



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Willkommen

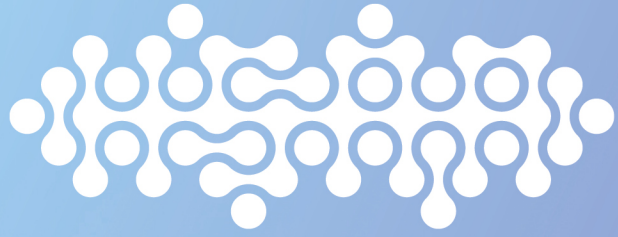
VON GREENFIELD ZU
BROWNFIELD

Heutige Agenda

Sitzung 1 (heute):

0. **Workflow-Auffrischung** — Git, PRs, Hooks, KI-Werkzeuge
1. **Den Kreis schließen** — Stack Overflow Challenge Runde 2 Feedback
2. **Die Brownfield-Realität** — was sich ändert, wenn der Code bereits existiert
3. **Einstieg in Locust** — unsere reale Codebasis + Live-Demos
4. **Projekt-Kontextdateien** — CLAUDE.md, copilot-instructions.md in Aktion
5. **Erste Schritte** — Klonen, Einrichten, erste Erkundung

Sitzung 2 (nächste): Erfolg messen + 60 Min. praktische Übungen



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Auffrischung des professionellen Workflows

0

Das Feature-Branch-Modell

Goldene Regel: Niemals direkt auf main pushen.

Jede Änderung lebt auf ihrem eigenen Branch. Der `main` - (oder `master` -) Branch ist immer in einem deploybaren Zustand.



- Mehrere Entwickler:innen arbeiten parallel, ohne sich gegenseitig zu behindern
- Änderungen werden geprüft, bevor sie main erreichen
- Fehlerhafter Code bleibt isoliert auf dem Feature-Branch

Vertiefung in [Software Engineering Vorlesung 3](#) — [Gesamtkurs](#)

Der Workflow in 6 Schritten

branch → commit → push → PR → review → merge

1. **Branch** — `git checkout -b feature/my-change`
2. **Commit** — Kleine, fokussierte Änderungen mit klaren Messages (*Hooks laufen hier!*)
3. **Push** — `git push -u origin feature/my-change`
4. **Pull Request** — Auf GitHub öffnen, Was und Warum beschreiben
5. **Review** — Teammitglied prüft (*CI läuft hier!*), fordert Änderungen an oder genehmigt
6. **Merge** — PR wird in main gemergt, Branch gelöscht

Die Schritte 2-3 wiederholen sich — committen und pushen Sie fortlaufend, nicht erst am Ende.

Wir behandeln **Hooks** und **CI** auf den nächsten Folien im Detail.

Namenskonventionen für Branches

Präfix	Anwendungsfall	Beispiel
feature/	Neue Funktionalität	feature/team3-stats-coverage
fix/	Fehlerbehebungen	fix/percentile-overflow
refactor/	Code-Verbesserungen	refactor/extract-logging-helpers

Regeln:

- Kleinbuchstaben, Bindestriche zwischen Wörtern
- Aussagekräftig — ein Teammitglied sollte den Zweck am Namen erkennen
- Kurz — niemand möchte feature/add-comprehensive-test-coverage-for-the-stats-module tippen

Was ist ein Pull Request?

Ein PR ist ein **Kommunikationsartefakt**, nicht nur eine Code-Einreichung.

Jeder PR sollte erklären:

- **Was** sich geändert hat — Zusammenfassung des Diffs
- **Warum** es sich geändert hat — Motivation, Issue-Referenz
- **Wie man testet** — Schritte, denen ein Reviewer folgen kann

KI kann PR-Beschreibungen entwerfen — aber prüfen Sie, ob sie beschreiben, was der Code **tut**, nicht was er **tun sollte**.

Branch Protection

Unser Locust-Repository hat Branch Protection auf `master` :

- **Erfordert 1 genehmigendes Review** vor dem Merge
- **Keine direkten Pushes** — jede Änderung geht über einen PR
- Admins können dies umgehen (nur Dozent)

Wenn `git push origin master` abgelehnt wird — **das ist Absicht**, kein Fehler.

Dies simuliert reale Open-Source-Projekte, in denen Maintainer jeden Beitrag prüfen.

