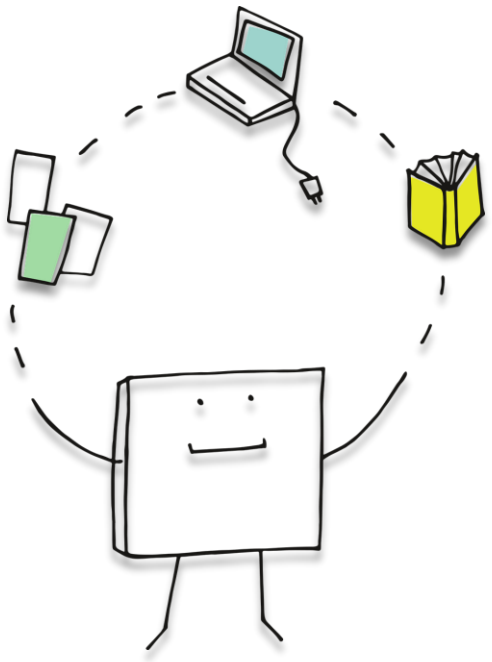
Testing ohne Stress: Wie KI uns den Alltag erleichtern kann

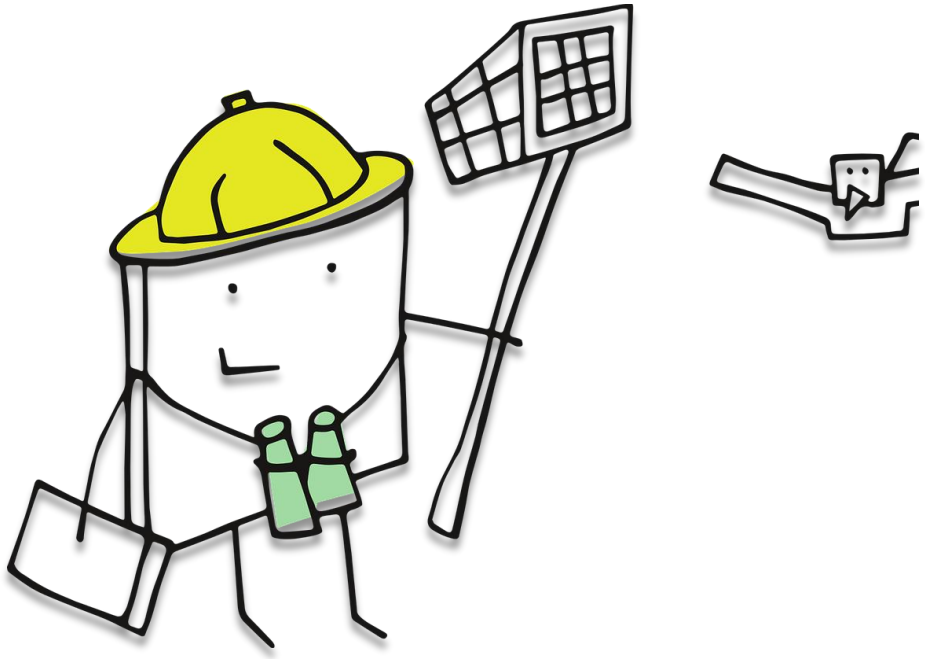
HANNAH EBERT & MILENA FLUCK

Das erwartet euch

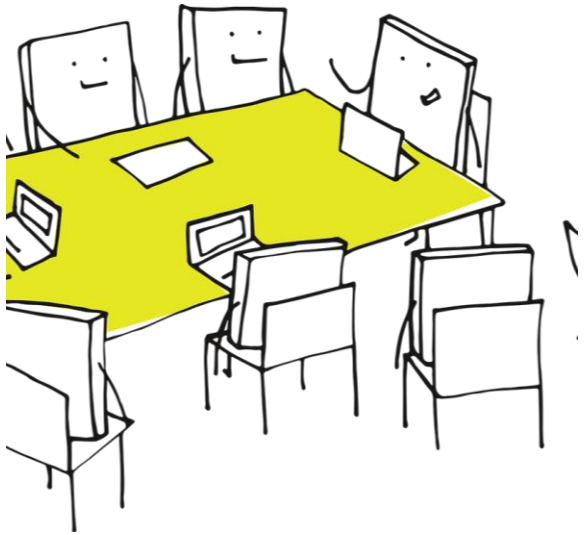
- Unit Testing mit Jest
- Was ist GenAI?
- Was sind Large Language Models?
- KI fürs Unit-Testen
- Praktische Übungen



