PRAXIS-GUIDE · CCD AKADEMIE

Warum KI dein Legacy-Projekt nicht rettet

Die 5 stillen Fehler im Umgang mit Copilot, Claude & Co. — und warum dein Entwickler-Team trotz KI nicht schneller liefert.

Ein Praxis-Guide für Entwicklungsleiter, CTOs und Team Leads

Stefan Lieser · CCD Akademie GmbH

"Wir setzen jetzt auf KI" — und dann?

Zwei Jahre nach der KI-Welle in der Softwareentwicklung lohnt sich ein nüchterner Blick auf die Teams, mit denen ich täglich arbeite:

- ✗ Releases sind nicht häufiger.
- ✗ Production-Bugs sind nicht weniger.
- ✗ Der Legacy-Code wird immer noch gemieden.
- ✓ Aber jeder Entwickler hat eine Copilot-Lizenz.

Wenn dir das bekannt vorkommt: Das liegt selten an der KI. Es liegt an der Codebasis, auf die sie trifft — und an der Art, wie dein Team sie nutzt.

In diesem Guide benenne ich 5 Fehler, die ich in Trainings und Code-Reviews bei mittelständischen Software-Teams immer wieder sehe. Pro Fehler erhältst du eine Diagnose-Frage für dein Team und einen sofort umsetzbaren ersten Schritt.

Was du **nicht** bekommst: das vollständige Rezept. Die Tipps sind Pflaster. Das tiefere Problem — warum dein Team überhaupt in dieser Lage ist — adressieren wir am Ende.

ÜBER MICH

Warum erzähle ich dir das überhaupt?

Stefan Lieser

Seit 2008 Clean Code Trainer · Mitbegründer clean-code-developer.de · Gründer der CCD Akademie GmbH

Seit 2008 arbeite ich mit Entwicklungsteams mittelständischer Software-Unternehmen — in Trainings, Code-Reviews und als externer Berater für Entwicklungsleiter und CTOs. Mein Fokus: die Clean Code Developer Prinzipien und Praktiken. Genau die Themen, die jetzt durch KI gleichzeitig wichtiger und gefährdeter werden — saubere Architektur, Tests, Clean Code.

In den letzten zwei Jahren habe ich zahlreiche Teams begleitet, die KI in ihren Entwicklungsalltag integriert haben. Die 5 Fehler in diesem Guide sind keine Theorie. Ich sehe sie im echten Code echter Projekte — und ich habe gesehen, was passiert, wenn man sie ignoriert.

17+	100+	1
Jahre Trainings mit Entwicklungsteams	begleitete Teams in mittelständischen Firmen	Fokus: Clean Code in der KI-Ära

TEILNEHMER-FEEDBACK

Was Teilnehmer nach dem Training sagen

> *Nach dem Training haben wir endlich verstanden, warum unser Copilot-Code immer mehr divergierte. Heute legen wir vor jedem Modul eine kurze Architektur-Skizze ab — die KI liefert seitdem deutlich konsistentere Vorschläge.*
>
> **— Entwicklungsleiter, mittelständisches Software-Haus**

> *Stefans Hinweis, dass KI-Tests die KI-Hausaufgabe prüfen, hat bei uns gesessen. Wir haben unsere PR-Regeln entsprechend angepasst — schon nach einem Quartal deutlich weniger Production-Bugs in den betroffenen Modulen.*
>
> **— CTO, B2B-Softwarehersteller**

> *Wir wollten ein altes Modul 'einfach mit KI' modernisieren. Stefan hat uns auf Charakterisierungs-Tests gebracht — ohne die hätten wir bestehende Fehler einfach mit refactored.*
>
> **— Team Lead, FinTech**

→ Los geht's. Hier kommen die 5 Fehler.

FEHLER #1

Du lässt KI auf schlechten Code los

⚠ DAS SYMPTOM	Copilot schlägt etwas vor, das "irgendwie passt". Der Entwickler übernimmt es. Zwei Sprints später ist die Klasse 800 Zeilen lang, niemand versteht mehr, was sie tut — und der nächste Bug kostet drei Tage.

WARUM DAS PASSIERT

KI lernt aus Mustern. Sie spiegelt die Patterns deines Codes zurück. Sind die Patterns sauber, beschleunigt KI dich. Sind sie schlecht, **multipliziert** KI die Probleme — schneller und in mehr Dateien als jeder Junior es könnte.

KI hat kein Architekturverständnis. Sie hat Statistik.

