lassen sich ändern.

# Teil VIII

# Anhang

# Kapitel 19

# Anti-Pattern im Überblick

*Sammlung aller Anti-Pattern aus dieser Unterlage — als Nachschlagewerk, zur Prüfungsvorbereitung und als Gesprächsgrundlage im Team.*

**Wie man diese Seite benutzt**

Nicht von oben nach unten lesen. Nimm dir eine Spalte vor und frage für jede Zeile: Trifft das bei uns zu? Drei ehrliche Treffer sind ein besseres Ergebnis als vierzig durchgelesene Zeilen.
Als Teamübung: Jeder markiert für sich die fünf zutreffendsten Punkte. Anschließend vergleichen. Die Punkte, die alle markiert haben, sind der Arbeitsvorrat. Die, bei denen die Einschätzungen auseinandergehen, sind das interessantere Gespräch.

## 19.1 Organisation und Kultur

| Anti-Pattern | Woran man es erkennt | Kapitel |
|---|---|---|
| Das DevOps-Team | Neue Abteilung, alte Übergaben — drei Silos statt zwei | 01 |
| Werkzeug zuerst | „Wir führen X ein, dann wird alles gut" | 01 |
| Lokale Optimierung | Ein Team wird schneller, die Durchlaufzeit bleibt gleich | 01 |
| Kultur per Dekret | Werteplakate im Flur, unveränderte Zielvorgaben | 02 |
| Automatisierung ohne Kultur | Pipeline da, aber niemand traut sich auszuliefern | 02 |
| Architektur ohne Organisation | Microservices im Entwurf, zentrale Freigabe im Alltag | 02 |
| Kennzahlen als Druckmittel | Teams werden anhand ihrer Zahlen verglichen | 15 |
| Verbesserung ohne Zeitbudget | „wenn mal Luft ist"— also nie | 16 |

## 19.2 Code und Versionsverwaltung

| Anti-Pattern | Woran man es erkennt | Kapitel |
|---|---|---|
| Langlebige Feature-Branches | Zweige leben Wochen, Zusammenführung schmerzt | 03 |
| Sammelcommits | „Änderungen an 47 Dateien" | 03 |
| Direkt auf `main` pushen | Kein Branch-Schutz aktiv | 03 |
| Geheimnisse im Repository | Passwörter in der Historie | 03 |
| Build-Ergebnisse eingecheckt | Repository wird groß, ständige Konflikte | 03 |
| Riesige Pull Requests | 2.000 Zeilen, Prüfung wird zur Formalie | 03 |
| `-force` auf gemeinsame Zweige | Fremde Arbeit verschwindet | 03 |

## 19.3 Testen

| Anti-Pattern | Woran man es erkennt | Kapitel |
|---|---|---|
| Eistüte statt Pyramide | Viele E2E-Tests, kaum Unit-Tests, 20 Minuten Laufzeit | 04 |
| Abdeckung als Zielvorgabe | Tests ohne Zusicherungen, nur zur Quotenerfüllung | 04 |
| Produktivdaten im Test | Datenbankabzug auf dem Testsystem | 04 |
| Sporadische Fehlschläge geduldet | „Einfach nochmal laufen lassen" | 04 |
| Manuelle Regressionstests | Eine Liste, die jemand durchklickt | 04 |
| Nur der Gutfall wird getestet | Randfälle brechen erst in Produktion | 04 |
| Tests erst am Projektende | Befunde, wenn ihre Behebung am teuersten ist | 04 |

## 19.4 Pipeline und Auslieferung

| Anti-Pattern | Woran man es erkennt | Kapitel |
|---|---|---|
| Nächtlicher Build | Rückmeldung nach 14 Stunden | 05 |
| Roter `main` wird toleriert | Niemand weiß, was funktioniert | 05 |
| Pipeline dauert 45 Minuten | Niemand wartet die Rückmeldung ab | 05 |
| Pipeline zusammengeklickt | Nicht versioniert, nicht reproduzierbar | 07 |
| Pro Umgebung neu bauen | Getestet wurde nie das Produktivartefakt | 06 |
| Konfiguration im Artefakt | Ein Image je Umgebung | 06 |
| Migration ohne Rückwärtskompatibilität | Rollback unmöglich | 06 |
| Kein Rauchtest nach dem Ausrollen | Störung fällt Anwendern zuerst auf | 06 |
| `latest` als Tag | Unklar, was läuft | 07 |
| Geheimnisse im Protokoll | `set -x` in Skripten mit Zugangsdaten | 07 |
| Fremde Actions auf beweglichen Tags | Lieferkettenrisiko | 07 |
| Dauerhafte Runner mit Rückständen | „Läuft mal, läuft mal nicht" | 07 |
| Scan ohne Konsequenz | Bericht wird erzeugt und abgelegt | 07 |
| Ausrollen ohne Rückweg | Rollback wurde nie geübt | 08 |
| Canary ohne Messung | Man merkt nichts und schaltet trotzdem weiter | 08 |
| Feature Flags ohne Ablaufdatum | Codepfad-Wildwuchs nach zwei Jahren | 08 |
| Rollback erst nach der Ursachenanalyse | Ausfallzeit verlängert sich | 08 |