Ziele



- Überblick gewinnen
- Tools und praktische Tipps sammeln

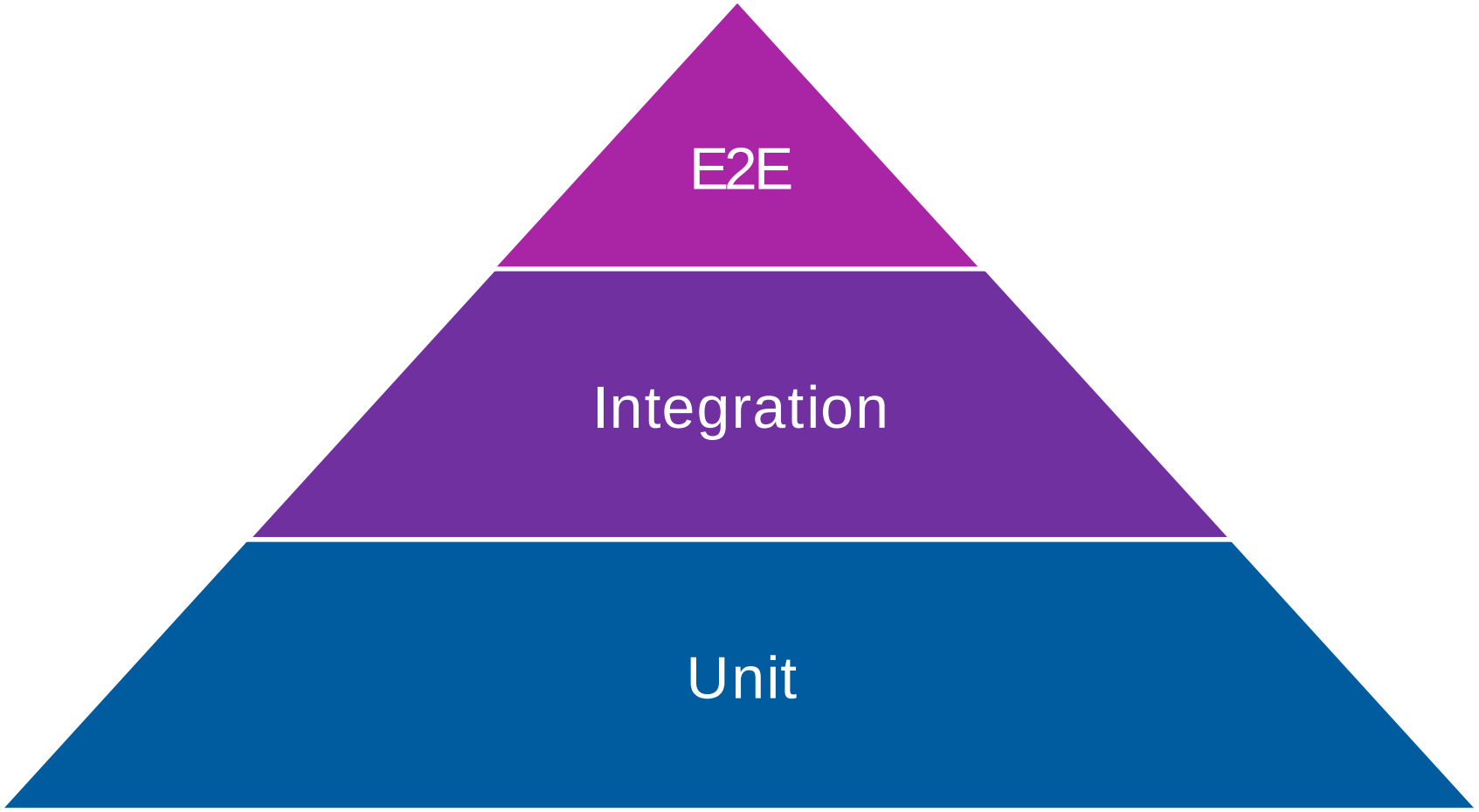


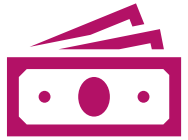
Survey Time

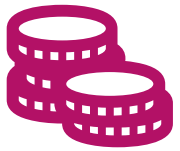
HTTPS://WWW.MENTI.COM/ALGWQGH7O97M

Was ist Unit Testing?