Pre-Commit Hooks

Automatisierte Qualitätstore, die vor jedem `git commit` ausgeführt werden.

Das Locust-Repository hat dies in `.pre-commit-config.yaml`:

```
repos:
- repo: https://github.com/astral-sh/ruff-pre-commit
  hooks:
    - id: ruff          # Linter
      args: [--fix]
    - id: ruff-format   # Formatter
```

Wenn Ihr Code PEP-8-Verstöße hat → **Commit abgelehnt**. Beheben Sie die Verstöße, dann committen Sie erneut.

Das System **funktioniert korrekt** — es ist kein Problem.

CI/CD in 30 Sekunden

CI (Continuous Integration): Führt Tests + Linting automatisch bei jedem Push/PR aus.

In einem professionellen Projekt: Nichts wird gemergt, ohne dass CI bestanden hat.

In unserem Lehr-Repository: CI ist **deaktiviert** (GitHub Actions-Kontingent).

Das bedeutet: **Sie sind die CI.**

```
# Tests vor jedem Push ausführen
uv run pytest locust/test/test_stats.py -x -q

# Linter ausführen (oder auf Pre-Commit Hooks vertrauen)
uv run ruff check . && uv run ruff format --check
```

pytest-Ausgaben lesen

```
locust/test/test_stats.py::TestRequestStats::test_log ..... PASSED
locust/test/test_stats.py::TestRequestStats::test_extend .. PASSED
locust/test/test_stats.py::TestStatsError::test_parse ... FAILED
```

- **PASSED** — Test hat das erwartete Verhalten verifiziert
- **FAILED** — Assertion hat nicht gehalten. Lesen Sie die Ausgabe, um Soll vs. Ist zu sehen.
- **ERROR** — Test konnte nicht ausgeführt werden (Import-Fehler, fehlende Fixture, etc.)

Nützliche Flags: `-x` stoppt beim ersten Fehler, `-q` reduziert Ausgaben, `-v` zeigt Details

Ihre KI-Werkzeuge heute

GitHub Copilot

- Chat-Panel + Inline-Vervollständigungen
- Mehrere Modelle (GPT-4o, Claude, Gemini)
- Liest `.github/copilot-instructions.md`
- Dieses Werkzeug kennen Sie bereits

Claude Code *(neu)*

- Terminal-Panel in VS Code (oder eigenständige CLI)
- Claude-Modelle (bis zu 1M Token Kontextfenster)
- Liest `CLAUDE.md` aus dem Projektstammverzeichnis
- Installation: `npm install -g @anthropic-ai/claude-code`

Beide können: Dateien lesen/schreiben, Befehle ausführen, Tests starten, Commits erstellen im **Agent-Modus**.

Interaktionsmodi

Modus	Was er tut	Wann einsetzen
Ask	Beantwortet Fragen, keine Dateiänderungen	Code verstehen, Architektur erkunden
Agent / Edit	Liest, schreibt, führt Befehle aus	Tests schreiben, Refactoring, Bugs beheben
Plan	Entwirft einen Ansatz, keine Änderungen bis zur Freigabe	Komplexe mehrstufige Aufgaben, Architekturentscheidungen

Für die heutigen Übungen: Agent-Modus ist Ihr primärer Modus.

Die KI liest Ihre `CLAUDE.md` (oder `.github/copilot-instructions.md` für Copilot), versteht das Projekt und kann Tests ausführen, um ihre eigene Arbeit zu verifizieren.

Weiterführende Ressourcen

Git & Workflow:

[Feature-Branch-Entwicklung](#) — SE-Kursvorlesung zu Git-Grundlagen, PRs, Branch Protection, CI/CD

KI-Werkzeuge:

[Vorlesung 2 Folien](#) — Copilot vs Claude Code Vergleich, Einrichtung, Berechtigungen

Wenn Sie den SE-Kurs belegt haben, war dieses Kapitel Wiederholung. Falls nicht — setzen Sie Lesezeichen für diese Links.

