

SE · ASSOCIATE · APPLY

AI-Assisted Software Engineer

Teilnehmerhandbuch mit Beispielen und Übungen

Mit Modellen entwickeln, ohne die Verantwortung abzugeben.

Foundation ✓

Associate

Professional

Expert

Umfang 7 Module · ca. 20 Stunden

Prüfung 25 Fragen · 50 Minuten · Bestehensgrenze 75 %

Voraussetzung AI Foundations Professional (AF) oder anerkannte Gleichwertigkeit

Rechtsstand August 2026

CertifAI · Version 1.0 · Ausgabe DE

Dieses Handbuch bereitet auf die Zertifizierungsprüfung AI-Assisted Software Engineer vor. Die Teilnahme an diesem Kurs ist keine Voraussetzung für die Prüfung; gleichwertige Vorbereitungswege werden anerkannt.

Vor dem Start

Einordnung, Voraussetzungen, Arbeitsweise

Wofür dieser Track steht

Gegenstand ist das **Entwickeln mit Modellen** — Codegenerierung im Alltag, Review des Erzeugten, Lizenz- und Sicherheitsfragen, Tests als Sicherheitsnetz. Es geht nicht darum, KI zu bauen, sondern darum, mit KI Software zu bauen und dabei Autor zu bleiben.

Nicht Gegenstand sind Modelle selbst — Architektur, Feinabstimmung, Inferenz gehören zu *AI Core & ML Developer* (ML). Angriffssicherheit als eigene Disziplin gehört zu *AI Security & Red Team Specialist* (SR); Modul 3 behandelt davon nur, was in generiertem Code regelmäßig auftaucht und wie du es im Review erkennst.

Was du mitbringen solltest

Praktische Programmiererfahrung in mindestens einer Sprache. Die Beispiele sind in TypeScript und Python gehalten, aber die Fehlerklassen sind sprachunabhängig — wer Java, C# oder Go schreibt, erkennt sie wieder. Aus *AI Foundations* werden Halluzination, Kontextfenster und die Grenzen der Prüfbarkeit vorausgesetzt.

Für die Übungen brauchst du einen Codeassistenten, mit dem du tatsächlich arbeitest. Dieser Track lässt sich nicht durch Lesen lernen.

DER SATZ, UM DEN ES IN DIESEM TRACK GEHT

Wer den Code merged, hat ihn geschrieben. Nicht das Modell. Die Signatur unter dem Commit ist deine, die Verantwortung im Vorfall ist deine, und „das kam vom Assistenten“ ist im Postmortem keine Aussage, sondern eine Auskunft über deinen Prozess.

Hinweise zur Qualifikation

CertifAI ist ein unabhängiger Anbieter. Dieses Zertifikat ist ein Industriezertifikat und keine staatlich anerkannte Qualifikation. Es wird von CertifAI ausgestellt und ist nicht behördlich zugelassen.

Teile dieses Handbuchs wurden KI-gestützt erstellt und anschließend fachlich geprüft und freigegeben. Die Prüfungsfragen werden unabhängig vom Kursmaterial entwickelt. Die Bewertung erfolgt deterministisch gegen einen von Fachexperten freigegebenen Antwortschlüssel; KI kommt dabei nicht zum Einsatz.

Die Codebeispiele mit Schwachstellen dienen ausschließlich dem Erkennen im Review. Sie sind bewusst so gehalten, dass sie Fehlerklassen zeigen, ohne einsatzfähige Angriffe zu enthalten. Lizenz- und urheberrechtliche Hinweise in Modul 4 sind eine Einordnung, keine Rechtsberatung.

Inhalt

Modul 1 Mit Modellen entwickeln	4
Modul 2 Review von generiertem Code	8
Modul 3 Sicherheitsschwächen in Vorschlägen	14
Modul 4 Lizenz und Herkunft	19
Modul 5 Geheimnisse und Datenflüsse	23
Modul 6 Tests als Sicherheitsnetz	27
Modul 7 Agentische Arbeitsweisen	31
Musterprüfung 25 Fragen	35
Lösungen	39
Glossar	47
Prüfungsinformationen	48

Mit Modellen entwickeln

ca. 2,5 Stunden • Übung und Selbsttest am Ende

LERNZIELE

- Du wählst den passenden Arbeitsmodus für die jeweilige Aufgabe.
- Du gibst Kontext gezielt statt vollständig.
- Du schreibst eine Implementierungsspezifikation, bevor du generieren lässt.
- Du erkennst die Aufgaben, bei denen ein Assistent zuverlässig mehr kostet als er bringt.

1.1 Drei Arbeitsmodi

MODUS	WOFÜR	REVIEWAUFWAND
Vervollständigung im Editor, zeilenweise	Wiederkehrende Muster, Signaturen, Boilerplate	Niedrig — du liest im Schreiben mit
Dialog Chat neben dem Editor	Eine Funktion, ein Refactoring, eine Erklärung	Mittel — abgegrenzter Diff
Agent mehrere Dateien, eigene Schritte	Umbauten über Dateigrenzen, Migrationen	Hoch — siehe Modul 7

Die letzte Spalte ist die, nach der man auswählen sollte. Der Gewinn eines Modus liegt in der gesparten Schreibzeit; die Kosten liegen in der Lesezeit. Bei der Vervollständigung ist das Verhältnis günstig, weil du ohnehin im Code stehst. Beim Agenten kippt es, sobald der Diff größer wird, als du in einem Zug prüfen kannst.

1.2 Kontext gezielt geben

Der verbreitete Reflex ist, dem Assistenten möglichst viel zu zeigen — das ganze Projekt, alle offenen Dateien. Das verschlechtert die Ergebnisse, weil die relevante Stelle mit Ablenkung konkurriert, und es erhöht die Menge an Code, die dein Arbeitgeber an einen Dienst überträgt (Modul 5).

Was tatsächlich hilft, ist eng umrissen:

- **Die Schnittstellen**, die der neue Code benutzt — Typdefinitionen, Signaturen, nicht die Implementierungen.
- **Ein bestehendes Beispiel** derselben Art aus eurem Projekt. Das überträgt eure Konventionen zuverlässiger als jede Beschreibung.

- **Die Fehlerbehandlung**, die bei euch üblich ist — sonst erfindet der Assistent eine eigene.
- **Version und Abhängigkeiten**, wenn die Bibliothek sich zwischen Hauptversionen deutlich geändert hat. Ohne Angabe schreibt das Modell gegen den Stand, der in seinen Trainingsdaten überwog — oft eine ältere Hauptversion.

1.3 Spezifikation vor Code

Das ist derselbe Gedanke wie in *AI Prompt Engineer Specialist*, angewandt auf Code: Ein Auftrag an einen Assistenten ist eine Spezifikation, kein Wunsch. Fünf Punkte, drei Minuten:

```
# Aufgabe
Funktion, die offene Rechnungen eines Kunden für einen Zeitraum liefert.

# Signatur
getOpenInvoices(customerId: string, from: Date, to: Date): Promise<Invoice[]>

# Randbedingungen
- Nur Rechnungen mit status = 'open'
- Zeitraum inklusive beider Grenzen
- Bei mehr als 500 Treffern: Fehler werfen, nicht kürzen
- Bestehende Repository-Klasse verwenden, kein direktes SQL
- Fehlerbehandlung wie in getPayments() – Result-Typ, keine Exceptions

# Nicht tun
- Keine neuen Abhängigkeiten
- Kein Caching (kommt später separat)

# Fertig, wenn
- Die drei Testfälle in invoices.spec.ts laufen
- Leerer Zeitraum liefert [] statt Fehler
```

Der Abschnitt „Nicht tun“ ist der, den man weglässt und dann nachträglich aus dem Diff entfernt. Modelle fügen gern Hilfsbibliotheken, Caches und Protokollierung hinzu, weil das in ihren Trainingsdaten oft dabeistand.

DIE AUFGABEN, BEI DENEN EIN ASSISTENT MEHR KOSTET

- **Fehlersuche in unbekanntem Fremdcode.** Das Modell erzeugt plausible Erklärungen für Verhalten, das es nicht beobachten kann. Du prüfst dann Hypothesen statt Fakten.
- **Entscheidungen über Architektur.** Du bekommst die verbreitetste Lösung, nicht die für euren Kontext passende — und die Begründung klingt in beiden Fällen gleich gut.
- **Sehr kleine Änderungen.** Drei Zeilen selbst schreiben ist schneller als drei Zeilen spezifizieren, generieren und prüfen.
- **Alles, was du nicht beurteilen kannst.** Wenn du den Vorschlag nur annehmen oder ablehnen, aber nicht bewerten kannst, hast du keinen Assistenten, sondern ein Risiko mit Autovervollständigung.

BEISPIEL 1.1 — DERSELBE AUFTRAG, ZWEI ERGEBNISSE

WUNSCH

Schreib mir eine Funktion, die offene Rechnungen für einen Kunden holt.

SPEZIFIKATION

siehe 1.3 — Signatur, Randbedingungen, Nicht-Tun, Abnahmekriterium

Ergebnis links: Eine Funktion mit direktem SQL statt der Repository-Klasse, einer neu eingeführten Datumsbibliothek, ``console.log``-Aufrufen und einer stillschweigenden Begrenzung auf 100 Ergebnisse. Alles davon ist plausibel, nichts davon war gewünscht. Der Review dauert länger als das Selbstschreiben.

Ergebnis rechts: Ein Diff, der genau die spezifizierten Punkte trifft und in zwei Minuten zu prüfen ist. Die drei Minuten Spezifikation sind der Grund dafür.

ÜBUNG 1 — VOM TICKET ZUR SPEZIFIKATION

Aus dem Backlog: „Als Nutzer möchte ich meine Exportdateien auch nach 30 Tagen noch herunterladen können.“

1. Notiere fünf Fragen, die du beantwortet haben musst, bevor du generieren lässt.
2. Schreibe die vollständige Spezifikation nach dem Muster aus 1.3 — mit Signatur, Randbedingungen, „Nicht tun“ und Abnahmekriterium.
3. Welche Teile dieser Aufgabe würdest du *nicht* generieren lassen? Begründe mit dem Kasten oben.
4. Welche vier Kontextstücke gibst du dem Assistenten mit — und welche bewusst nicht?

KERNAUSSAGEN

- Wähle den Modus nach dem Reviewaufwand, nicht nach der Schreibersparnis.
- Kontext gezielt: Schnittstellen, ein Beispiel aus dem Projekt, eure Fehlerbehandlung, Versionen.
- Spezifikation vor Code — einschließlich „Nicht tun“.
- Was du nicht beurteilen kannst, lässt du nicht generieren.

Selbsttest Modul 1

1. Wonach wählt man den Arbeitsmodus sinnvollerweise aus?

- a) Nach der gesparten Schreibzeit.
- b) Nach dem entstehenden Reviewaufwand.
- c) Nach der Größe des Projekts.
- d) Nach der verwendeten Sprache.

2. Warum gehört „Nicht tun“ in die Spezifikation?

- a) Um den Prompt zu verlängern.
- b) Weil Modelle sonst Abhängigkeiten, Caches und Protokollierung ergänzen.
- c) Weil es die Laufzeit verbessert.
- d) Weil Linter es verlangen.