Die Testpyramide







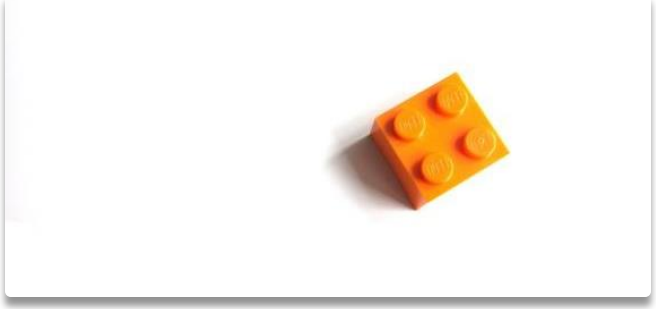
Ein Unit

- Der kleinste Baustein einer Applikation:
  - Funktion
  - Klasse
  - Component

- **Grundannahme:**
  - Wenn alle Units funktionieren, dann funktioniert auch die Applikation.

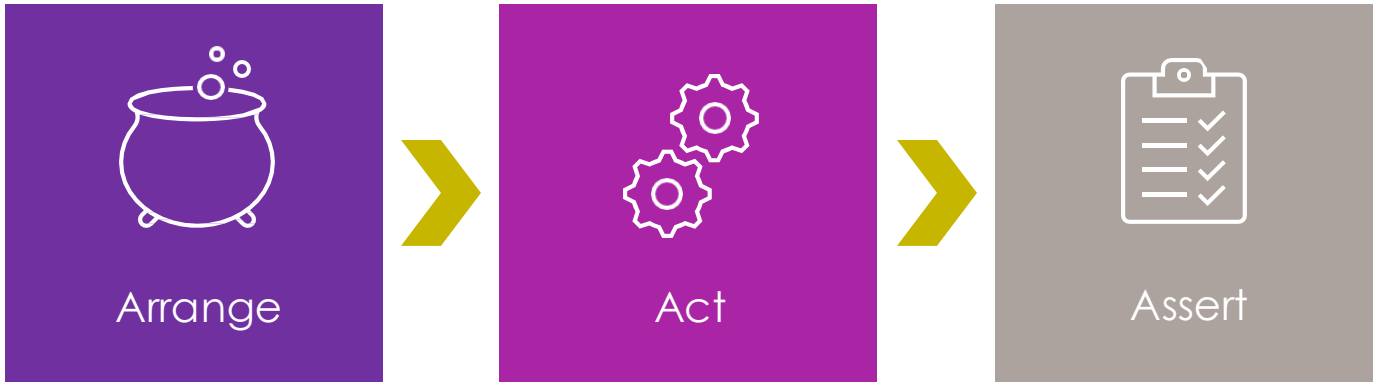
Vorteile von Units

- Geringere Komplexität des Codes führt zu geringerer Komplexität des Tests
- Code lesbarer
- Testcode einfacher zu implementieren
- Testcode schneller automatisiert ausführbar



Triple-A-Prinzip

Arrange | Given
Unter der Bedingung Jurassic Park ist offen und ein T. Rex ist ausgebrochen
Act| When
Führe isEvakuierungErforderlich(x,y) aus
Assert| Then
Dann soll das Ergebnis Ja sein.



Was ist Jest?