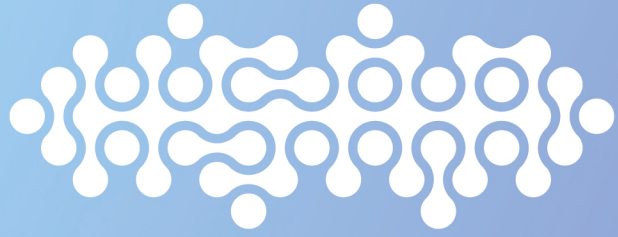
Kurze Umfrage

Bewerten Sie Ihren Komfort mit diesem Workflow (1-5):

1. "Was ist ein Branch?"
2. "Ich habe von PRs gehört, aber noch nie einen erstellt"
3. "Ich kann den grundlegenden Workflow, brauche aber Erinnerungshilfen"
4. "Vertraut — ich nutze das regelmäßig"
5. "Ich könnte das unterrichten"

Heben Sie die Hand für Ihre Zahl.

Das zeigt uns, wie viel Unterstützung Sie bei den Übungen benötigen.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Den Kreis schließen

1

Feedback-Quiz (5 Min.)

Öffnen Sie den Google-Forms-Link: forms.gle/2ZZJsCvr3dgfAKsm9

Zentrale Fragen:

- Welchen Ansatz haben Sie verwendet? Strikt Ansatz 3 oder gemischt?
- Hat es sich verbessert? (Skala 1-5)
- Iterationszahl: Wie viele Prompts? Vergleich mit Runde 1.
- KI-Tagebuch: Overhead oder nützlich?
- **Zuversicht bei unbekanntem Code?** ← Merken Sie sich diese Antwort.

Was wir erwarten

Iterationszahlen sinken. Ansatz 3 verlagert das Denken nach vorne. Jeder Prompt ist eingegrenzt und spezifisch.

Zuversicht steigt. Nicht weil die KI besser ist — sondern weil der *Prozess* Prüfpunkte bietet.

Tagebuch-Reibung hat sich verlagert. Den Eintrag zu schreiben geht jetzt schnell. Die eigentliche Reibung: **Commit-Disziplin** — ein Commit pro KI-Interaktion, strukturierte Messages. Das ist kein Overhead — das ist Kommunikation.

Gemischte Ansätze sind in Ordnung. Der Punkt: *Wie* Sie mit der KI interagieren, beeinflusst was Sie erhalten.

Eine kleine Ergänzung zum Referenz-Repository

Öffnen Sie das Referenz-Repository: github.com/l3gj0n/so-challenge. Schauen Sie ins Stammverzeichnis.

Zwei neue Dateien:

- **CLAUDE.md** — 45 Zeilen KI-Kontext
- **.github/copilot-instructions.md** — derselbe Inhalt für Copilot

Diese definieren auch das **Tagebucheintragsformat** — so müssen Sie die Struktur nie in Ihren Commit-Messages ausformulieren. Sagen Sie einfach "schreibe einen Tagebucheintrag" und die KI referenziert ihre eigene Kontextdatei.

Für ein 250-Zeilen-Projekt: ~500 Token Kontext. **Später heute:** eine CLAUDE.md für ein 10.000-Zeilen-Projekt. Dasselbe Konzept. **Völlig andere Dimension.**

Die Frage, die diese Vorlesung einleitet

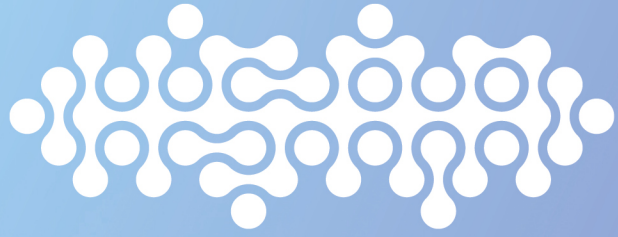
Schauen Sie sich die Feedback-Ergebnisse an für:

"Wie zuversichtlich sind Sie, KI-Werkzeuge in einer Codebasis einzusetzen, die Sie NICHT geschrieben haben?"

Wenn das Muster zutrifft: **niedriger als Ihre allgemeine Zuversicht.**

Das ist eine ehrliche, zutreffende Einschätzung. Alles bisher war **Greenfield** — Sie haben von einem leeren Verzeichnis aus gestartet, Sie haben jede Entscheidung getroffen.