Review von generiertem Code

ca. 3 Stunden · Übung und Selbsttest am Ende

LERNZIELE

- Du kennst die sieben Fehlerklassen, die in generiertem Code überdurchschnittlich häufig sind.
- Du erkennst halluzinierte Bibliotheken und Funktionen und weißt, warum sie ein Sicherheitsproblem sind.
- Du arbeitest mit einer Reviewreihenfolge, die die teuren Fehler zuerst findet.
- Du hältst das Verhältnis von Diffgröße zu Reviewkapazität bewusst.

2.1 Die Tempo-Falle

Code zu erzeugen ist um ein Vielfaches schneller geworden. Code zu *lesen* ist genauso langsam wie vorher. Dieses Missverhältnis ist die zentrale Tatsache dieses Moduls, und aus ihr folgt fast alles Weitere.

In der Praxis äußert es sich als Nachlassen der Prüftiefe: Der erste Vorschlag wird gründlich gelesen, der fünfte überflogen, der zehnte akzeptiert, weil die vorherigen ja gut waren. Das ist keine Charakterschwäche, sondern eine vorhersagbare Ermüdung — und sie tritt bei erfahrenen Entwicklerinnen genauso auf.

Die Gegenmaßnahme ist keine Selbstdisziplin, sondern eine Begrenzung: **Nimm nicht mehr Code an, als du in einem Zug prüfen kannst.** Praktisch heißt das kleine Aufträge, kleine Diffs, häufige Zwischenstände.

2.2 Sieben Fehlerklassen

Generierter Code ist nicht allgemein schlechter, aber er irrt *anders*. Diese sieben Muster sind überdurchschnittlich häufig, weil sie in plausiblen Code besonders plausibel aussehen:

KLASSE	MERKMAL
Falsche API-Nutzung	Signatur oder Verhalten stimmen nicht mit der tatsächlichen Bibliothek überein — häufig eine andere Hauptversion
Halluzinierte Funktionen	Aufrufe, die es so nicht gibt; die Namen klingen richtig
Fehlende Fehlerbehandlung	Der glückliche Pfad ist vollständig, alles andere fehlt oder wird verschluckt
Grenzfälle	Leere Liste, ein Element, <code>null</code> , Zeitzonen, Sonderzeichen
Ressourcen	Verbindungen, Dateien, Sperren werden geöffnet und nicht freigegeben
Nebenläufigkeit	Fehlendes <code>await</code> , Wettlaufsituationen, gemeinsam genutzter Zustand
Leistung im Kleinen	Abfragen in Schleifen, unnötige Kopien, quadratisches Verhalten bei Listen

Die vierte und die siebte Klasse haben eine Gemeinsamkeit: Sie bestehen alle Tests, wenn die Tests aus derselben Quelle stammen — dazu Modul 6.

2.3 Halluzinierte Pakete sind ein Sicherheitsproblem

Wenn ein Modell einen Paketnamen erfindet, ist das zunächst nur ärgerlich: Die Installation schlägt fehl. Gefährlich wird es dadurch, dass Modelle *dieselben* Namen wiederholt erfinden — und ein Angreifer diesen Namen im öffentlichen Paketverzeichnis registrieren kann. Wer dann kommentarlos installiert, holt sich fremden Code ins Projekt und in die Entwicklungsumgebung.

REGEL FÜR JEDE NEUE ABHÄNGIGKEIT

Bevor du ein vorgeschlagenes Paket installierst: Existiert es wirklich? Wer veröffentlicht es? Seit wann, mit wie vielen Versionen, wie vielen Downloads? Passt die Beschreibung zum Zweck? Ist es das, was das Ökosystem tatsächlich verwendet — oder ein Name, der nur so klingt?

Und die vorgelagerte Frage, die häufiger die richtige ist: *Brauchen wir hier überhaupt eine neue Abhängigkeit?* Ein Assistent schlägt Bibliotheken vor, weil in seinen Trainingsdaten Bibliotheken verwendet wurden — nicht, weil euer Projekt eine braucht.

2.4 Reviewreihenfolge

Lies generierten Code nicht von oben nach unten. Prüfe in dieser Reihenfolge, weil sie die teuren Fehler zuerst findet:

- 1. Tut es das Richtige?** Vergleich gegen deine Spezifikation aus Modul 1 — nicht gegen das, was der Code zu tun scheint.
- 2. Grenzen und Fehler.** Was passiert bei leer, `null`, sehr groß, gleichzeitig, Netz weg?

3. **Sicherheit.** Eingaben, Berechtigungen, Geheimnisse — Modul 3.
4. **Fremde Bestandteile.** Neue Abhängigkeiten, kopierte Blöcke — Modul 2.3 und Modul 4.
5. **Passung.** Konventionen, Namen, Struktur eures Projekts.
6. **Stil.** Zuletzt, und am besten durch den Formatierer.

Der häufigste Reviewfehler ist, bei 5 und 6 anzufangen, weil sie am leichtesten zu sehen sind — und nach zwanzig Stilkommentaren zufrieden zu sein, ohne Punkt 1 geprüft zu haben.

BEISPIEL 2.1 — EIN DIFF, SIEBEN BEFUNDE

```
async function exportInvoices(customerId: string, from: Date, to: Date) {
  const conn = await pool.connect()
  const res = await conn.query(
    `SELECT * FROM invoices
     WHERE customer_id = '${customerId}'
     AND created_at BETWEEN '${from}' AND '${to}'`)

  const rows = []
  for (let i = 0; i <= res.rows.length; i++) {
    const inv = res.rows[i]
    const items = await conn.query(
      `SELECT * FROM items WHERE invoice_id = ${inv.id}`)
    rows.push({ ...inv, items: items.rows, total: inv.total.toFixed(2) })
  }

  try { await uploadExport(`exports/${customerId}.csv`, toCsv(rows)) }
  catch (e) {}

  return rows
}
```

Der Code sieht ordentlich aus, ist benannt wie üblich und wird von einem Test grün gemeldet, der eine Rechnung mit einer Position prüft. Die Befunde, in der Reihenfolge aus 2.4:

1 — Berechtigung fehlt. Nirgends wird geprüft, ob der Aufrufende diesen `customerId` sehen darf. Das ist der teuerste Befund und der, den kein Linter findet.

2 — SQL zusammengesetzt statt parametrisiert. `customerId` geht direkt in die Abfrage. Auch wenn heute nur interner Code aufruft: Der nächste Aufrufer ist ein Endpunkt.

3 — Schleifengrenze. `i <= length` läuft einen Durchgang zu weit; `inv` ist dann undefined und der Zugriff bricht ab. Ein Test mit genau einer Zeile fällt darüber, ein Test mit null Zeilen nicht.

4 — Abfrage in der Schleife. Bei 400 Rechnungen sind es 401 Abfragen. Fällt in der Entwicklung mit drei Datensätzen nie auf.

5 — Verbindung nicht freigegeben. Kein `finally`, kein `release()`. Nach einigen hundert Aufrufen ist der Verbindungspool leer, und der Ausfall trifft die ganze Anwendung, nicht nur diese Funktion.

6 — Fehler verschluckt. Der leere `catch` lässt den Upload scheitern und die Funktion Erfolg melden. Das ist die unangenehmste Sorte Fehler: Sie erzeugt keine Meldung, sondern fehlende Daten.

7 — Typannahme. `inv.total.toFixed(2)` setzt eine Zahl voraus; viele Treiber liefern Dezimalspalten als Zeichenkette. Dann bricht es ab — oder, schlimmer, es funktioniert in der Entwicklung und nicht in der Produktion, weil dort ein anderer Treiber läuft.

Nebenbei: `exports/${customerId}.csv` baut einen Pfad aus einer Eingabe. Selbst wenn heute nur Zahlen kommen — der Dateiname ist Nutzerdaten, und der gehört bereinigt.

ÜBUNG 2 — REVIEW

Der Assistent liefert diese Funktion zum Zusammenführen von Nutzerprofilen:

```
def merge_profiles(primary_id, duplicate_ids):
    primary = db.get_user(primary_id)
    for dup_id in duplicate_ids:
        dup = db.get_user(dup_id)
        for key in dup.keys():
            if not primary.get(key):
                primary[key] = dup[key]
        db.delete_user(dup_id)
        log.info(f"merged {dup_id} into {primary_id}: {dup}")
    db.save_user(primary)
    return primary
```

1. Finde mindestens sechs Befunde. Ordne jeden einer der sieben Klassen aus 2.2 zu.
2. Zwei der Befunde sind so schwer, dass du den Diff ablehnen würdest, statt Änderungen anzumerken. Welche, und warum?
3. Ein Befund betrifft nicht den Code, sondern das Protokoll. Welcher, und was ist daran falsch?
4. Was hätte in der Spezifikation aus Modul 1 stehen müssen, damit mindestens drei dieser Befunde gar nicht erst entstehen?
5. Schreibe den einen Testfall, der den gefährlichsten Befund zuverlässig aufdeckt.

KERNAUSSAGEN

- Erzeugen ist schneller geworden, Lesen nicht. Begrenze die Diffgröße, nicht die Sorgfalt.
- Sieben Fehlerklassen, die in plausiblen Code besonders plausibel aussehen.
- Halluzinierte Paketnamen sind ein Lieferkettenrisiko — jede neue Abhängigkeit wird geprüft.
- Reviewreihenfolge: Richtigkeit, Grenzen, Sicherheit, Fremdes, Passung, Stil.
- Ein grüner Test beweist nichts, wenn er dieselben Annahmen teilt wie der Code.

Selbsttest Modul 2

1. Warum ist ein halluzinierter Paketname mehr als ein Ärgernis?
 - a) Er verlängert die Installation.
 - b) Modelle erfinden wiederholt denselben Namen, den ein Angreifer registrieren kann.
 - c) Er verstößt gegen Lizenzbedingungen.
 - d) Er erhöht die Paketgröße.

2. Womit beginnt ein Review von generiertem Code?

- a) Mit den Namenskonventionen.
- b) Mit dem Abgleich gegen die Spezifikation.
- c) Mit der Formatierung.
- d) Mit den Kommentaren.

Sicherheitsschwächen in Vorschlägen

ca. 3 Stunden • Übung und Selbsttest am Ende

ZUM ZUSCHNITT

Behandelt wird das Erkennen verbreiteter Schwachstellenmuster im Review. Nicht behandelt werden Angriffsdurchführung, Ausnutzung und Sicherheitsarchitektur — das ist *AI Security & Red Team Specialist* (SR). Die Beispiele zeigen unsichere Muster und ihre Korrektur, keine einsatzfähigen Angriffe.

LERNZIELE

- Du kannst erklären, warum generierter Code systematisch unsichere Muster enthält.
- Du erkennst sechs verbreitete Schwachstellenmuster im Diff.
- Du unterscheidest Authentifizierung von Autorisierung — der häufigste fehlende Prüfungsschritt.
- Du weißt, welche Sicherheitsfragen ein Werkzeug erledigen kann und welche nicht.

3.1 Warum das systematisch passiert

Modelle haben aus öffentlich verfügbarem Code gelernt. Der größte Teil davon ist Beispielcode: Tutorials, Antworten in Foren, Demoprojekte. Beispielcode ist auf *Kürze und Verständlichkeit* optimiert — genau deshalb lässt er Validierung, Berechtigungsprüfung und Fehlerbehandlung weg. Das ist im Tutorial richtig und in eurer Anwendung eine Lücke.