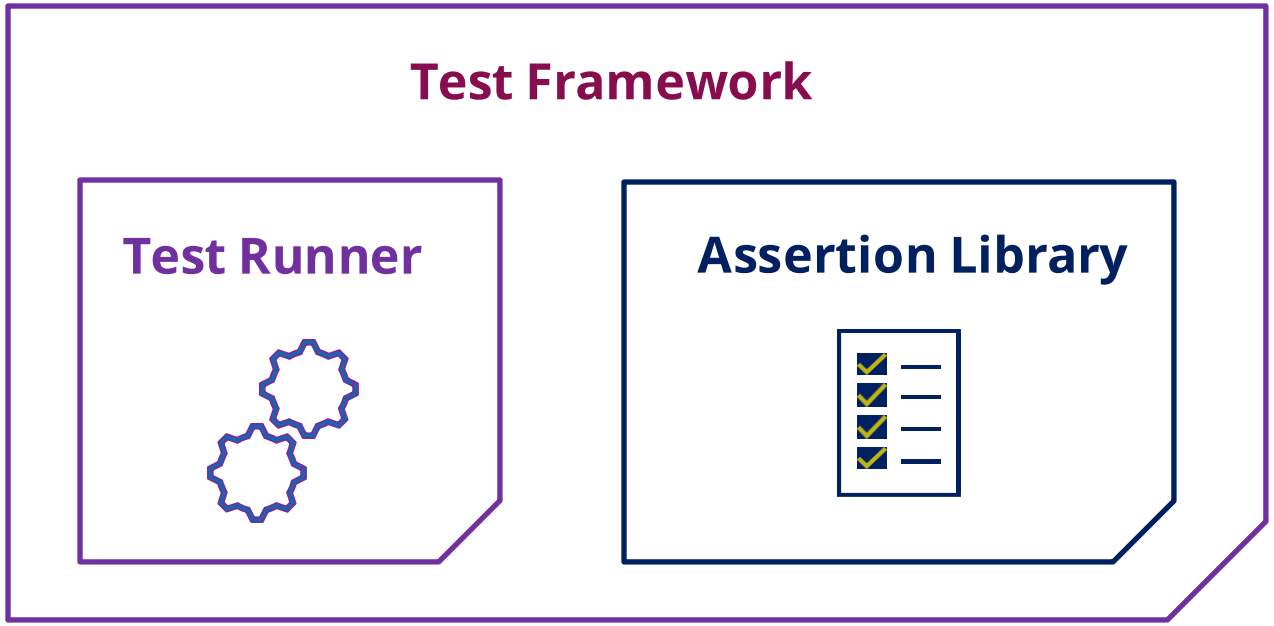
Repository

https://github.com/MilenaFluck/js-days-unit-testing-with-ai

Jest

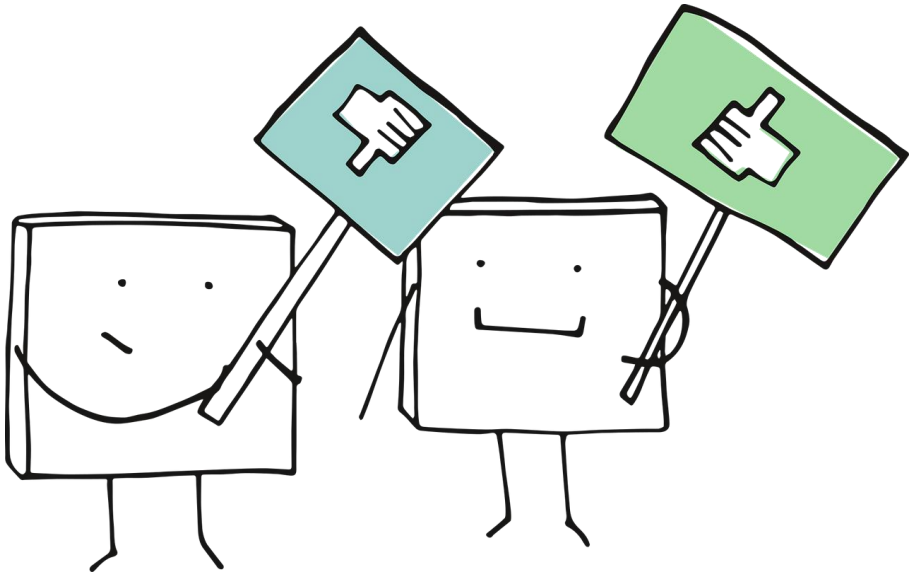
- Ein Javascript Testing Framework
- Entwickelt von Facebook
- Gute Doku