Aber so sehen die meisten Software-Jobs nicht aus.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Die Brownfield-Realität

2

Wie Entwickler:innen tatsächlich ihre Zeit verbringen

60-80 % des professionellen Software-Entwicklungsaufwands fließen in die **Wartung und Weiterentwicklung bestehender Systeme**.

70 % der Entwicklungszeit entfallen auf **Programmverstehen** — Code lesen, Aufrufhierarchien navigieren, mentale Modelle aufbauen.

(Minelli, Mocci & Lanza 2015; LaToza, Venolia & DeLine 2006)

Die primäre Entwicklertätigkeit ist nicht das Schreiben — es ist das **Lesen**.

Greenfield vs Brownfield: 5 zentrale Unterschiede

Greenfield (SO Challenge)

- Sie kontrollieren den Kontext
- Tests sind aspirativ
- Sie schreiben mehr als Sie lesen
- "Fertig" = es funktioniert
- Sie bestimmen die Konventionen

Brownfield (Locust)

- Kontext ist massiv, KI braucht Hilfe
- Tests sind ein Sicherheitsvertrag
- 70 % Lesen, 30 % Schreiben
- "Fertig" = Checkliste mit 7 Punkten
- Konventionen sind vorgegeben

KI im Brownfield: Nützlicher UND gefährlicher

Wo KI glänzt:

- Code-Erklärung ("Was tut diese Funktion?")
- Testgenerierung ("Schreibe Tests für dieses Modul")
- Refactoring ("Teile diese 99-Zeilen-Methode auf")

Wo KI Schwierigkeiten hat:

- Architekturentscheidungen (kennt das *Warum* nicht)
- Ungeschriebene Konventionen (kann Implizites nicht befolgen)
- Scope-Kontrolle (schlägt größere Änderungen vor als nötig)

Das Muster: KI-Schwächen sind Symptome von *implizitem Wissen*. Machen Sie es explizit → die KI befolgt es.

Implizites Wissen explizit machen

Sobald eine Konvention aufgeschrieben ist — in einer `CLAUDE.md` (oder `.github/copilot-instructions.md`), einer Definition of Done, einem Architekturdokument — wird die KI deutlich besser darin, ihr zu folgen.

"Verwende immer die bestehende `StatsEntry`-Klasse, erstelle nie eine parallele Stats-Struktur."

Das ist genau die Art von Regel, der eine KI konsistent folgen kann, über Hunderte von Zeilen, ohne es zu vergessen.

Aber "besser" ist nicht "perfekt." Mehr dazu gleich.

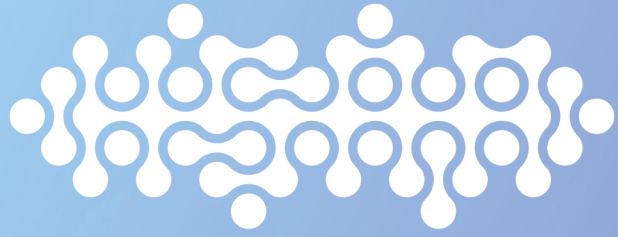
Diskussion

Denken Sie an ein Projekt, an dem Sie gearbeitet haben (Praktikum, Abschlussarbeit, Nebenprojekt oder auch eine Gruppenarbeit):

Welche impliziten Konventionen gab es, die nie dokumentiert wurden?

- Ein Namensmuster, dem alle folgten, das aber niemand dokumentierte?
- Ein Modul, von dem "jeder weiß", dass man es nicht anfassen sollte?
- Ein Testmuster, das überall verwendet wird, aber in keinem Styleguide steht?

2 Minuten: Besprechen Sie es mit Ihrer Sitznachbarin oder Ihrem Sitznachbarn.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Einstieg in das Locust- Ökosystem

3

Was ist Locust?

Ein Python-Lasttest-Framework. ~25.000 GitHub-Sterne, aktive Entwicklung.

- Benutzerverhaltensskripte in reinem Python schreiben
- Simuliert Tausende gleichzeitiger Benutzer (gevent Greenlets)
- Echtzeit-Web-UI unter `localhost:8089`
- Verteilter Modus über mehrere Maschinen

Warum Locust für diesen Kurs? Performance IST das Produkt. Passend dimensionierte Codebasis (~10K Zeilen). Verwendet pytest. Aktiv und real.