Dazu kommt: Sicherer Code sieht selten spektakulär anders aus. Eine fehlende Berechtigungsprüfung ist eine *nicht vorhandene* Zeile — und nichts ist im Review schwerer zu sehen als etwas, das fehlt.

3.2 Sechs Muster

1 — Fehlende Autorisierung

Der häufigste und teuerste Befund. Der Code prüft, ob jemand *angemeldet* ist, aber nicht, ob er *diese Ressource* sehen darf.

NUR AUTHENTIFIZIERUNG

```
app.get('/orders/:id', auth,
  async (req, res) => {
    const o = await db.order(req.params.id)
    res.json(o)
  })
```

MIT AUTORISIERUNG

```
app.get('/orders/:id', auth,
  async (req, res) => {
    const o = await db.order(req.params.id)
    if (!o) return res.sendStatus(404)
    if (o.userId !== req.user.id
      && !req.user.isAdmin)
      return res.sendStatus(404)
    res.json(o)
  })
```

Beachte 404 statt 403 im zweiten Fall: Ein 403 verrät, dass die Bestellung existiert. Und beachte, dass die Prüfung nach dem Laden steht — die Ressource muss bekannt sein, bevor man über sie entscheiden kann.

2 — Eingaben in Abfragen und Befehle

Zusammengesetzte SQL-Abfragen, Shell-Aufrufe mit Nutzerdaten, aus Eingaben gebaute Dateipfade. Die Korrektur ist immer dieselbe Form: **Struktur und Daten trennen** — Parameter statt Verkettung, Argumentlisten statt Kommandozeilen, Zuordnung auf erlaubte Werte statt direkter Pfadverwendung.

3 — Eigene Kryptografie

Selbst gebaute Verschlüsselung, Passwörter mit einfachen Hashverfahren ohne Salt, Zufallszahlen aus dem allgemeinen Zufallsgenerator statt aus der kryptografischen Quelle. Regel: Für Passwörter ein dafür vorgesehenes Verfahren aus der Standardbibliothek oder einer etablierten Bibliothek, für Zufall die kryptografische Variante, und nie ein eigenes Verfahren.

4 — Zu freundliche Voreinstellungen

CORS für alle Ursprünge geöffnet, Fehlerausgaben mit Stapelverfolgung im Betrieb, Debugmodus aktiv, Zertifikatsprüfung abgeschaltet, weil es in der Entwicklung nicht lief. Solche Zeilen entstehen als Zwischenlösung und werden nie zurückgenommen.

5 — Fehlende Ratenbegrenzung und Mengengrenzen

Ein Anmeldeendpunkt ohne Begrenzung, ein Upload ohne Größenbeschränkung, eine Abfrage ohne Obergrenze der Ergebnismenge. Generierter Code enthält solche Grenzen fast nie, weil sie im Beispiel stören.

6 — Vertrauen in Daten von außen

Antworten fremder Dienste, hochgeladene Dateien, Werte aus Kopfzeilen werden ohne Prüfung weiterverwendet. Der Grundsatz aus dem Prompt-Track gilt hier genauso: *Was nicht aus deinem Code stammt, ist Daten und wird validiert.*

3.3 Was Werkzeuge erledigen und was nicht

WERKZEUG FINDET

Bekannte unsichere Aufrufe, verkettete Abfragen, veraltete Abhängigkeiten, hartkodierte Geheimnisse

WERKZEUG FINDET NICHT

Fehlende Autorisierung, fachlich falsche Berechtigungslogik, zu weite Rechte, fehlende Mengengrenzen

Statische Analyse, Abhängigkeitsprüfung und Geheimnis-Scanner gehören in jede Zuliefererkette und finden die erste Spalte zuverlässig. Die zweite Spalte findet nur ein Mensch, der die fachliche Absicht kennt — und sie enthält die teureren Fehler.

BEISPIEL 3.1 — EIN ENDPUNKT, FÜNF FRAGEN

```
@app.post("/api/documents/{doc_id}/share")
def share(doc_id: str, email: str, user = Depends(current_user)):
    doc = db.get_document(doc_id)
    db.add_share(doc_id, email)
    send_mail(email, f"{user.name} hat ein Dokument geteilt: "
              f"https://app.example.com/d/{doc_id}")
    return {"ok": True}
```

Darf dieser Nutzer dieses Dokument teilen? Wird nicht geprüft. Jeder angemeldete Nutzer kann jedes Dokument teilen, dessen Kennung er kennt. Der schwerste Befund.

Existiert das Dokument? doc wird geladen und nie verwendet — bei einem unbekannten Dokument wird trotzdem eine Freigabe angelegt und eine Mail versendet.

Ist die Adresse gültig und zulässig? Keine Prüfung. Damit ist der Endpunkt außerdem ein Werkzeug, um in fremdem Namen Mails mit einem Link an beliebige Adressen zu senden.

Wie oft darf das passieren? Keine Begrenzung.

Wird es protokolliert? Nein. Eine Freigabe ist eine sicherheitsrelevante Handlung und gehört mit Akteur, Ziel und Zeitpunkt ins Protokoll.

Kein einziger dieser fünf Punkte ist an einer vorhandenen Zeile erkennbar. Alle fünf sind fehlende Zeilen. Deshalb funktioniert Sicherheitsreview nur mit einer Liste von Fragen, nicht durch Lesen.

ÜBUNG 3 — SICHERHEITSREVIEW

Der Assistent liefert einen Endpunkt zum Zurücksetzen des Passworts:

```
@app.post("/api/password-reset")
def reset(email: str):
    user = db.find_user(email)
    if not user:
        return {"error": "Kein Konto mit dieser Adresse"}
    token = str(random.randint(100000, 999999))
    db.save_reset_token(user.id, token)
    send_mail(email, f"Ihr Code: {token}")
    return {"ok": True, "user_id": user.id}

@app.post("/api/password-reset/confirm")
def confirm(user_id: str, token: str, new_password: str):
    saved = db.get_reset_token(user_id)
    if saved == token:
        db.set_password(user_id, hashlib.md5(
            new_password.encode()).hexdigest())
        return {"ok": True}
    return {"error": "Falscher Code"}
```

1. Finde mindestens sieben Befunde und ordne sie den Mustern aus 3.2 zu.
2. Zwei Befunde ermöglichen zusammen die Übernahme eines fremden Kontos. Welche, und in welcher Reihenfolge wirken sie?
3. Ein Befund ist eine Informationspreisgabe, die für sich harmlos wirkt. Welcher, und warum ist er es nicht?
4. Welche zwei der sieben hätte ein statisches Analysewerkzeug gefunden, welche nicht?
5. Schreibe die korrigierte Fassung der ersten Funktion in Stichpunkten — was ändert sich, und warum jeweils?

KERNAUSSAGEN

- Generierter Code erbt die Auslassungen von Beispielcode: Validierung, Berechtigung, Fehlerbehandlung.
- Fehlende Autorisierung ist der häufigste und teuerste Befund — und eine fehlende Zeile.
- Struktur und Daten trennen: Parameter, Argumentlisten, Zuordnung auf erlaubte Werte.
- Werkzeuge finden bekannte Muster; fachliche Berechtigungsfehler findet nur ein Mensch.
- Sicherheitsreview arbeitet mit einer Fragenliste, nicht durch Lesen.

Selbsttest Modul 3

1. Warum enthält generierter Code systematisch weniger Sicherheitsprüfungen?

- a) Weil Modelle Sicherheit ablehnen.
- b) Weil Beispielcode auf Kürze optimiert ist und Prüfungen weglässt.
- c) Weil Sicherheitsprüfungen selten dokumentiert sind.
- d) Weil sie die Laufzeit erhöhen.

2. Welchen Befund findet ein statisches Analysewerkzeug am wenigsten zuverlässig?

- a) Verkettete SQL-Abfragen.
- b) Hartkodierte Zugangsdaten.
- c) Fehlende Autorisierungsprüfung.
- d) Veraltete Abhängigkeiten.

Lizenz und Herkunft

ca. 2,5 Stunden • Übung und Selbsttest am Ende

EINORDNUNG, KEINE RECHTSBERATUNG

Dieses Modul beschreibt Fragen, die sich stellen, und Verfahren, mit denen Unternehmen ihnen begegnen. Die Rechtslage zu generiertem Code ist in Bewegung und unterscheidet sich zwischen Rechtsordnungen. Konkrete Fälle gehören zu einer Person, die dazu beraten darf. Rechtsstand August 2026.

LERNZIELE

- Du kennst die drei Fragen, die sich bei generiertem Code stellen.
- Du erkennst Anzeichen dafür, dass ein Vorschlag aus einer erkennbaren Quelle stammen könnte.
- Du kennst die Instrumente, mit denen Unternehmen damit umgehen.
- Du weißt, welche Eingaben aus Lizenzgründen problematisch sind.

4.1 Drei Fragen

Darf ich das eingeben? Fremder Code unter Vertraulichkeitsvereinbarung, Code eines Kunden, Code unter einer Lizenz, die Weitergabe einschränkt — die Eingabeseite wird selten bedacht und ist oft die klarere Frage von den dreien.

Wem gehört, was herauskommt? Ob und wann generierter Code urheberrechtlich geschützt ist, hängt von der Rechtsordnung und vom Anteil menschlicher Gestaltung ab. Praktisch relevant wird es, wenn ihr Software lizenziert, verkauft oder Gewährleistung übernimmt.

Verletzt es fremde Rechte? Modelle können in seltenen Fällen längere Passagen wiedergeben, die einem Werk aus den Trainingsdaten sehr nahekommen. Das Risiko ist bei kurzen, üblichen Konstrukten praktisch null und steigt mit der Länge und Eigenart des Ausgegebenen.

4.2 Woran man aufmerksam wird

Es gibt keine zuverlässige Erkennung, aber es gibt Anzeichen, bei denen ein zweiter Blick angebracht ist:

- **Ungewöhnliche Länge am Stück** — mehrere Dutzend Zeilen zusammenhängender, spezifischer Logik ohne Rückfrage.
- **Fremde Konventionen** — Namen, Kommentarstil oder Einrückung, die nicht zu eurem Projekt und nicht zum übrigen Vorschlag passen.

- **Mitgelieferte Kommentare mit Eigennamen** — Projektbezeichnungen, Autorennamen, Verweise auf Fehlerberichte.
- **Sehr spezifische Konstanten** — Tabellen, Suchmuster, Kennzahlen, die man nicht herleitet, sondern aus einer bestimmten Quelle hat.

Bei Verdacht ist der einfachste Schritt eine wörtliche Suche nach einer markanten Zeile im Netz. Findet sich ein Treffer in einem Projekt mit Lizenz, ist die Frage nicht mehr theoretisch.