## 19.5 Infrastruktur und Betrieb

| Anti-Pattern | Woran man es erkennt | Kapitel |
|---|---|---|
| Shell-Skripte statt deklarativer Beschreibung | Zweiter Durchlauf richtet Schaden an | 09 |
| Nach dem Automatismus von Hand nachbessern | Drift kehrt sofort zurück | 09 |
| Einmal ausgeführt und nie wieder | Drift wächst unbemerkt | 09 |
| Schneeflockenumgebungen | „Auf der Abnahme läuft es aber" | 06 |
| Terraform-Zustandsdatei im Git | Geheimnisse öffentlich | 09 |
| Mehrere Dienste in einem Container | Eine VM im Container-Gewand | 10 |
| SSH-Server im Container | Änderungen, die beim Neustart verschwinden | 10 |
| Container als Root | Läuft nicht unter OpenShift, unnötige Angriffsfläche | 10 |
| `SIGTERM` ignoriert | Fehler bei jedem Rolling Update | 10 |
| Kubernetes für drei Container | Aufwand ohne Gegenwert | 10 |
| Änderungen von Hand am Zielsystem | Werden beim nächsten Abgleich überschrieben | 11 |
| Anwendungs- und Konfigurationsrepo vermischt | Keine saubere Freigabe möglich | 11 |

## 19.6 Überwachung und Störungen

| Anti-Pattern | Woran man es erkennt | Kapitel |
|---|---|---|
| Alarm auf Ursachen statt Symptome | „CPU über 80 %"weckt jemanden | 12 |
| Durchschnittslatenz | Ausreißer verschwinden im Mittelwert | 12 |
| Alarme ohne Handlungsanweisung | Ratlosigkeit um drei Uhr nachts | 12 |
| Zu viele Alarme | Echte Meldungen gehen unter | 12 |
| Personenbezogene Daten in Logs | Vertragsdaten in der zentralen Sammlung | 12 |
| `/health` prüft auch die Datenbank | Neustartkaskade bei DB-Störung | 12 |
| Erst überwachen, wenn es brennt | Störung ohne Datenlage | 12 |
| Ursachensuche vor Wiederherstellung | Ausfallzeit verlängert sich | 13 |
| Incident Commander tippt mit | Niemand koordiniert | 13 |
| Postmortem endet beim Menschen | Dieselbe Störung kommt wieder | 13 |
| Maßnahmen ohne Termin und Verantwortliche | Werden nie umgesetzt | 13 |
| Runbooks veralten | Bereitschaft steht ohne Anleitung da | 13 |
| Bereitschaft ohne Ausgleich | Erschöpfung, Kündigungen | 13 |

## 19.7 Sicherheit und Regulierung

| Anti-Pattern | Woran man es erkennt | Kapitel |
|---|---|---|
| Sicherheitsprüfung erst am Ende | Ausnahmegenehmigungen statt behobener Befunde | 14 |
| Scanner ohne Bearbeitungsverfahren | 1.400 Befunde, die niemand ansieht | 14 |
| Unbefristete Ausnahmen | „Temporärßeit drei Jahren | 14 |
| Automatische Updates ohne Tests | Fremder Code ungeprüft in Produktion | 14 |
| Kein SBOM | „Sind wir betroffen?"bleibt tagelang offen | 14 |
| Sicherheit mit Vetorecht ohne Beteiligung | Umgehung, Schattenprozesse | 14 |
| „Regulierung verbietet Automatisierung" | Stillstand mit falscher Begründung | 17 |
| Freigabe als Formalie | Vier-Augen-Prinzip nur auf dem Papier | 17 |
| Notfalländerung als Regelweg | Ein Drittel aller Änderungen ist „dringend" | 17 |
| Nachweise nachträglich erzeugen | Lückenhaft und angreifbar | 17 |
| Personenkonten mit Produktionszugriff | Funktionstrennung nicht belegbar | 17 |
| Papierprozess parallel zur Automatik | Doppelter Aufwand, widersprüchliche Stände | 17 |

## 19.8 Die sieben schwersten

Wenn nur sieben Punkte bearbeitet werden können, dann diese — sortiert nach Wirkung:

1. **Kein Rollback geübt.** Ohne erprobten Rückweg ist jede Auslieferung ein Wagnis, und alles Weitere baut darauf auf.
2. **Roter Hauptzweig toleriert.** Damit ist Continuous Integration wirkungslos, egal welche Werkzeuge installiert sind.
3. **Produktivdaten im Testsystem.** Ein Datenschutzverstoß mit Meldepflicht, unabhängig von allem anderen.
4. **Postmortem endet beim menschlichen Fehler.** Garantiert die Wiederholung derselben Störung.
5. **Schuldsuche nach Störungen.** Zerstört die Meldebereitschaft und damit die Informationsgrundlage.
6. **Langlebige Feature-Branches.** Verhindert alles, was in Teil C beschrieben wird.