Unsere Lehrkopie

github.com/hs-aalen-software-engineering/locust

- **Vollständiger Import** von `locustio/locust` — komplette Commit-Historie (6.000+ Commits)
- **Keine Verbindung** zum Upstream — Ihre PRs bleiben hier
- **Actions deaktiviert** — Sie führen Tests lokal aus
- **Branch Protection** auf `master` — 1 Review erforderlich
- **CLAUDE.md** + **copilot-instructions.md** — KI-Kontext vorkonfiguriert

Dies simuliert Open-Source-Beiträge ohne Risiko.

Verzeichnisstruktur

```
locust/
├─ locust/                # Kern-Python-Paket
│   ├── __init__.py       # Öffentliche API + gevent Monkey-Patching
│   ├── stats.py          # Request-Statistiken (1.250 Zeilen)
│   ├── runners.py        # Local/Master/Worker Runner
│   ├── web.py            # Flask Web-UI + REST API
│   ├── user/             # User-Modell + @task-System
│   ├── contrib/          # Protokoll-Clients (mqtt, postgres...)
│   └─ test/              # Testsuite (~700 Tests)
├─ examples/              # 40+ Beispiel-Locustfiles
├─ CLAUDE.md              # KI-Kontext für Claude Code (615 Zeilen!)
├─ .github/copilot-instructions.md # KI-Kontext für Copilot
└─ .pre-commit-config.yaml # ruff Hooks
```

Wichtige Module

Modul	Zeilen	Funktion
stats.py	1.250	Antwortzeiten, Perzentile, Aggregation
runners.py	1.500+	Benutzer spawnen, verteilter Modus
web.py	800+	Dashboard-UI, REST API
user/users.py	300+	User/HttpUser-Basisklassen
user/task.py	500+	@task-Dekorator, TaskSet
clients.py	550	HttpSession mit Stats-Integration

Tests befinden sich **innerhalb** von `locust/test/` — nicht in einem separaten `tests/`-Verzeichnis.

Live-Demo: Einen Lasttest ausführen

```
# locustfile.py
from locust import HttpUser, task, between

class QuickStartUser(HttpUser):
    wait_time = between(1, 3)
    host = "https://httpbin.org"

    @task
    def hello_world(self):
        self.client.get("/get")
```

```
uv run locust -f locustfile.py
# Öffnen Sie http://localhost:8089
```

Nur `httpbin.org` oder `localhost` als Ziel verwenden. Tests gegen fremde Server können rechtliche Konsequenzen haben.

Probieren wir es live aus. Falls die Weboberfläche nur Rohtext zeigt, muss das React-Frontend gebaut werden — fragen Sie Ihr KI-Tool.

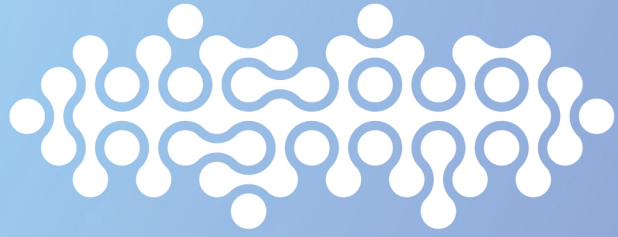
Live-Demo: Die schnelle Test-Teilmenge ausführen

Nicht alle Tests sind schnell. Manche dauern 2+ Minuten. Verwenden Sie die kuratierte Teilmenge:

```
uv run pytest locust/test/test_env.py locust/test/test_tags.py \  
locust/test/test_users.py locust/test/test_locust_class.py \  
locust/test/test_wait_time.py locust/test/test_util.py \  
locust/test/test_html_filename.py -x -q
```

125 Tests, ~15 Sekunden, alle bestanden. Ihre Übungspakete haben noch schnellere Befehle (~8-22s).

Denken Sie daran: **Sie sind die CI.** Führen Sie Tests vor jedem Push aus.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Projekt-Kontextdateien

4

Das Kontextproblem

Sie haben gerade eine **10.000-Zeilen-Codebasis** geöffnet. Sie möchten Ihre KI um Hilfe bitten.

Aber die KI weiß **nichts** über dieses Projekt. Sie weiß nicht:

- Dass `stats.py` die Perzentilberechnung handhabt
- Dass Tests `unittest.TestCase`-Basisklassen verwenden, keine pytest-Fixtures
- Dass Tests in `locust/test/` liegen, nicht in `tests/`
- Dass GitHub Actions deaktiviert sind

Ohne Kontext gibt die KI **generische, oft falsche** Antworten.