4.3 Instrumente, mit denen Unternehmen umgehen

INSTRUMENT	WAS ES LEISTET
Werkzeugrichtlinie	Welche Assistenten erlaubt sind, in welchen Repositories, in welcher Kontoform — die Grundlage für alles Weitere
Anbieterzusagen	Manche Anbieter geben Zusagen zur Freistellung bei Ansprüchen Dritter oder bieten Filter gegen wörtliche Übereinstimmungen. Prüfen, was im Vertrag steht — nicht, was in der Werbung steht.
Herkunftsprüfung	Werkzeuge, die generierten Code gegen öffentliche Quellen abgleichen; sinnvoll ab einer bestimmten Projektgröße
Abhängigkeitsverzeichnis	Eine gepflegte Aufstellung aller Abhängigkeiten mit Lizenz — die häufigere Lizenzfrage betrifft ohnehin die Bibliotheken, nicht den generierten Code
Kennzeichnung im Commit	Vermerk, dass ein Beitrag KI-gestützt entstand. Kostet nichts und ist bei jeder späteren Frage die Grundlage

DIE PRAKTISCH HÄUFIGERE LIZENZFRAGE

In der Praxis entsteht das Lizenzproblem selten durch den generierten Code selbst, sondern durch die **Bibliothek, die der Assistent vorschlägt**. Ein Vorschlag, der eine Abhängigkeit unter einer Copyleft-Lizenz einführt, kann Folgen für die Lizenzierung eures gesamten Produkts haben — und niemand hat diese Entscheidung bewusst getroffen.

Deshalb gehört zu jeder neuen Abhängigkeit neben der Existenzprüfung aus Modul 2.3 die Lizenzprüfung. Beides ist derselbe Handgriff und dauert zwei Minuten.

BEISPIEL 4.1 — VIER SITUATIONEN

A — Der Assistent schlägt eine dreizeilige Hilfsfunktion zum Formatieren eines Datums vor. *Unbedenklich*. Kurze, übliche Konstrukte haben keine Schöpfungshöhe, und dieselbe Lösung schreiben tausend Leute gleich.

B — Der Assistent liefert eine 120-zeilige Implementierung eines speziellen Komprimierungsverfahrens, komplett mit Konstantentabelle und Kommentaren in einem fremden Stil. *Zweiter Blick*. Länge, Eigenart und die Tabelle sind alle drei Anzeichen. Wörtliche Suche, und im Zweifel eine Bibliothek verwenden statt es im Projekt zu haben.

C — Du fügst den Quelltext eines Kundenprojekts in den Chat ein, um einen Fehler zu finden. *Problematisch, und zwar auf der Eingabeseite*. Hier geht es nicht um Urheberrecht am Ergebnis, sondern um die Vertraulichkeitsvereinbarung mit dem Kunden — dazu Modul 5.

D — Der Assistent schlägt vor, eine Bibliothek unter einer starken Copyleft-Lizenz einzubinden, um ein PDF zu erzeugen. *Entscheidung außerhalb deiner Zuständigkeit*. Das ist keine technische Wahl, sondern eine, die die Lizenzierung eures Produkts betrifft. Alternative suchen oder eskalieren.

ÜBUNG 4 — ENTSCHEIDUNGEN

Entscheide für jede Situation: unbedenklich, zweiter Blick, oder eskalieren. Begründe mit den Abschnitten 4.1 bis 4.3 und nenne den konkreten nächsten Schritt.

1. Der Assistent generiert eine Sortierfunktion mit einem ungewöhnlichen Verfahren und einem Kommentar, der auf eine Veröffentlichung von 1997 verweist.
2. Ihr entwickelt ein internes Werkzeug, das nie das Haus verlässt. Der Assistent schlägt eine Bibliothek unter starkem Copyleft vor.
3. Dasselbe, aber das Werkzeug wird an Kunden ausgeliefert.
4. Eine Kollegin fügt eine Spezifikation eines Partners, die unter Vertraulichkeit steht, in den Chat ein, um daraus Datenklassen generieren zu lassen.
5. Der Assistent liefert eine Funktion, die exakt so in eurem eigenen Repository schon existiert.

Zusatzfrage: Welche zwei dieser fünf Situationen ließen sich durch eine Werkzeugrichtlinie im Voraus klären, sodass niemand einzeln entscheiden muss?

KERNAUSSAGEN

- Drei Fragen: Eingabe erlaubt, Rechte am Ergebnis, Verletzung fremder Rechte.
- Anzeichen: Länge am Stück, fremde Konventionen, Eigennamen, spezifische Konstanten.
- Die häufigere Lizenzfrage betrifft die vorgeschlagene Bibliothek, nicht den Code.
- Werkzeugrichtlinie, Anbieterzusagen, Herkunftsprüfung, Abhängigkeitsverzeichnis, Kennzeichnung im Commit.
- Lizenzentscheidungen mit Produktwirkung werden eskaliert, nicht im Diff getroffen.

Selbsttest Modul 4

1. Wo entsteht in der Praxis am häufigsten ein Lizenzproblem?

- a) Im generierten Code selbst.
- b) Bei der vom Assistenten vorgeschlagenen Abhängigkeit.
- c) In den Kommentaren.
- d) In den Testdaten.

2. Welches Anzeichen rechtfertigt einen zweiten Blick am ehesten?

- a) Eine dreizeilige Hilfsfunktion.
- b) Eine lange, eigenartige Implementierung mit spezifischer Konstantentabelle.
- c) Ein englischer Variablenname.
- d) Eine Funktion ohne Kommentare.

Geheimnisse und Datenflüsse

ca. 2,5 Stunden • Übung und Selbsttest am Ende

LERNZIELE

- Du weißt, was ein Codeassistent tatsächlich überträgt — und was du davon steuern kannst.
- Du erkennst die vier Wege, auf denen Geheimnisse in Vorschläge und Verläufe geraten.
- Du handelst richtig, wenn ein Geheimnis exponiert wurde — in der richtigen Reihenfolge.
- Du kannst die Fragen benennen, die vor der Einführung eines Assistenten zu klären sind.

5.1 Was übertragen wird

Ein Codeassistent überträgt in der Regel mehr als die Zeile, an der du stehst. Je nach Werkzeug und Einstellung kann das umfassen: die offene Datei, benachbarte oder kürzlich geöffnete Dateien, die Projektstruktur, Abhängigkeitsdateien, den Chatverlauf und — bei agentischen Werkzeugen — die Ausgabe von Befehlen, die sie ausführen.

Das ist nicht heimlich, es steht in der Dokumentation. Es wird nur selten gelesen, weil die Frage „was schickt mein Editor eigentlich?“ sich niemand stellt, solange es funktioniert.

FÜNF FRAGEN AN DEIN WERKZEUG — HEUTE BEANTWORTBAR

1. Welche Dateien werden übertragen: nur die offene, das Projekt, oder was der Agent liest?
2. Werden Eingaben zum Training verwendet? Bei Unternehmenskonten meist nicht, bei Privatkonten oft doch — und das ist der eigentliche Unterschied zwischen beiden.
3. Wie lange werden Verläufe gespeichert, und wer im Unternehmen kann sie einsehen?
4. In welcher Region wird verarbeitet, und gibt es einen Auftragsverarbeitungsvertrag?
5. Gibt es eine Ausschlussliste für Dateien und Verzeichnisse, und ist sie eingerichtet?

Frage 5 ist die, die du selbst beantworten kannst, und sie wirkt sofort.

5.2 Vier Wege, auf denen Geheimnisse abfließen

Die offene Datei. Eine `.env` im Editor geöffnet, während der Assistent aktiv ist. Die naheliegendste und häufigste Ursache.

Das Einfügen in den Chat. Eine Fehlermeldung mit vollständiger Verbindungszeichenfolge, ein Protokollauszug mit Token, eine Konfigurationsdatei „zum Kontext“.

Der generierte Code selbst. Modelle schreiben gelegentlich Platzhalter, die wie echte Zugangsdaten aussehen — und manchmal Werte, die aus Beispielen stammen. Beides landet im Commit, wenn niemand hinsieht.

Die Werkzeugausgabe bei agentischen Läufen. Ein Agent führt `env`, `git log` oder ein Skript aus; die Ausgabe geht in den Kontext und damit an den Dienst. Das ist der Weg, den man am wenigsten kommen sieht.

5.3 Wenn es passiert ist

Die richtige Reihenfolge ist gegenläufig zum Reflex. Der Reflex ist, den Chatverlauf zu löschen. Das ist der letzte Schritt und der unwichtigste.

1. **Rotieren.** Das Geheimnis ist verbrannt, sobald es das Haus verlassen hat. Neuen Schlüssel erzeugen, alten sofort ungültig machen. Alles andere ist nachrangig.
2. **Wirkung prüfen.** Was konnte man mit diesem Schlüssel? Zugriffsprotokolle des betroffenen Systems auf ungewöhnliche Zugriffe seit dem Zeitpunkt durchsehen.
3. **Melden.** Nach eurem Vorfallsprozess. Auch dann, wenn nichts passiert ist — die Meldung ist die Grundlage dafür, dass der Weg beim nächsten Mal geschlossen ist.
4. **Ursache schließen.** Ausschlussliste ergänzen, Geheimnis-Scanner vor dem Commit einrichten, Datei aus dem Repository entfernen, falls sie dort landete.
5. **Verlauf löschen,** soweit das Werkzeug es erlaubt.

Wer bei Schritt 5 anfängt, hat ein gültiges Geheimnis im Umlauf und ein gutes Gefühl.

5.4 Was man einrichtet, bevor etwas passiert

- **Ausschlussliste** für `.env`, Schlüsseldateien, Zertifikate, Konfigurationen mit Zugangsdaten, Verzeichnisse mit Kundendaten.
- **Geheimnis-Scanner vor dem Commit** — fängt sowohl deine eigenen Fehler als auch die des Assistenten.
- **Unternehmenskonto statt Privatkonto**, mit vertraglicher Grundlage und abgeschalteter Trainingsnutzung.
- **Klare Repository-Freigabe:** in welchen Projekten der Assistent laufen darf und in welchen nicht — besonders bei Kundenprojekten unter Vertraulichkeit.
- **Geheimnisse nie in Dateien**, sondern aus einem Tresor oder aus der Umgebung. Dann kann auch nichts Falsches im Kontext landen.

BEISPIEL 5.1 — EIN VORFALL IN VIER MINUTEN

Was passierte: Eine Entwicklerin debuggt einen fehlgeschlagenen Aufruf und fügt die vollständige Fehlermeldung in den Chat ein. Darin: die Verbindungszeichenfolge der Testdatenbank samt Passwort. Der Assistent antwortet hilfreich, das Problem ist gelöst, die Sache ist vergessen.

Was daran zählt: Es ist „nur“ die Testdatenbank. Aber Testdatenbanken enthalten häufig Kopien echter Daten, das Passwort ist oft dasselbe wie anderswo, und das Netz zwischen Test und Produktion ist selten so getrennt, wie das Architekturbild behauptet.

Richtiges Vorgehen: Passwort rotieren (vier Minuten). Prüfen, ob dasselbe Passwort an anderer Stelle verwendet wird — dann auch dort. Zugriffe auf die Testdatenbank seit dem Zeitpunkt durchsehen. Melden. Danach die eigentliche Ursache: Warum steht ein Passwort im Klartext in einer Fehlermeldung? Das ist der Befund, der bleibt.

ÜBUNG 5 — DATENFLÜSSE UND EIN VORFALL

Teil A: Nimm den Assistenten, mit dem du arbeitest, und beantworte die fünf Fragen aus 5.1 schriftlich, mit Beleg aus der Dokumentation oder den Einstellungen. Notiere, welche du nicht beantworten kannst.