◉ BEISPIEL AUS DEM TRAINING	*Ein Team mit 7 Entwicklern setzt Copilot seit 9 Monaten ein, ohne seine Architektur zu dokumentieren. Im Code-Review finden wir die gleiche Validierungslogik in 6 verschiedenen Klassen — alle leicht unterschiedlich, alle KI-generiert. Vor Copilot existierte sie an einer Stelle.*

? DIAGNOSE-FRAGE	Auf welcher Grundlage entscheidet ein Entwickler, ob ein KI-Vorschlag zur Architektur passt? *Wenn die Antwort sinngemäß "Bauchgefühl" lautet, hast du das Problem.*

✓ ERSTER SCHRITT	Pro Modul eine einseitige Architektur-Skizze: Welche Verantwortung hat dieses Modul, welche nicht. Diese Skizze wird Teil des KI-Kontextes (siehe Fehler #3).

FEHLER #2

KI ohne Tests ist eine Wette

⚠ DAS SYMPTOM

KI-generierter Code geht durch den Review, wird gemerged — drei Wochen später meldet der Support einen Bug, den niemand erklären kann.

WARUM DAS PASSIERT

Tests sind das einzige Sicherheitsnetz, das prüft, ob der Code wirklich das tut, was die KI behauptet. Ohne Tests verlässt sich dein Team auf das Bauchgefühl des Reviewers — und Bauchgefühl skaliert nicht mit der Code-Menge, die KI produziert.

Mit KI schreibt dein Team plötzlich dreimal so viel Code. Aber niemand schreibt dreimal so viele Tests.

◉ BEISPIEL AUS DEM TRAINING

In einem Team mit Copilot-Lizenz für alle wächst die Code-Basis innerhalb von sechs Monaten um 60 %. Die Testabdeckung sinkt im selben Zeitraum von 71 % auf 48 % — neuer Code wird gemerged, ohne dass jemand Tests nachzieht. Im selben Quartal: drei Production-Bugs in zwei Modulen, die vorher als stabil galten.

? DIAGNOSE-FRAGE

Lass dir die Testabdeckung der drei Module zeigen, in denen Copilot am häufigsten arbeitet. Vergleiche mit dem Stand vor 12 Monaten.

Wenn die Abdeckung gleich geblieben oder gesunken ist, baut ihr gerade Risiko schneller auf, als ihr es absichern könnt.

✓ ERSTER SCHRITT

PR-Regel einführen: KI-generierter Code wird nur gemerged, wenn er von Tests abgedeckt ist, die nicht ausschließlich von der KI geschrieben wurden. Sonst prüft die KI ihre eigene Hausaufgabe.

FEHLER #3

Deine Entwickler nutzen KI wie Stack Overflow — ohne Kontext

⚠ DAS SYMPTOM

Die Lösungen, die deine Entwickler aus der KI holen, funktionieren isoliert. Im Zusammenspiel mit dem Rest des Systems machen sie Ärger: falsche Abstraktionsebene, falsches Pattern, falscher Stil.

WARUM DAS PASSIERT

Viele Entwickler stellen der KI isolierte Fragen ("Wie sortiere ich diese Liste?"). Sie geben keinen Kontext mit: nicht die Domain, nicht die Architektur, nicht die Konventionen des Projekts. Die KI antwortet generisch — und generischer Code passt selten in spezifische Systeme.

◉ BEISPIEL AUS DEM TRAINING

In einem Workshop lasse ich Entwickler dieselbe Aufgabe einmal ohne und einmal mit Architektur-Kontext an die KI stellen. Ergebnis ohne Kontext: ein generischer Service mit eigenen DTOs, eigenem Exception-Handling, eigenem Naming. Mit Kontext: ein Service, der sich sauber in die bestehende Layer-Architektur einfügt. Gleiches Modell, gleicher Entwickler — zehn Minuten Unterschied im Aufwand, ein Sprint Unterschied in der Folge.

? DIAGNOSE-FRAGE

Frage drei verschiedene Entwickler: "Welchen Kontext gibst du der KI, bevor du eine Frage stellst?"

Wenn die Antworten stark auseinandergehen, fehlt euch ein Standard. Wenn alle "gar keinen" sagen, fehlt euch mehr als das.

✓ ERSTER SCHRITT

Ein Prompt-Template im Repository ablegen: Architektur-Skizze + Coding-Conventions + Modul-Verantwortung. Jeder KI-Aufruf nutzt diesen Vorspann.

FEHLER #4

Junioren produzieren Code, den sie nicht erklären können

⚠ **DAS SYMPTOM**

PR-Reviews dauern länger, nicht kürzer. In Diskussionen kann der Autor nicht erklären, warum sein Code so aussieht — "das hat Copilot vorgeschlagen, ich hab's eingebaut".

WARUM DAS PASSIERT

KI ersetzt den Lerneffekt, statt ihn zu beschleunigen. Vor zwei Jahren mussten Junioren ein Problem **verstehen**, um es zu lösen. Heute kopieren sie eine funktionierende Lösung, ohne den Weg dorthin gegangen zu sein.