Warum Projekt-Kontextdateien wichtig sind

Moderne KI-Modelle sind intelligent genug, um eine Codebasis eigenständig zu erkunden. Aber Kontextdateien geben ihnen einen **Vorsprung**:

- Ohne: die KI liest Dateien, leitet Muster ab, kommt *irgendwann* zum Ziel
- Mit: die KI **weiß sofort** über Konventionen, Testmuster und Fallstricke Bescheid

Der Unterschied ist nicht richtig vs falsch — es ist **schnell und konsistent** vs langsam und ad-hoc. Bei 10K+ Zeilen zählt dieser Vorsprung.

600 Zeilen strukturierter Kontext sparen Minuten bei jeder Interaktion.

Aufbau der Locust CLAUDE.md

Öffnen Sie sie jetzt im Projektstammverzeichnis: `CLAUDE.md` + `.github/copilot-instructions.md`

Abschnitt	Zweck
Zusammenarbeitsprinzipien	Verhaltensanweisungen für die KI
Wichtige Dateien zum Einstieg	Geordneter Onboarding-Leitfaden (10 Dateien)
Testdetails	Basisklassen, Muster, Testbefehle
Was NICHT anfassen	Actions deaktiviert, Build-Artefakte, etc.

Die SO-Challenge-Version: 45 Zeilen. Diese hier: **615 Zeilen**. Dasselbe Konzept, andere Dimension.

Verbreitung in der Praxis

Dies ist kein Lehrkonstrukt. Große Open-Source-Projekte tun dies:

Projekt	Sterne	CLAUDE.md
React	244K	Minimal — Monorepo-Aufteilungshinweis
Deno	106K	Detailliert — Architektur, Debugging, Testen
Next.js	138K	Symmlink zu AGENTS.md
Zed	78K	Coding-Stilregeln ("keine Abkürzungen")

Projekt	Sterne	copilot-instructions.md
VS Code	165K	Architektur, Coding-Richtlinien, Testen
.NET Runtime	15K	Build-System, Konventionen, Beitragsregeln

Dasselbe Prinzip, verschiedene Werkzeuge. Unser Locust-Repository hat **beides**.

Was CLAUDE.md NICHT garantiert

Ein reales Beispiel aus der Kursvorbereitung:

Wir baten die KI, ein Benchmark-Skript zu generieren. Die Projektanweisungsdatei (`CLAUDE.md` / `copilot-instructions.md`) sagt explizit "ruff vor dem Committen ausführen." Das Ergebnis:

- Benchmark-Code **funktionierte korrekt**
- Hatte aber **PEP-8-Verstöße**, die der Linter erkannt hätte
- Nachbesserung: KI korrigierte den Code, **vergaß aber den Tagebucheintrag**

Projektanweisungsdateien sind Einfluss, nicht Durchsetzung.

Das Drei-Schichten-Modell

Schicht	Was	Wie
Einfluss	CLAUDE.md / copilot-instructions	Probabilistisch — KI befolgt es <i>meistens</i>
Durchsetzung	Pre-Commit Hooks, Linter, CI	Deterministisch — ruff besteht oder scheitert
Mensch	Ihre Disziplin, Code Review	Urteilsvermögen — erkennt semantische Probleme

Alle drei Schichten arbeiten zusammen. In unserem Repository ist CI deaktiviert — also sind **Hooks + Sie** das Sicherheitsnetz.

Auch im Zeitalter KI-gestützter Entwicklung bleiben automatisierte Prüfungen unverzichtbar.

Live-Demo: Pre-Commit Hook in Aktion

Code mit einem absichtlichen PEP-8-Verstoß schreiben...

```
git add benchmarks/run_benchmarks.py  
git commit -m "Add benchmark"  
# ruff lehnt den Commit ab!
```

Verstoß beheben → Commit gelingt.

Das ist Ihr Sicherheitsnetz. Die KI wird nicht immer den Styleguide befolgen. Der Hook fängt es ab.