Teil B: Ein Agent soll einen fehlschlagenden Test untersuchen. Er führt dazu selbständig `npm test`, `cat .env.test` und `git log -p` aus.

1. Welche der drei Ausgaben ist die problematischste, und warum?
2. `git log -p` ist weniger offensichtlich problematisch als es ist. Woran liegt das?
3. Welche zwei Maßnahmen aus 5.4 hätten alle drei Fälle abgedeckt?
4. Der Agent hat das Passwort aus `.env.test` in einer Zusammenfassung wiederholt, die im Ticket landete. Beschreibe dein Vorgehen in der richtigen Reihenfolge.

KERNAUSSAGEN

- Assistenten übertragen mehr als die aktuelle Zeile; was genau, steht in der Dokumentation.
- Vier Wege: offene Datei, Einfügen in den Chat, generierter Code, Werkzeugausgabe bei Agenten.
- Bei einem Vorfall zuerst rotieren, zuletzt den Verlauf löschen.
- Ausschlussliste, Geheimnis-Scanner, Unternehmenskonto, Repository-Freigabe, Tresor statt Datei.

Selbsttest Modul 5

1. Ein API-Schlüssel wurde versehentlich in den Chat eingefügt. Was ist der erste Schritt?

- a) Den Chatverlauf löschen.
- b) Den Schlüssel rotieren.
- c) Den Anbieter kontaktieren.
- d) Die Datei aus dem Repository entfernen.

2. Welcher Abflussweg wird am wenigsten erwartet?

- a) Die offene Datei.
- b) Das Einfügen in den Chat.
- c) Die Ausgabe von Befehlen, die ein Agent selbst ausführt.
- d) Der generierte Code.

Tests als Sicherheitsnetz

ca. 3 Stunden • Übung und Selbsttest am Ende

LERNZIELE

- Du erkennst die Annahmefalle: Tests, die dieselben Annahmen teilen wie der getestete Code.
- Du schreibst Testfälle aus der Spezifikation statt aus der Implementierung.
- Du beurteilst Abdeckungszahlen richtig und kennst ihre Grenze.
- Du setzt Charakterisierungstests bei Umbauten und Migrationen ein.

6.1 Die Annahmefalle

Der naheliegende Ablauf ist: Code generieren lassen, dann Tests dazu generieren lassen. Das Ergebnis ist grün und wertlos.

Denn beide stammen aus derselben Quelle und teilen dieselben Annahmen. Hat das Modell angenommen, die Liste sei nie leer, prüft der generierte Test genau diesen Fall nicht. Hat es `total` für eine Zahl gehalten, übergibt der Test eine Zahl. Der Test bestätigt, dass der Code tut, was das Modell dachte — nicht, dass er tut, was gebraucht wird.

DIE REGEL

Testfälle kommen aus der Spezifikation, nicht aus dem Code. Wenn du Tests generieren lässt, dann aus der Anforderung — und ohne die Implementierung im Kontext. Am besten schreibst du die Fälle als Liste selbst, bevor du den Code hast, und lässt dann nur noch das Gerüst erzeugen.

Ein einfaches Erkennungsmerkmal: Wenn dein Test bei einer *korrekten* alternativen Implementierung fehlschlägt, testest du die Implementierung, nicht das Verhalten.

6.2 Testfälle, die man selbst festlegt

Für praktisch jede Funktion sind das dieselben sechs Gruppen. Sie kosten fünf Minuten und finden mehr als jede generierte Testsuite:

GRUPPE	BEISPIELE
Normalfall	Zwei bis drei typische Eingaben mit erwartetem Ergebnis
Grenzen	Leer, ein Element, Maximum, genau am Schwellenwert
Ungültig	<code>null</code> , falscher Typ, negative Zahl, Zeichenkette statt Zahl
Fehler	Abhängigkeit nicht erreichbar, Zeitüberschreitung, Berechtigung fehlt
Nebenwirkungen	Wird zweimaliger Aufruf richtig behandelt? Werden Ressourcen freigegeben?
Regression	Für jeden gefundenen Fehler ein Test, der ihn festhält

Die letzte Gruppe ist die, die eine Testsuite über die Zeit wertvoll macht. Ein Fehler, der einmal auftrat und keinen Test bekam, tritt wieder auf.

6.3 Abdeckung richtig lesen

Abdeckung misst, welche Zeilen ausgeführt wurden — nicht, ob etwas geprüft wurde. Ein Test ohne einzige Zusicherung erzeugt vollständige Abdeckung.

Bei generiertem Code ist das besonders trügerisch, weil dort viel Code entsteht, der vom glücklichen Pfad durchlaufen wird. Hohe Abdeckung bei fehlender Fehlerbehandlung bedeutet nur, dass die Fehlerbehandlung gar nicht erst existiert.

Nützlicher als die Gesamtzahl sind zwei Blicke: **Welche Zweige** nicht abgedeckt sind — dort steht meist die Fehlerbehandlung. Und, wenn ihr es einsetzt, **Mutationstests**: Sie verändern den Code absichtlich und prüfen, ob ein Test das bemerkt. Das ist die einzige Messung, die die Annahmefalle aus 6.1 tatsächlich aufdeckt.

6.4 Charakterisierungstests bei Umbauten

Für die typische Assistentenaufgabe „bau dieses Modul um“ gibt es ein Verfahren, das den Unterschied zwischen einem ruhigen und einem unruhigen Nachmittag macht:

1. **Vor** dem Umbau Tests schreiben, die das *heutige* Verhalten festhalten — auch das, was dir nicht gefällt. Diese Tests bewerten nicht, sie beschreiben.
2. Sicherstellen, dass sie auf dem alten Stand grün sind.
3. Umbauen lassen.
4. Dieselben Tests laufen lassen. Jede Abweichung ist eine Verhaltensänderung — entweder gewollt und dann dokumentiert, oder ungewollt und dann ein Fehler.

Ohne diesen Schritt beantwortest du nach einem großen Umbau die Frage „hat sich etwas geändert?“ mit einem Gefühl. Mit ihm beantwortest du sie mit einer Liste.

BEISPIEL 6.1 — DERSELBE CODE, ZWEI TESTSUITEN

Die Funktion: `calculateDiscount(order, customer)` — Rabatt in Prozent, gestaffelt nach Bestellwert, mit Zuschlag für Stammkunden, Obergrenze 30 %.

Generierte Suite (aus dem Code erzeugt): fünf Tests. Bestellwert 50 → 0 %. Bestellwert 500 → 5 %. Bestellwert 5.000 → 10 %. Stammkunde bei 500 → 10 %. Nicht-Stammkunde bei 500 → 5 %. Alle grün, Abdeckung 94 %.

Was sie nicht prüft: Bestellwert 0. Negativer Bestellwert (Gutschrift). Genau der Schwellenwert 499,99 gegen 500,00. Die Obergrenze von 30 % — der Fall, in dem Staffeln und Stammkundenzuschlag zusammen darüber liegen. `customer` ist `null`. Rundung bei 5,555 %.

Der entscheidende Fall ist die Obergrenze: Die Implementierung addiert Staffeln und Zuschlag und begrenzt anschließend — oder sie begrenzt zuerst und addiert dann. Das sind zwei verschiedene Ergebnisse, beide plausibel, und die Spezifikation sagt welches gilt. Die generierte Suite prüft es nicht, weil das Modell die Reihenfolge, die es selbst gewählt hat, für die richtige hält.

Aufwand für die fehlenden sechs Fälle: etwa zehn Minuten. *Wert:* genau die Fälle, wegen denen später jemand eine Gutschrift ausstellt.

ÜBUNG 6 — TESTPLAN GEGEN DIE ANNAHMEFALLE

Spezifikation: `splitPayment(amount, participants)` teilt einen Betrag in Cent gleichmäßig auf Teilnehmer auf. Rest wird auf die ersten Teilnehmer verteilt. Die Summe der Teilbeträge muss exakt dem Gesamtbetrag entsprechen.

1. Schreibe die Testfälle nach den sechs Gruppen aus 6.2 — mindestens zwölf, mit erwartetem Ergebnis.
2. Drei deiner Fälle würde eine aus dem Code generierte Suite mit hoher Wahrscheinlichkeit nicht enthalten. Welche, und warum nicht?
3. Ein Fall prüft die zentrale Zusicherung der Spezifikation, unabhängig von der Implementierung. Welcher?
4. Formuliere einen Test, der bei einer korrekten alternativen Implementierung ebenfalls grün ist — und einen, der das nicht wäre. Was unterscheidet sie?
5. Die Funktion soll umgebaut werden, um zusätzlich ungleiche Anteile zu unterstützen. Welche Charakterisierungstests schreibst du vorher?

KERNAUSSAGEN

- Code und Tests aus derselben Quelle teilen dieselben Annahmen — und dieselben blinden Flecken.
- Testfälle aus der Spezifikation, ohne Implementierung im Kontext.
- Sechs Gruppen: Normalfall, Grenzen, Ungültig, Fehler, Nebenwirkungen, Regression.
- Abdeckung misst Ausführung, nicht Prüfung; Zweigabdeckung und Mutationstests sagen mehr.
- Vor jedem größeren Umbau Charakterisierungstests — sie machen Verhaltensänderungen sichtbar.

Selbsttest Modul 6

1. Warum sind aus dem Code generierte Tests trügerisch?

- a) Sie laufen langsamer.
- b) Sie teilen die Annahmen des Codes und prüfen deshalb dessen blinde Flecken nicht.
- c) Sie erzeugen zu wenig Abdeckung.
- d) Sie sind schwer zu lesen.

2. Wozu dienen Charakterisierungstests?

- a) Neue Anforderungen zu prüfen.
- b) Das heutige Verhalten festzuhalten, damit ein Umbau sichtbare Abweichungen erzeugt.
- c) Die Abdeckung zu erhöhen.
- d) Die Laufzeit zu messen.

Agentische Arbeitsweisen

ca. 3 Stunden · Übung und Selbsttest am Ende

LERNZIELE

- Du erkennst, wann eine agentische Aufgabe kippt — vorher, nicht hinterher.
- Du schneidest Aufträge so zu, dass der Diff prüfbar bleibt.
- Du legst Wirkungsbereich und Abbruchkriterien fest, bevor du startest.
- Du hältst Merge-Disziplin unter Zeitdruck.

7.1 Was ein Agent anders macht

Ein Agent liest Dateien, führt Befehle aus, ändert mehrere Stellen und wiederholt das, bis er sein Ziel für erreicht hält. Der Gewinn ist echt: Aufgaben, die über Dateigrenzen laufen — eine Umbenennung, eine Migration, das Nachziehen einer Schnittstellenänderung — waren bisher mühsam und sind jetzt schnell.

Der Preis ist ebenso echt und wird unterschätzt: Der Agent erzeugt in fünf Minuten mehr Änderung, als ein Mensch in einer Stunde prüft. Damit verschiebt sich der Engpass von der Erzeugung zur Prüfung — und das ist die eine Tatsache, um die dieses Modul kreist.