7. **Kennzahlen zum Teamvergleich.** Verdirbt die Zahlen und das Vertrauen gleichzeitig.

## 19.9 Reflexionsfrage

*Welches Anti-Pattern auf dieser Seite kennst du aus eigener Erfahrung so gut, dass du es jemandem erklären könntest — und was war damals der Grund, warum es so gemacht wurde?*

Der zweite Teil ist der wichtigere. Anti-Pattern entstehen fast nie aus Nachlässigkeit, sondern aus einer Entscheidung, die damals vernünftig war. Wer den ursprünglichen Grund kennt, argumentiert beim Ändern deutlich überzeugender.

# Kapitel 20

# Glossar und Literatur

## 20.1 Glossar

| Begriff | Bedeutung |
|---|---|
| **A/B-Test** | Zwei Varianten parallel im Betrieb, Entscheidung anhand von Daten |
| **Artefakt** | Ergebnis des Bauvorgangs: Image, Paket, Binärdatei |
| **Blameless Postmortem** | Aufarbeitung einer Störung ohne Schuldzuweisung |
| **Blue/Green** | Zwei vollständige Umgebungen, Umschaltung per Lastverteiler |
| **Branch** | Entwicklungszweig in der Versionsverwaltung |
| **CAB** | Change Advisory Board, Änderungsgremium nach ITIL |
| **CALMS** | Culture, Automation, Lean, Measurement, Sharing |
| **Canary** | Neue Version zunächst für einen kleinen Verkehrsanteil |
| **Change Failure Rate** | Anteil der Auslieferungen, die eine Störung verursachen |
| **CI / CD / CD** | Continuous Integration, Delivery, Deployment |
| **Commit** | Festgeschriebene Änderung in der Versionsverwaltung |
| **Container** | Isolierter Prozess mit eigenem Dateisystem, teilt den Kernel |
| **CVE** | Eindeutige Kennung einer bekannten Schwachstelle |
| **DAST** | Dynamische Sicherheitsprüfung der laufenden Anwendung |
| **Deployment** | Neue Version läuft auf den Servern |
| **Digest** | Kryptografische Prüfsumme, identifiziert ein Image eindeutig |
| **DIKW** | Data, Information, Knowledge, Wisdom |
| **DMAIC** | Define, Measure, Analyse, Improve, Control — Six-Sigma-Zyklus |
| **DORA (Forschung)** | DevOps Research and Assessment, Quelle der vier Kennzahlen |
| **DORA (Verordnung)** | Digital Operational Resilience Act, EU 2022/2554 |
| **Drift** | Abweichung zwischen beschriebenem Soll und tatsächlichem Ist |
| **Error Budget** | Erlaubter Ausfallanteil, der aus dem SLO folgt |
| **Expand-Contract** | Schemaänderung in rückwärtskompatiblen Schritten |
| **Feature Flag** | Schalter, der eine Funktion zur Laufzeit aktiviert |
| **Flow-Effizienz** | Bearbeitungszeit geteilt durch Durchlaufzeit |
| **GitOps** | Git als Quelle der Wahrheit, Zielsystem gleicht sich laufend ab |
| **Idempotenz** | Wiederholte Ausführung ändert nichts |

## 20.2 Bücher

| Titel | Autoren | Wofür |
|---|---|---|
| **The Phoenix Project** | Gene Kim, Kevin Behr, George Spafford | Roman. Der beste Einstieg für alle, die noch nicht überzeugt sind — man erkennt den eigenen Betrieb auf jeder zweiten Seite wieder. |
| **The DevOps Handbook** | Gene Kim, Jez Humble, Patrick Debois, John Willis | Das Standardwerk. Die Drei Wege ausführlich, mit vielen Fallbeispielen. |
| **Accelerate** | Nicole Forsgren, Jez Humble, Gene Kim | Die Forschungsgrundlage hinter den DORA-Kennzahlen. Dünn, dicht, überzeugend. |
| **Continuous Delivery** | Jez Humble, David Farley | Der Klassiker zur Auslieferungskette. Technisch, ausführlich, zeitlos. |
| **Site Reliability Engineering** | Betsy Beyer u. a. (Google) | SLO, Fehlerbudget, Bereitschaft, Postmortems. Frei online lesbar. |
| **Team Topologies** | Matthew Skelton, Manuel Pais | Wie man Teams schneidet. Die praktische Anwendung von Conways Gesetz. |
| **The Unicorn Project** | Gene Kim | Fortsetzung des Phoenix Project aus Entwicklersicht. |