Jest: Batteries included



Erwartungen formulieren: Jest Matcher

- ToBeTruthy
- ToBeNull
- ToStrictEqual
- ToHaveBeenCalled
- ToThrow
- ToContain
- …



Aufbau einer Test-Suite

- Strukturieren Testcode innerhalb einer Datei
  - describe
  - it | test

```
describe('Robot Invasion', () => {
    it('should take place', () => {
        const robotInvasion = true;
        expect(robotInvasion).toBe(true);
    });
});
```

Side Effects

- Problem 1: System beeinflusst den Test durch ungewollte Nebenwirkungen
- Problem 2: Test hat ungewollte Nebenwirkungen auf das System

Stubs, Spies & Mocks

- **Stubs:**
  - gibt immer den statischen Return-Wert zurück
  - wenn uns vollkommen egal ist, wie die Entität sich verhält
- **Spies:**
  - "Wrapper" um Funktionen oder leeren Ersatz für Funktionen
  - erlaubt uns zu verfolgen, ob und wie eine Funktion aufgerufen wurde.
- **Mocks:**
  - Ersatz für eine API
  - Liefert uns stattdessen ein test-spezifisches Verhalten

Coverage

- **Statement Coverage:** Aufgerufene Funktionen (auch als Mock)
- **Branch Coverage:** Ausgeführte Funktionen
- **Functions Coverage (Conditional Coverage):** Ausgeführte Konditionen
- **Lines Coverage:** Abdeckung aller Zeilen, die jemals ausgeführt werden könnten

```
-------------------------------------------------------|---------|----------|---------|---------|-----------------------
File                                                   | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------------------------------------------|---------|----------|---------|---------|-----------------------
All files                                              |   82.77 |    65.45 |      80 |   82.65 |
 app                                                   |     100 |      100 |     100 |     100 |
  app.component.html                                   |     100 |      100 |     100 |     100 |
  app.component.ts                                     |     100 |      100 |     100 |     100 |
 app/chapter-3-writing-good-tests/example-assertions   |     100 |      100 |     100 |     100 |
  example-assertions.component.ts                      |     100 |      100 |     100 |     100 |
 app/chapter-3-writing-good-tests/example-code-clean   |    92.3 |       75 |      75 |    92.3 |
  example-code-clean.ts                                |    92.3 |       75 |      75 |    92.3 | 11
 app/chapter-3-writing-good-tests/example-code-dirty   |      44 |    36.36 |     100 |      44 |
  example-code-dirty.ts                                |      44 |    36.36 |     100 |      44 | 16-21,25-28,32-35,38-41
```

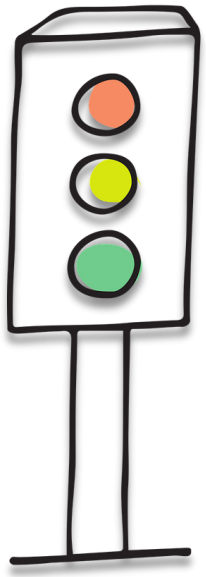
Was sind Grundsätze für "gute" Unit-Tests?