7.2 Wann eine Aufgabe kippt

Vier Merkmale, die einzeln handhabbar und zusammen ein Problem sind:

MERKMAL	WARUM ES KIPPT
Unscharfes Ziel	Ohne Abnahmekriterium weiß weder der Agent noch du, wann fertig ist
Großer Wirkungsbereich	Viele Dateien heißt vieles, was du prüfen musst und nicht wirst
Keine Tests	Ohne automatische Rückmeldung fehlt dem Agenten das Abbruchsignal — und dir die Kontrolle
Unbekanntes Gebiet	Du kannst den Diff nicht beurteilen, also prüfst du ihn faktisch nicht

Ein Auftrag mit drei oder vier dieser Merkmale gehört nicht an einen Agenten, sondern in eine Zerlegung. Ein Auftrag mit keinem ist die ideale Agentenaufgabe: klar umrissen, wenige Dateien, grüne Tests als Abbruchkriterium, vertrautes Gebiet.

7.3 Auftrag zuschneiden

Fünf Angaben, die vor dem Start feststehen:

1. **Ein Ziel**, in einem Satz. Wenn du „und“ brauchst, sind es zwei Aufträge.

2. **Wirkungsbereich:** welche Verzeichnisse er ändern darf und welche ausdrücklich nicht.
3. **Abbruchkriterium:** „fertig, wenn npm test grün ist und der Typprüfer nichts meldet". Ein Agent ohne messbares Ziel arbeitet, bis er sich selbst überzeugt hat.
4. **Verbote:** keine neuen Abhängigkeiten, keine Formatierung fremder Dateien, keine Schemaänderungen, keine Migrationsskripte.
5. **Größenbudget:** Wenn der Diff größer wird als etwa 300 Zeilen, abbrechen und zerlegen. Diese Zahl ist nicht heilig, aber irgendeine Zahl muss es sein — sonst entscheidet die Müdigkeit.

DIE FORMATIERUNGSFALLE

Ein Agent, der nebenbei die Formatierung in acht Dateien anpasst, erzeugt einen Diff, in dem die drei Zeilen echter Änderung nicht mehr zu finden sind. Das ist der häufigste Grund, warum ein Review faktisch nicht stattfindet — nicht Nachlässigkeit, sondern Unauffindbarkeit.

Gegenmittel: Formatierung als Verbot in den Auftrag, und in eurem Projekt einen Formatierer, der ohnehin automatisch und getrennt läuft. Dann sind Formatänderungen nie Teil eines inhaltlichen Diffs.

7.4 Merge-Disziplin

Am Ende steht immer dieselbe Situation: Ein großer Diff, grüne Tests, Zeitdruck. Drei Regeln, die genau dann gelten:

Grüne Tests sind kein Review. Sie sagen, dass die geprüften Annahmen halten. Modul 6 erklärt, warum das bei generiertem Code weniger bedeutet als sonst.

Ungelesenes wird nicht gemerged. Wenn du den Diff nicht durchgesehen hast, ist die ehrliche Antwort, den Auftrag zu zerlegen und erneut laufen zu lassen — nicht, ihn durchzuwinken und sich vorzunehmen, später hinzusehen.

Der Commit trägt deinen Namen. Vermerke, dass er KI-gestützt entstand — das ist die Kennzeichnung aus Modul 4 und kostet eine Zeile. Aber die Verantwortung wandert damit nicht mit. Sie bleibt bei dir.

BEISPIEL 7.1 — DERSELBE AUFTRAG, ZWEIMAL ZUGESCHNITTEN

KIPPT

„Migriere unser Backend von der alten Datenbankbibliothek auf die neue und mach die Tests wieder grün.“

47 Dateien · 2.100 Zeilen Diff
• 3 neue Abhängigkeiten
• Formatierung überall
• niemand liest das

PRÜFBAR

Auftrag 1: Nur `src/db/adapter.ts` auf die neue Bibliothek umstellen. Schnittstelle nach außen unverändert. Fertig, wenn `adapter.spec.ts` grün ist. Keine neuen Abhängigkeiten, keine Formatierung anderer Dateien.

Dann Auftrag 2 für die Aufrufer, in drei Teilen nach Modul.

Was den Unterschied macht, ist nicht die Fähigkeit des Agenten — beide Aufträge kann er ausführen. Es ist die Frage, ob am Ende jemand das Ergebnis beurteilen konnte. Rechts entstehen vier Diffs zu je 200 bis 400 Zeilen, jeder in fünfzehn Minuten prüfbar. Links entsteht ein Diff, der gemerged wird, weil die Tests grün sind.

Und der Nebeneffekt: Wenn Auftrag 2 scheitert, steht Auftrag 1 bereits im Hauptzweig und muss nicht wiederholt werden.

ÜBUNG 7 — ZERLEGEN UND BEGRENZEN

Aufgabe aus dem Sprint: *„Wir wollen Mandantenfähigkeit einführen. Jede Anfrage gehört zu einer Organisation, alle Abfragen müssen danach filtern, und die bestehenden Daten brauchen eine Standardorganisation.“*

1. Bewerte die Aufgabe nach den vier Merkmalen aus 7.2. Wie viele treffen zu?
2. Zerlege sie in mindestens vier Agentenaufträge, jeder mit den fünf Angaben aus 7.3.
3. Einer deiner Aufträge darf auf keinen Fall von einem Agenten ausgeführt werden. Welcher, und warum?
4. Welches Abbruchkriterium verwendest du für den Auftrag, der die Abfragen umstellt — und warum reichen grüne Tests hier nicht?
5. Nach Auftrag 2 hat der Agent 900 Zeilen geändert, die Tests sind grün, und du hast zwanzig Minuten bis zum Feierabend. Was tust du?

KERNAUSSAGEN

- Der Engpass verschiebt sich von der Erzeugung zur Prüfung.
- Vier Kippmerkmale: unscharfes Ziel, großer Wirkungsbereich, keine Tests, unbekanntes Gebiet.
- Fünf Angaben vor dem Start, einschließlich Größenbudget.
- Formatierung gehört nie in einen inhaltlichen Diff.
- Grüne Tests sind kein Review. Ungelesenes wird nicht gemerged.

Selbsttest Modul 7

1. Wohin verschiebt sich der Engpass beim agentischen Arbeiten?

- a) Zur Erzeugung.
- b) Zur Prüfung.
- c) Zur Ausführung der Tests.
- d) Zur Formatierung.

2. Warum ist ein Agent, der nebenbei formatiert, ein Reviewproblem?

- a) Die Formatierung ist meist falsch.
- b) Die inhaltliche Änderung ist im Diff nicht mehr auffindbar.
- c) Es verlangsamt die Tests.
- d) Es erzeugt Konflikte im Formatierer.

Musterprüfung

25 Fragen · 50 Minuten · Bestehensgrenze 75 % (19 von 25)

Unter realen Bedingungen bearbeiten. Genau eine Antwort ist richtig. Lösungen mit Begründung ab Seite 44.

- 1. Wonach wählt man den Arbeitsmodus sinnvollerweise aus?**
 - a) Nach der gesparten Schreibzeit.
 - b) Nach dem entstehenden Reviewaufwand.
 - c) Nach der Sprache.
 - d) Nach der Teamgröße.

- 2. Welcher Kontext hilft am zuverlässigsten, die Konventionen eines Projekts zu treffen?**
 - a) Das gesamte Repository.
 - b) Ein bestehendes Beispiel derselben Art aus dem Projekt.
 - c) Eine Beschreibung des Stils in Prosa.
 - d) Die Linter-Konfiguration.

- 3. Warum gehört ein Abschnitt „Nicht tun“ in die Spezifikation?**
 - a) Um Token zu sparen.
 - b) Weil Modelle sonst Abhängigkeiten, Caches und Protokollierung ergänzen.
 - c) Weil es die Testabdeckung erhöht.
 - d) Weil es die Antwort verkürzt.

- 4. Bei welcher Aufgabe kostet ein Assistent am ehesten mehr, als er bringt?**
 - a) Boilerplate erzeugen.
 - b) Fehlersuche in unbekanntem Fremdcode.
 - c) Eine Funktion nach Spezifikation schreiben.
 - d) Testgerüste anlegen.

- 5. Was ist die zentrale Tatsache hinter der Tempo-Falle?**
 - a) Modelle werden schneller.
 - b) Erzeugen ist schneller geworden, Lesen nicht.
 - c) Tests laufen länger.
 - d) Reviews sind unbeliebt.

- 6. Welche Fehlerklasse ist in generiertem Code überdurchschnittlich häufig?**
 - a) Syntaxfehler.
 - b) Nicht freigegebene Ressourcen.
 - c) Falsche Einrückung.
 - d) Fehlende Kommentare.

7. Warum ist ein halluzinierter Paketname ein Sicherheitsproblem?

- a) Er verlängert die Installation.
- b) Modelle erfinden wiederholt denselben Namen, den ein Angreifer registrieren kann.
- c) Er verletzt Lizenzbedingungen.
- d) Er erhöht die Bundle-Größe.

8. Womit beginnt ein Review von generiertem Code?

- a) Mit dem Stil.
- b) Mit dem Abgleich gegen die Spezifikation.
- c) Mit den Tests.
- d) Mit den Abhängigkeiten.

9. Ein leerer `catch`-Block bewirkt am ehesten:

- a) bessere Laufzeit.
- b) einen Fehler, der als Erfolg gemeldet wird.
- c) einen Absturz.
- d) eine Warnung des Linters, die genügt.

10. Warum enthält generierter Code systematisch weniger Sicherheitsprüfungen?

- a) Modelle lehnen Sicherheit ab.
- b) Beispielcode ist auf Kürze optimiert und lässt Prüfungen weg.
- c) Sicherheitsprüfungen sind schlecht dokumentiert.
- d) Sie verlangsamen die Ausführung.

11. Was ist der Unterschied zwischen Authentifizierung und Autorisierung?

- a) Kein praktischer.
- b) Authentifizierung klärt wer, Autorisierung klärt was dieser Jemand darf.
- c) Autorisierung ist die Anmeldung.
- d) Authentifizierung betrifft nur APIs.

12. Warum antwortet man bei fehlender Berechtigung oft mit 404 statt 403?

- a) 403 ist veraltet.
- b) 403 verrät, dass die Ressource existiert.
- c) 404 ist schneller.
- d) Browser behandeln 403 falsch.

13. Wie behebt man zusammengesetzte SQL-Abfragen richtig?

- a) Eingaben maskieren.
- b) Struktur und Daten trennen — parametrisierte Abfragen.
- c) Die Abfrage kürzen.
- d) Ein ORM verwenden reicht immer.

14. Welchen Befund findet ein statisches Analysewerkzeug am wenigsten zuverlässig?

- a) Hartkodierte Zugangsdaten.
- b) Verkettete SQL-Abfragen.
- c) Fehlende Autorisierungsprüfung.
- d) Veraltete Abhängigkeiten.

15. Wo entsteht in der Praxis am häufigsten ein Lizenzproblem?

- a) Im generierten Code selbst.
- b) Bei der vorgeschlagenen Abhängigkeit.
- c) In den Kommentaren.
- d) In den Testdaten.

16. Welches Anzeichen rechtfertigt einen zweiten Blick auf die Herkunft?

- a) Eine dreizeilige Hilfsfunktion.
- b) Eine lange, eigenartige Implementierung mit spezifischer Konstantentabelle.
- c) Englische Bezeichner.
- d) Fehlende Typannotationen.