Das Resultat: Code, den niemand erklären kann — also auch niemand in 6 Monaten warten kann. Dein Bus-Factor sinkt. Dein Onboarding-Aufwand steigt.

◉ **BEISPIEL AUS DEM TRAINING**

Ein Junior öffnet einen PR mit einer eleganten LINQ-Kette über vier GroupBy-Stufen. Im Review fragt der Senior nach der Gruppierungslogik — der Junior weiß nicht, was sie tut. Er hat den Vorschlag von Copilot übernommen, weil die Tests grün waren. Drei Wochen später ein Production-Bug genau in dieser Stelle. Niemand im Team kann ihn zügig debuggen, weil niemand das Konstrukt versteht.

? **DIAGNOSE-FRAGE**

Im nächsten Code-Review: Bitte den Autor, eine zentrale Funktion ohne IDE, am Whiteboard, in eigenen Worten zu erklären.

Wenn er stockt, weißt du Bescheid.

✓ **ERSTER SCHRITT**

"Erklär-Regel" einführen: Wer KI-Output mergen will, muss ihn vorher dem Reviewer in eigenen Worten erklären. Wer nicht erklären kann, mergt nicht.

FEHLER #5

Du erwartest, dass KI dein Legacy-Problem löst — sie macht es schlimmer

⚠ DAS SYMPTOM

"Wir refactoren das später mit KI" ist zur Strategie geworden. In der Zwischenzeit wächst der Legacy-Code weiter, weil niemand sich traut, ihn anzufassen — schließlich kommt ja bald das Wunder.

WARUM DAS PASSIERT

KI kann nur sicher refactoren, wenn zwei Dinge da sind: **eine klare Spezifikation** (was soll der Code tun?) und **Tests** (woher wissen wir, dass er es noch tut?). Genau diese beiden Dinge fehlen in Legacy-Code per Definition.

Ohne Spec und Tests refactored die KI nicht — sie produziert einen neuen Bug auf einer anderen Ebene.

◉ BEISPIEL AUS DEM TRAINING

Ein Kunde will ein 15 Jahre altes Abrechnungsmodul mit KI modernisieren. Wir starten mit einer Bestandsaufnahme: keine Tests, keine Spec, drei leicht unterschiedliche Versionen derselben Berechnung im Code. Nach zwei Wochen Charakterisierungs-Tests stellt sich heraus, dass zwei dieser drei Berechnungen falsch sind — seit Jahren. Ohne diese Tests hätte die KI den Fehler einfach mit refactored.

? DIAGNOSE-FRAGE

Hat dein Team in den letzten 6 Monaten ein Modul angefasst, das vorher als "zu komplex zum Anfassen" galt?

Wenn nein: KI hat das Problem nicht gelöst. Sie hat es verschoben.

✓ ERSTER SCHRITT

Vor dem nächsten "wir nehmen dafür KI"-Versuch: Charakterisierungs-Tests schreiben — Tests, die das aktuelle Verhalten festhalten, egal ob richtig oder falsch. Erst dann ist KI ein Beschleuniger statt ein Risiko.

DAS EIGENTLICHE THEMA

Die 5 Tipps sind Pflaster. Das eigentliche Problem liegt tiefer.

Wenn du den Guide bis hier gelesen hast, ist dir wahrscheinlich aufgefallen: Alle 5 Fehler haben eine gemeinsame Wurzel.

Es geht nicht um KI. Es geht um die Code-Basis, auf die KI trifft.

Solange dein Team ohne saubere Architektur, ohne Tests und ohne Clean Code Prinzipien arbeitet, wird **jede** neue Technologie — KI, neues Framework, neue Sprache — die bestehenden Probleme nur schneller sichtbar machen.

Die gute Nachricht: Das lässt sich ändern. Ich arbeite seit 2008 mit Entwicklungsteams in genau dieser Situation und habe in der CCD Akademie ein Trainingsprogramm aufgebaut, das exakt diese Lücke schließt — nicht in der Theorie, sondern direkt am Code deines Teams.

— • —

Wenn du wissen willst, wo dein Team gerade steht und welcher Schritt der richtige nächste ist, lass uns 30 Minuten sprechen.

Kein Verkaufsgespräch. Eine Standortbestimmung. Du beschreibst kurz, was bei euch passiert — ich sage dir ehrlich, ob ein Training euch helfen würde oder nicht.

SO GEHT'S WEITER

→ Buche jetzt einen Termin mit mir!

Jetzt einen Termin buchen

https://cal.com/stefan-lieser/30min

Telefon · 0151/42073487 E-Mail · stefan.lieser@ccd-akademie.de

Stefan Lieser · CCD Akademie GmbH