## 20.3 Quellen im Netz

| Quelle | Inhalt |
|---|---|
| `dora.dev` | Aktuelle State-of-DevOps-Berichte und Kennzahlendefinitionen |
| `12factor.net` | Die Zwölf-Faktoren-Methodik |
| `sre.google/books` | Die SRE-Bücher von Google, kostenfrei |
| `trunkbaseddevelopment.com` | Ausführlich zu Branching-Strategien |
| `owasp.org` | Sicherheit: Top Ten, Cheat Sheets, Werkzeuge |
| `martinfowler.com` | Grundlagenartikel zu CI, Testpyramide, Feature Flags |
| `bafin.de` | Aufsichtsrechtliche Vorgaben, DORA-Umsetzung |
| `postmortems.pagerduty.com` | Praxisleitfaden für Störungsaufarbeitung |

## 20.4 Werkzeuge nach Einsatzzweck

| Zweck | Auswahl |
|---|---|
| Versionsverwaltung | Git; Server: GitLab, Gitea, Forgejo, GitHub |
| CI/CD | GitLab CI, GitHub Actions, Jenkins, Woodpecker, Tekton |
| Tests | pytest, JUnit, Playwright, k6 |
| Linting | ruff, eslint, shellcheck, ansible-lint, yamllint |
| Container | Podman, Docker, Buildah, Skopeo |
| Orchestrierung | Quadlet/systemd, k3s, Kubernetes, OpenShift |
| Infrastructure as Code | Ansible, Terraform, OpenTofu, Pulumi |
| GitOps | Argo CD, Flux, systemd-Timer |
| Registry | Quay, Harbor, Nexus, Artifactory |
| Observability | Prometheus, Grafana, Loki, Tempo, OpenTelemetry |
| Alarmierung | Alertmanager, Grafana Alerting |
| Sicherheit | Trivy, Grype, Syft, Bandit, Semgrep, gitleaks, checkov, cosign |
| Geheimnisse | HashiCorp Vault, OpenBao, SOPS, Podman Secrets |
| Störungen | PagerDuty, Opsgenie, GLPI, OTOBO |

Die Liste ist eine Auswahl, keine Empfehlung. Für den Einstieg gilt: **Nimm, was im Haus schon vorhanden ist.** Ein mittelmäßiges Werkzeug, das eingeführt und beherrscht wird, schlägt ein hervorragendes, das nach dem Pilotprojekt liegen bleibt.

## 20.5 Zertifizierungen

| Zertifizierung | Anbieter | Passt zu |
|---|---|---|
| **EX188** Containers | Red Hat | Podman, Container-Grundlagen |
| **EX280** OpenShift Administration | Red Hat | Betrieb von OpenShift |
| **EX294** RHCE (Ansible) | Red Hat | Automatisierung unter RHEL |
| **CKA / CKAD** | CNCF | Kubernetes-Betrieb bzw. -Entwicklung |
| **ITIL 4 Foundation** | PeopleCert | Betriebsprozesse, Schnittstelle zu DevOps |
| **ISTQB Foundation** | ISTQB | Testgrundlagen |

## 20.6 Wie es weitergeht

Wer diese Unterlage durchgearbeitet hat, kennt die Bausteine. Der nächste Schritt ist nicht das nächste Buch, sondern die Anwendung — und zwar in dieser Reihenfolge:

1. **Messen.** Die vier Kennzahlen aus Kapitel 15 für einen echten Ablauf erheben. Ohne Ausgangswert ist jede Verbesserung eine Behauptung.
2. **Den Engpass finden.** Eine Wertstromanalyse nach Kapitel 16, gemeinsam mit den Beteiligten.
3. **Eine Sache ändern.** Genau eine, mit Termin und Verantwortlichem.
4. **Nachmessen.** Nach vier bis sechs Wochen.
5. **Wiederholen.**

**Zum Schluss noch dies.**

Der Reiseführer aus dem Anhalter durch die Galaxis ist bekanntlich deshalb so erfolgreich, weil er zwar an vielen Stellen ungenau ist, dafür aber zwei beruhigende Worte auf dem Umschlag trägt und deutlich preiswerter als die Konkurrenz.
Für diese Unterlage gilt Ähnliches: Sie ist an einigen Stellen vereinfacht, sie wird in zwei Jahren an mehreren Stellen überholt sein, und sie ersetzt keine Erfahrung. Was sie leisten kann, ist die Panik zu nehmen — vor der Begriffsflut, vor dem ersten roten Build, vor der ersten Störung um drei Uhr nachts.
Der Rest kommt durch Machen. Und ein Handtuch schadet bekanntlich auch nie.