17. Welche Frage an das eigene Werkzeug kann man selbst beantworten und sofort umsetzen?

- a) Wo wird verarbeitet?
- b) Wird zum Training verwendet?
- c) Ist eine Ausschlussliste eingerichtet?
- d) Wie lange werden Verläufe gespeichert?

18. Ein Schlüssel wurde in den Chat eingefügt. Was ist der erste Schritt?

- a) Verlauf löschen.
- b) Rotieren.
- c) Anbieter informieren.
- d) Commit rückgängig machen.

19. Welcher Abflussweg für Geheimnisse wird am wenigsten erwartet?

- a) Offene Datei.
- b) Einfügen in den Chat.
- c) Ausgabe von Befehlen, die ein Agent selbst ausführt.
- d) Generierter Code.

20. Warum sind aus dem Code generierte Tests trügerisch?

- a) Sie sind zu kurz.
- b) Sie teilen die Annahmen des Codes.
- c) Sie laufen langsamer.
- d) Sie prüfen zu viele Fälle.

21. Woran erkennt man einen Test, der die Implementierung statt das Verhalten prüft?

- a) Er ist lang.
- b) Er schlägt bei einer korrekten alternativen Implementierung fehl.
- c) Er hat keine Zusicherung.
- d) Er läuft nur lokal.

22. Was misst Testabdeckung?

- a) Ob etwas geprüft wurde.
- b) Welche Zeilen ausgeführt wurden.
- c) Die Qualität der Zusicherungen.
- d) Die Zahl der Fehler.

23. Wozu dienen Charakterisierungstests?

- a) Neue Anforderungen prüfen.
- b) Heutiges Verhalten festhalten, damit ein Umbau Abweichungen sichtbar macht.
- c) Abdeckung erhöhen.
- d) Laufzeit messen.

24. Welches Merkmal lässt eine agentische Aufgabe am ehesten kippen?

- a) Ein klar umrissenes Ziel.
- b) Ein großer Wirkungsbereich ohne Tests.
- c) Ein vertrautes Gebiet.
- d) Ein kleiner Diff.

25. Grüne Tests nach einem Agentenlauf bedeuten:

- a) Der Diff kann gemerged werden.
- b) Die geprüften Annahmen halten — ein Review ersetzt das nicht.
- c) Der Code ist sicher.
- d) Die Abdeckung ist ausreichend.

Lösungen

Übungen · Selbsttests · Musterprüfung

Übungen

Übung 1 — Vom Ticket zur Spezifikation

Rückfragen: Wie lange sollen Dateien aufbewahrt werden — 90 Tage, ein Jahr, unbegrenzt? Wo liegen sie, und was kostet der Speicher? Wer darf herunterladen, und wie wird das geprüft? Sind die Dateien personenbezogen — dann gilt eine Löschfrist, keine Aufbewahrungsfrist? Was passiert mit bestehenden Dateien, die schon älter als 30 Tage sind?

Nicht generieren lassen: die Aufbewahrungsentscheidung und die Berechtigungslogik. Beides sind fachliche und teils rechtliche Entscheidungen — ein Assistent liefert dir die verbreitetste Variante, und die verbreitetste Variante bei Aufbewahrungsfristen ist regelmäßig die falsche.

Kontext: die bestehende Storage-Schnittstelle, ein vorhandener Download-Endpunkt als Vorbild, euer Berechtigungshelfer, das Schema der Export-Tabelle. *Nicht* mitgeben: das gesamte Repository und alles mit Kundendaten.

Übung 2 — Review

Befunde:

- 1 — Fehlende Autorisierung** (Sicherheit, in 2.2 nicht enthalten, aber Reviewpunkt 3): Niemand prüft, ob der Aufrufende diese Profile zusammenführen darf.
- 2 — Grenzfall:** `db.get_user` kann `None` liefern. Weder für `primary` noch für `dup` wird das geprüft; der nächste Zugriff bricht ab — nachdem möglicherweise bereits Nutzer gelöscht wurden.
- 3 — Falsche Bedingung:** `if not primary.get(key)` ist wahr für `None`, `""`, `0` und `False`. Ein bewusst auf `False` gesetztes Feld (etwa `newsletter_opt_in`) wird vom Duplikat überschrieben. Das ist die Leerwertfalle aus dem No-Code-Track, in Python-Form.
- 4 — Nebenwirkung vor Zusicherung:** `db.delete_user` läuft in der Schleife, `db.save_user(primary)` erst danach. Bricht es beim dritten Duplikat ab, sind zwei Nutzer gelöscht und das zusammengeführte Profil nicht gespeichert — die Daten sind weg.
- 5 — Keine Transaktion:** Mehrere schreibende Operationen ohne gemeinsame Klammer. Zusammen mit 4 ist das ein Datenverlustszenario.
- 6 — Kein Schutz gegen Selbstzusammenführung:** Ist `primary_id` in `duplicate_ids` enthalten, löscht die Funktion den Zielnutzer.
- 7 — Protokollierung:** `log.info(... {dup})` schreibt das *vollständige Nutzerprofil* ins Protokoll — personenbezogene Daten, möglicherweise besondere Kategorien, in einem System mit anderer Aufbewahrung und anderem Zugriffskreis. Das ist der Befund aus Frage 3.

Ablehnen statt anmerken würde ich wegen Befund 4/5 (Datenverlust) und Befund 1 (fehlende Autorisierung). Beides sind keine Detailkorrekturen, sondern ein anderer Entwurf.

In der Spezifikation hätte stehen müssen: „alles in einer Transaktion“, „Zusammenführen darf nur, wer beide Profile verwalten darf“, „ein Feld gilt nur dann als leer, wenn es `None` ist“, und „keine Profildaten protokollieren“. Das sind vier Zeilen, die drei bis vier der Befunde verhindert hätten.

Der aufdeckende Testfall: Zwei Duplikate, beim zweiten wirft `db.delete_user` eine Ausnahme. Erwartung: Der primäre Nutzer ist unverändert, kein Duplikat gelöscht. Dieser Test schlägt bei der vorliegenden Implementierung fehl und erzwingt die Transaktion.

Übung 3 — Sicherheitsreview

Befunde: (1) *Kontenaufzählung* — die Antwort verrät, ob eine Adresse registriert ist (Muster 6/ Informationspreisgabe). (2) *Rückgabe der `user_id`* an einen nicht authentifizierten Aufrufer. (3) *Vorhersagbarer Code* aus dem allgemeinen Zufallsgenerator, sechsstellig, ohne kryptografische Quelle (Muster 3). (4) *Keine Ablaufzeit* des Codes. (5) *Keine Versuchsbeschränkung* beim Bestätigen — sechsstellig ohne Beschränkung ist durchprobierbar (Muster 5). (6) *Keine Ratenbeschränkung* auf dem Anforderungsendpunkt (Muster 5). (7) *MD5 als Passworthash*, ohne Salt, ohne Kostenfaktor (Muster 3). (8) *Nicht-konstanter Vergleich* des Tokens. (9) *Token wird nach Verwendung nicht gelöscht* — mehrfach nutzbar.

Übernahme eines fremden Kontos entsteht aus (2) und (5): Der Angreifer erfährt über den Anforderungsendpunkt die `user_id` zu einer beliebigen Adresse und kann danach den sechsstelligen Code unbegrenzt durchprobieren. Erst zusammen ergeben die beiden Befunde einen vollständigen Weg — einzeln wirken beide harmlos. Genau deshalb prüft man Endpunkte im Zusammenhang.

Scheinbar harmlose Preisgabe: die unterschiedliche Antwort bei unbekannter Adresse. Für sich „nur“ eine Auskunft, tatsächlich der Baustein, mit dem man eine Liste gültiger Konten erzeugt — und die Grundlage für gezielte Angriffe auf diese Konten.

Werkzeug findet: MD5 als Passworthash und den schwachen Zufallsgenerator — beides bekannte Muster. **Werkzeug findet nicht:** Kontenaufzählung, fehlende Versuchsbeschränkung, fehlende Ablaufzeit, mehrfach nutzbares Token. Das sind fachliche Lücken, keine verbotenen Aufrufe.

Korrigierte erste Funktion: immer dieselbe Antwort, unabhängig davon ob das Konto existiert (gegen 1); keine `user_id` zurückgeben (gegen 2); Token aus kryptografischer Quelle mit ausreichender Länge (gegen 3); Ablaufzeit setzen, etwa 15 Minuten (gegen 4); Ratenbeschränkung je Adresse und je IP (gegen 6); Token nur als Hash speichern, damit ein Datenbankzugriff ihn nicht preisgibt.

Übung 4 — Entscheidungen

1 Zweiter Blick. Ungewöhnliches Verfahren plus Verweis auf eine Veröffentlichung sind zwei Anzeichen. Nächster Schritt: wörtliche Suche nach einer markanten Zeile; wenn es aus einer Bibliothek stammt, die Bibliothek verwenden.

2 Zweiter Blick, aber vermutlich unbedenklich: Starkes Copyleft entfaltet seine Wirkung typischerweise bei Weitergabe. Ein rein internes Werkzeug wird nicht weitergegeben. Trotzdem im Abhängigkeitsverzeichnis vermerken — Werkzeuge werden später ausgeliefert.

3 Eskalieren. Dieselbe Bibliothek, andere Lage: Auslieferung kann die Lizenzierung des Produkts betreffen. Keine Entwicklerentscheidung.

4 Eskalieren, und zwar sofort — hier ist bereits etwas passiert. Vertraulichkeitsvereinbarung mit dem Partner, Eingabeseite, siehe Modul 5.

5 Unbedenklich lizenzrechtlich, aber ein Befund anderer Art: Der Assistent hat euren eigenen Code reproduziert. Nicht duplizieren, sondern die bestehende Funktion verwenden.

Zusatzfrage: Situation 2 und 3 lassen sich durch eine Lizenzrichtlinie im Voraus klären („welche Lizenzen sind in ausgelieferten Produkten zulässig“), Situation 4 durch eine Werkzeugrichtlinie („in welchen Projekten und mit welchen Daten darf der Assistent laufen“). Damit muss niemand im Einzelfall entscheiden — und das ist der eigentliche Zweck solcher Richtlinien.

Übung 5 — Datenflüsse und ein Vorfall

Teil B, 1: `cat .env.test` ist die problematischste — sie legt Zugangsdaten direkt offen. `npm test` ist meist harmlos, kann aber Ausgaben mit Daten enthalten.

2: `git log -p` zeigt *vollständige historische Diffs*. Selbst wenn ein Geheimnis vor zwei Jahren aus dem Code entfernt wurde, steht es in der Historie — und wird damit übertragen. Genau deshalb ist „wir haben den Schlüssel doch längst rausgenommen“ keine Entwarnung: Er ist noch da, nur nicht mehr im aktuellen Stand.

3: Eine Ausschlussliste, die `.env*` erfasst, und die Regel, dass Geheimnisse nie in Dateien liegen, sondern aus einem Tresor kommen. Die zweite deckt zusätzlich den Historienfall ab, weil dann auch früher nie eines im Repository lag.

4: Passwort der Testdatenbank rotieren. Prüfen, ob es anderswo wiederverwendet wird — dann dort ebenfalls. Zugriffe seit dem Zeitpunkt durchsehen. Nach eurem Prozess melden. Die Zusammenfassung aus dem Ticket entfernen und prüfen, wer es gesehen hat. Ausschlussliste ergänzen. Zuletzt den Agentenverlauf löschen, soweit möglich.