Werkzeug-Äquivalente

Werkzeug	Anweisungsdatei	Speicherort
Claude Code	CLAUDE.md	Projektstammverzeichnis
GitHub Copilot	.github/copilot-instructions.md	.github/ -Verzeichnis
Cursor	.cursorrules	Projektstammverzeichnis

Dasselbe Prinzip: Projektkontext einmal kodieren, in einer Datei, die die KI automatisch liest.

Das Locust-Repository hat sowohl CLAUDE.md als auch copilot-instructions.md.
Welches Werkzeug Sie auch nutzen — Sie profitieren davon.

Ausblick auf Akt II

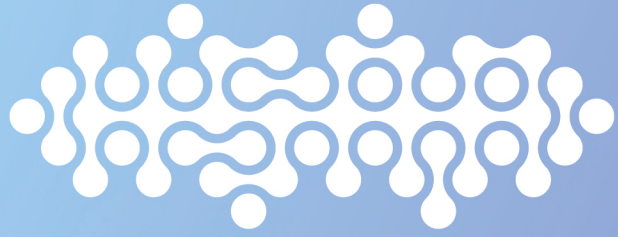
Sie wissen jetzt, dass Projekt-Kontextdateien **funktionieren** — Sie haben das Vorher/Nachher gesehen.

Sie wissen, dass es **nicht perfekt** ist — die Benchmark-Anekdote.

Aber Sie wissen noch nicht **warum**.

In Akt II lernen Sie, wie diese Anweisungen auf Token-Ebene verarbeitet werden — warum sie beeinflussen, aber nicht garantieren, warum längere Kontexte die Leistung verschlechtern, und warum die Reihenfolge der Anweisungen wichtig ist.

Vorerst: Vertrauen, aber verifizieren.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Erste Schritte

5

Setup-Checkliste

```
# 1. Klonen
git clone https://github.com/hs-aalen-software-engineering/locust.git
cd locust

# 2. Abhängigkeiten installieren
uv sync

# 3. Pre-Commit Hooks aktivieren
pip install pre-commit && pre-commit install

# 4. Schnelle Test-Teilmenge ausführen
uv run pytest locust/test/test_env.py locust/test/test_tags.py \
    locust/test/test_users.py locust/test/test_util.py -x -q
```

Wenn alle Tests bestehen: Sie sind in einem bekannt-guten Zustand. **Jetzt wird es praktisch** — folgen Sie der [Locust-Erkundung](#) Übung.

Branch und Tagebuch erstellen

```
# Branch Ihres Teams erstellen  
git checkout -b feature/<team-name>-<task>  
  
# Tagebuch-Unterordner erstellen  
mkdir -p diary/feature/<team-name>-<task>
```

Ihr erster Tagebucheintrag: `001-onboarding.md`

Dokumentieren Sie: Was Sie die KI gefragt haben, was Sie über die Architektur gelernt haben, was die KI richtig vs. falsch gemacht hat.

Committen Sie mit: `git commit -m "[diary] 001 – Onboarding und Erkundung"`

Ihre erste KI-Onboarding-Frage

Öffnen Sie Ihr KI-Werkzeug im Locust-Verzeichnis. Versuchen Sie:

"Was ist die High-Level-Architektur dieses Projekts?"

Vergleichen Sie dann die Antwort mit dem, was `CLAUDE.md` / `copilot-instructions.md` beschreibt.

- Wo fügt die KI Details hinzu, die über die Kontextdatei hinausgehen?
- Wo liegt sie falsch?
- Was müssten Sie wissen, um mit dem Beitragen zu beginnen?

Machen Sie das jetzt. Dies ist der Beginn Ihrer Übung.

Vollständige Übung: [Locust-Erkundung](#)

Nächste Sitzung

Sitzung 2 behandelt:

- **Erfolg messen** — Tests, Coverage, Benchmarks, Definition of Done
- **Übungspakete** — die spezifischen Ziele Ihres Teams
- **60 Minuten praktische Arbeit** — Tests schreiben, Code refactoren, PRs einreichen

Vor der nächsten Sitzung:

- Schließen Sie die [Locust-Erkundung](#) Übung ab
- Ihr Setup muss funktionieren: geklont, Tests bestanden, Pre-Commit Hooks aktiv



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Bis zur nächsten Sitzung

RICHTEN SIE IHR SETUP EIN!