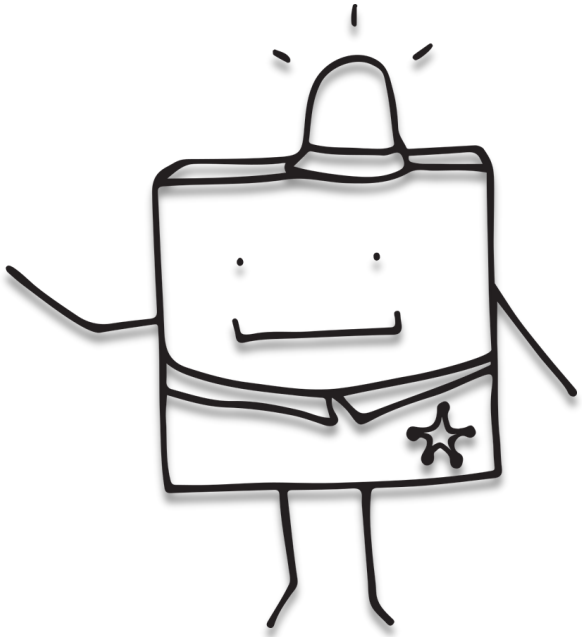
F.I.R.S.T.-Prinzipien

- **Fast:** dauern wenige Millisekunden
- **Independent/Isolated:** unabhängig voneinander, können in beliebiger Reihenfolge ausgeführt werden
- **Repeatable:** werden häufig und in verschiedenen Umgebungen ausgeführt => liefern verlässlich gleiche Ergebnisse
- **Self-Validation:** Ergebnisse werden programmatisch ausgewertet und liefern ein binäres Ergebnis
- **Timely/Thorough:** werden "rechtzeitig" (am besten vor dem Produktivcode) und "gründlich" geschrieben

Tipps für gute Tests

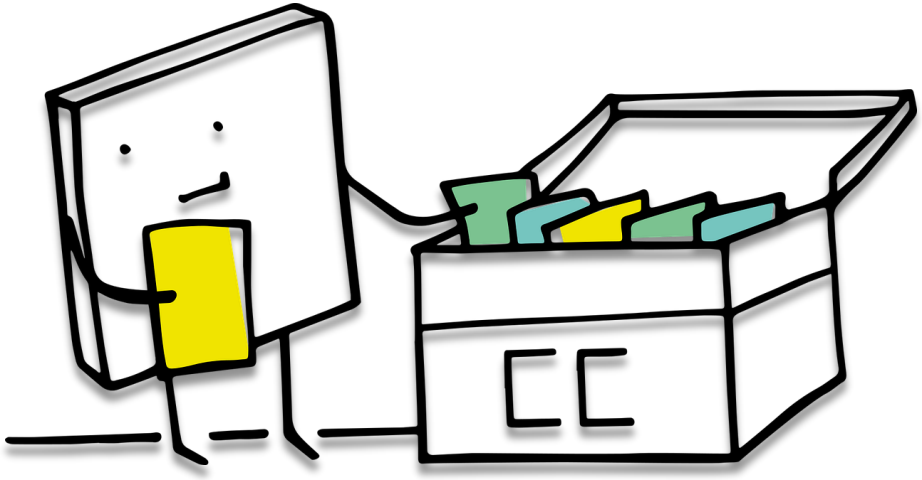
- **Regel 1:** Teste nichts, was du nicht ändern kannst (Third-Party-Code: externe Libs, Framework-Code, Browser APIs etc.)
- **Regel 2:** Halte es so einfach wie möglich.





Tipps für gute Tests

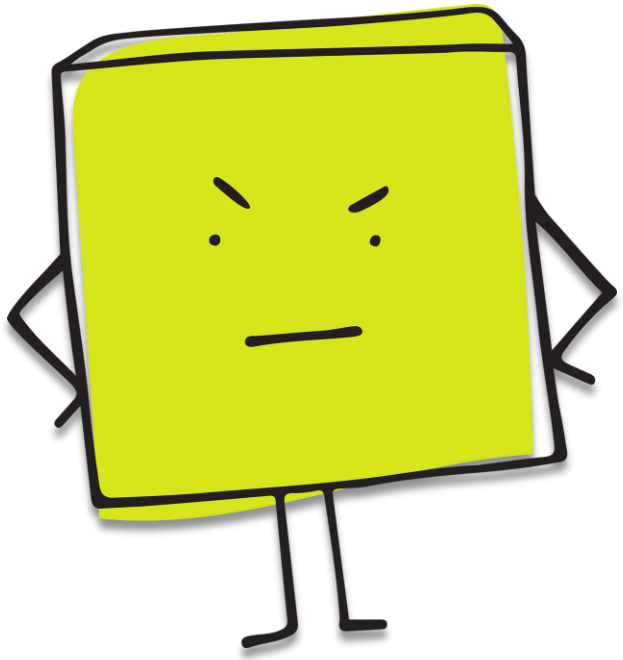
- **Regel 3:** Teste immer nur eine Sache pro Test.
- **Regel 4:** Halte die Anzahl an Assertions gering.



Clean Code = Clean Testing?