Übung 6 — Testplan gegen die Annahmefalle

Fälle (Auswahl): *Normalfall* — 300 Cent auf 3 → [100,100,100]; 301 auf 3 → [101,100,100]. *Grenzen* — 1 Cent auf 3 → [1,0,0]; 0 auf 3 → [0,0,0]; Betrag auf 1 Teilnehmer → alles; sehr großer Betrag. *Ungültig* — 0 Teilnehmer; negative Teilnehmerzahl; negativer Betrag; Nachkommastellen im Betrag. *Fehler* — `null` statt Liste. *Nebenwirkungen* — die Eingabeliste wird nicht verändert. *Regression* — je Fund einer.

Wahrscheinlich nicht in einer generierten Suite: 1 Cent auf 3 Teilnehmer (das Modell denkt in teilbaren Beträgen), 0 Teilnehmer (Division durch null — das Modell nimmt an, das komme nicht vor), und die Unveränderlichkeit der Eingabeliste (eine Nebenwirkung, an die man nur denkt, wenn man sie einmal erlebt hat).

Die zentrale Zusicherung: `sum(result) == amount`, für jede Eingabe. Dieser eine Test ist implementierungsunabhängig und fängt jeden Rundungsfehler — er ist mehr wert als die übrigen elf zusammen und lässt sich als Eigenschaftstest über viele Zufallseingaben laufen.

Implementierungsunabhängig vs. -abhängig: „Die Summe stimmt und kein Anteil weicht um mehr als 1 Cent vom Durchschnitt ab“ ist bei jeder korrekten Verteilung grün. „Das Ergebnis ist exakt [101,100,100]“ legt fest, *welche* Teilnehmer den Rest bekommen — das ist nur dann berechtigt, wenn die Spezifikation es vorschreibt. Hier tut sie es („die ersten“), also ist der Test zulässig; wäre die Reihenfolge offen, wäre er ein Implementierungstest.

Charakterisierungstests vor dem Umbau: Das heutige Verhalten für alle Fälle oben festhalten, einschließlich der Restverteilung auf die ersten Teilnehmer — gerade weil man beim Umbau auf ungleiche Anteile versucht sein wird, sie zu ändern.

Übung 7 — Zerlegen und begrenzen

1. Alle vier Merkmale treffen zu: Das Ziel ist unscharf („alle Abfragen“), der Wirkungsbereich ist das gesamte Backend, Tests decken Mandantentrennung sicher nicht ab, und Mandantenfähigkeit ist für die meisten Teams unvertrautes Gebiet. Das ist der Lehrbuchfall für Zerlegung.

2. Zerlegung: (a) Schema: Spalte `org_id` ergänzen, nullbar, ohne Nutzung — nur Migration, kein Anwendungscode. (b) Kontext: `org_id` aus der Anfrage ermitteln und im Anfragekontext bereitstellen; ein Modul, klare Tests. (c) Datenzugriff: eine Zugriffsschicht, die `org_id` zwingend erwartet — Fertig-Kriterium: der Typprüfer meldet jede Aufrufstelle, die sie nicht übergibt. (d) Aufrufer umstellen, in Teilen nach Fachbereich, je ein Auftrag. (e) Bestandsdaten: Standardorganisation setzen. (f) Spalte auf `NOT NULL` ziehen.

3. Nicht vom Agenten: Auftrag (e), die Migration der Bestandsdaten. Sie ist einmalig, wirkt auf echte Daten und ist nicht ohne Weiteres umkehrbar — genau die Kombination, bei der ein Mensch schreibt, prüft und auf einer Kopie testet.

4. Abbruchkriterium für (c)/(d): Grüne Tests reichen nicht, weil kein bestehender Test Mandantentrennung prüft — er kann also nicht fehlschlagen, wenn sie fehlt. Besseres Kriterium: Der Typprüfer meldet keine Aufrufstelle mehr ohne `org_id`, *und* ein neu geschriebener Test weist nach, dass ein Zugriff mit fremder `org_id` nichts liefert. Das ist der entscheidende Test, und er muss vor dem Umbau existieren.

5. Nicht mergen. Der Diff ist über dem Budget, du hast zwanzig Minuten, und dies ist genau die Situation aus 7.4. Auftrag zerlegen und die Teile morgen erneut laufen lassen — der Agent braucht dafür wieder fünf Minuten. Die Alternative ist, 900 ungelesene Zeilen an Mandantentrennung in den Hauptzweig zu bringen, und das ist der Diff, über den man später im Postmortem spricht.

Selbsttests

Modul 1: 1 b · 2 b **Modul 2:** 1 b · 2 b **Modul 3:** 1 b · 2 c **Modul 4:** 1 b · 2 b

Modul 5: 1 b · 2 c **Modul 6:** 1 b · 2 b **Modul 7:** 1 b · 2 b

Musterprüfung

NR.	LÖSUNG	BEGRÜNDUNG
1	b	Der Reviewaufwand ist der begrenzende Faktor (1.1).
2	b	Ein Beispiel überträgt Konventionen zuverlässiger als Prosa (1.2).
3	b	Modelle ergänzen, was in Trainingsdaten üblich war (1.3).
4	b	Plausible Erklärungen für nicht beobachtbares Verhalten (1.3, Kasten).
5	b	Das Missverhältnis von Erzeugen und Lesen (2.1).
6	b	Ressourcen werden geöffnet und nicht freigegeben (2.2).
7	b	Wiederholt erfundene Namen sind registrierbar (2.3).
8	b	Richtigkeit vor Stil (2.4).
9	b	Fehler wird verschluckt, Funktion meldet Erfolg (Beispiel 2.1).
10	b	Beispielcode ist auf Kürze optimiert (3.1).
11	b	Wer gegen was darf (3.2, Muster 1).
12	b	403 bestätigt die Existenz der Ressource (3.2).
13	b	Struktur und Daten trennen (3.2, Muster 2).
14	c	Fachliche Berechtigungslogik findet nur ein Mensch (3.3).
15	b	Die vorgeschlagene Abhängigkeit (4.3, Kasten).
16	b	Länge, Eigenart, spezifische Konstanten (4.2).
17	c	Die Ausschlussliste richtest du selbst ein (5.1).
18	b	Rotieren zuerst, Verlauf zuletzt (5.3).
19	c	Werkzeugausgabe bei agentischen Läufen (5.2).
20	b	Gleiche Quelle, gleiche Annahmen (6.1).
21	b	Das Erkennungsmerkmal aus dem Kasten in 6.1.
22	b	Ausführung, nicht Prüfung (6.3).
23	b	Heutiges Verhalten festhalten (6.4).
24	b	Großer Wirkungsbereich ohne Abbruchsignal (7.2).
25	b	Grüne Tests sind kein Review (7.4).

AUSWERTUNG

23-25 richtig — sehr sicher, tritt zur Prüfung an.

19-22 richtig — bestanden. Die echte Prüfung enthält Szenariofragen mit Codeauszügen; arbeite die betroffenen Module nach.

unter 19 — noch nicht ausreichend. Wiederhole die Module einschließlich der Übungen an echtem Code und die Musterprüfung frühestens nach zwei Tagen.

Glossar

Abbruchkriterium

Messbare Bedingung, bei der ein Agentenauftrag als erledigt gilt — etwa grüne Tests und ein sauberer Typprüfer.

Abdeckung

Anteil der bei Tests ausgeführten Zeilen oder Zweige. Misst Ausführung, nicht Prüfung.

Annahmefalle

Tests, die aus derselben Quelle stammen wie der Code und dessen blinde Flecken teilen.

Autorisierung

Prüfung, ob eine bekannte Person eine bestimmte Handlung an einer bestimmten Ressource ausführen darf.

Charakterisierungstest

Test, der bestehendes Verhalten beschreibt statt es zu bewerten; Grundlage für sichere Umbauten.

Halluzinierte Abhängigkeit

Vom Modell erfundener Paketname; ein Lieferkettenrisiko, wenn ein Angreifer ihn registriert.

Herkunftsprüfung

Abgleich generierten Codes gegen öffentliche Quellen zur Erkennung wörtlicher Übernahmen.

Mutationstest

Verfahren, das Code absichtlich verändert und prüft, ob Tests das bemerken.

Reviewreihenfolge

Richtigkeit, Grenzen, Sicherheit, Fremdes, Passung, Stil — in dieser Folge.

Spezifikation

Auftrag an einen Assistenten mit Signatur, Randbedingungen, Verboten und Abnahmekriterium.

Tempo-Falle

Erzeugen wurde schneller, Lesen nicht; die Prüftiefe sinkt mit jedem angenommenen Vorschlag.

Wirkungsbereich

Die Dateien und Verzeichnisse, die ein Agentenauftrag verändern darf.

Prüfungsinformationen

Fragen	25, Einfachauswahl und Szenariofragen mit Codeauszügen
Dauer	50 Minuten, ohne Unterbrechung
Bestehensgrenze	75 % (19 von 25)
Voraussetzung	AI Foundations Professional (AF) oder anerkannte Gleichwertigkeit
Hilfsmittel	keine
Bewertung	deterministisch gegen einen von Fachexperten freigegebenen Antwortschlüssel; kein KI-Einsatz
Wiederholung	nach 24 Stunden, im Rahmen der erworbenen Versuche
Gültigkeit	befristet; Rezertifizierung nach den Bedingungen des Zertifizierungsschemas

Geprüfte Kompetenzen

- Review von generiertem Code — Fehlerklassen, Reviewreihenfolge, Diffgrößen
- Lizenzfragen bei generiertem Code — Eingabe, Ergebnis, fremde Rechte, Abhängigkeiten
- Sicherheitsschwächen in Vorschlägen — sechs Muster, Autorisierung, Grenzen der Werkzeuge
- Geheimnisse und Datenflüsse — Abflusswege, Vorfallreihenfolge, Vorkehrungen
- Testabdeckung als Sicherheitsnetz — Annahmefälle, Testgruppen, Charakterisierungstests

Zulassung

Die Teilnahme an diesem Kurs ist **keine** Voraussetzung für die Prüfung. Neben der Kursvorbereitung stehen der direkte Prüfungsweg (Exam-Only) und der Nachweis gleichwertiger Vorkenntnisse offen; die Kriterien sind veröffentlicht. Auch das Foundation-Zertifikat kann durch gleichwertige Nachweise ersetzt werden.

Wie es weitergeht

Wer Systeme aus mehreren Modellinteraktionen bauen will, findet die Fortsetzung in *AI Agent & Automation Architect* (AA). Wer die einzelne Interaktion zuverlässiger machen will, in *AI Prompt Engineer Specialist* (PE). Wer Modul 3 vertiefen möchte, in *AI Security & Red Team Specialist* (SR).

CertifAI ist ein unabhängiger Anbieter. Dieses Zertifikat ist ein Industriezertifikat und keine staatlich anerkannte Qualifikation. Es wird von CertifAI ausgestellt und ist nicht behördlich zugelassen.

Teilnehmerhandbuch AI-Assisted Software Engineer (SE) · Version 1.0 · Ausgabe DE · Rechtsstand August 2026