- Man sollte Code nicht anpassen müssen, um ihn gut testen zu können
- **Aber:** Man sollte Code so schreiben, dass man ihn leicht testen kann

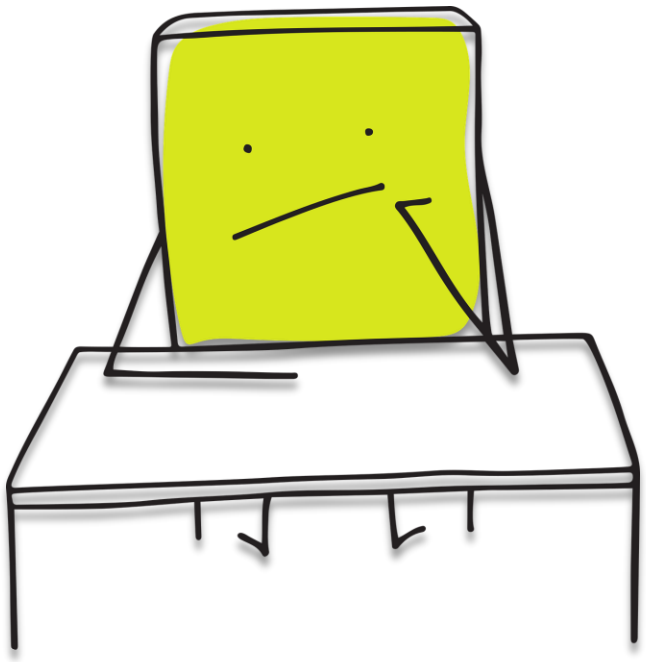
- Unit Tests fordern auch die Qualität des Quellcodes heraus



Was ist GenAI?

Das ist GenAI

- Generative künstliche Intelligenz bezieht sich auf Algorithmen, die auf Basis bestehender Daten neue **Inhalte wie Texte, Bilder oder Musik erzeugen**
- Sie **erkennen Muster** in den Daten und imitieren sie
- Diese KI-Modelle basieren häufig auf **neuronalen Netzen**



GenAI Architekturen

Architektur	Beispielhafte Aufgaben	Ausgabeformate/Beispiele
GANs (Generative Adversarial Networks)	- Realistische Bilder erzeugen (Deepfakes) - Künstliche Kunstwerke generieren	- Fotorealistische Bilder - Deepfake-Videos

GenAI Architekturen

Architektur	Beispielhafte Aufgaben	Ausgabeformate/Beispiele
GANs (Generative Adversarial Networks)	- Realistische Bilder erzeugen (Deepfakes) - Künstliche Kunstwerke generieren	- Fotorealistische Bilder - Deepfake-Videos
VAEs (Variational Autoencoders)	- Bildrekonstruktion - Datenkompression (z. B. in der Medizin)	- Verbesserte oder komprimierte Bilder - Medizinische Bildgebung

GenAI Architekturen

Architektur	Beispielhafte Aufgaben	Ausgabeformate/Beispiele
GANs (Generative Adversarial Networks)	- Realistische Bilder erzeugen (Deepfakes) - Künstliche Kunstwerke generieren	- Fotorealistische Bilder - Deepfake-Videos
VAEs (Variational Autoencoders)	- Bildrekonstruktion - Datenkompression (z. B. in der Medizin)	- Verbesserte oder komprimierte Bilder - Medizinische Bildgebung
Transformer-Architekturen	- Textgenerierung (z. B. Chatbots wie GPT) - Sprachübersetzung - Textzusammenfassung	- Automatische Texte (Blogposts, Geschichten) - Übersetzte Texte (z. B. in Google Translate)

Was sind transformer-basierte LLMs?

Transformer-basiert LLMs

- LLM = Large Language Model
- Basieren auf Deep Neural Networks (DNNs)
- Menschliche Sprache verstehen und generieren
- kontextabhängige Antworten geben
- Natural Language Processing (NLP)

Im Weiteren sprechen wir über LLMs (nicht LMs).

Natural Language Tasks

- Texte zusammenfassen
- Gedichte schreiben
- Geschichten erzählen
- Texte übersetzen
- Entitäten erkennen
- Etc.

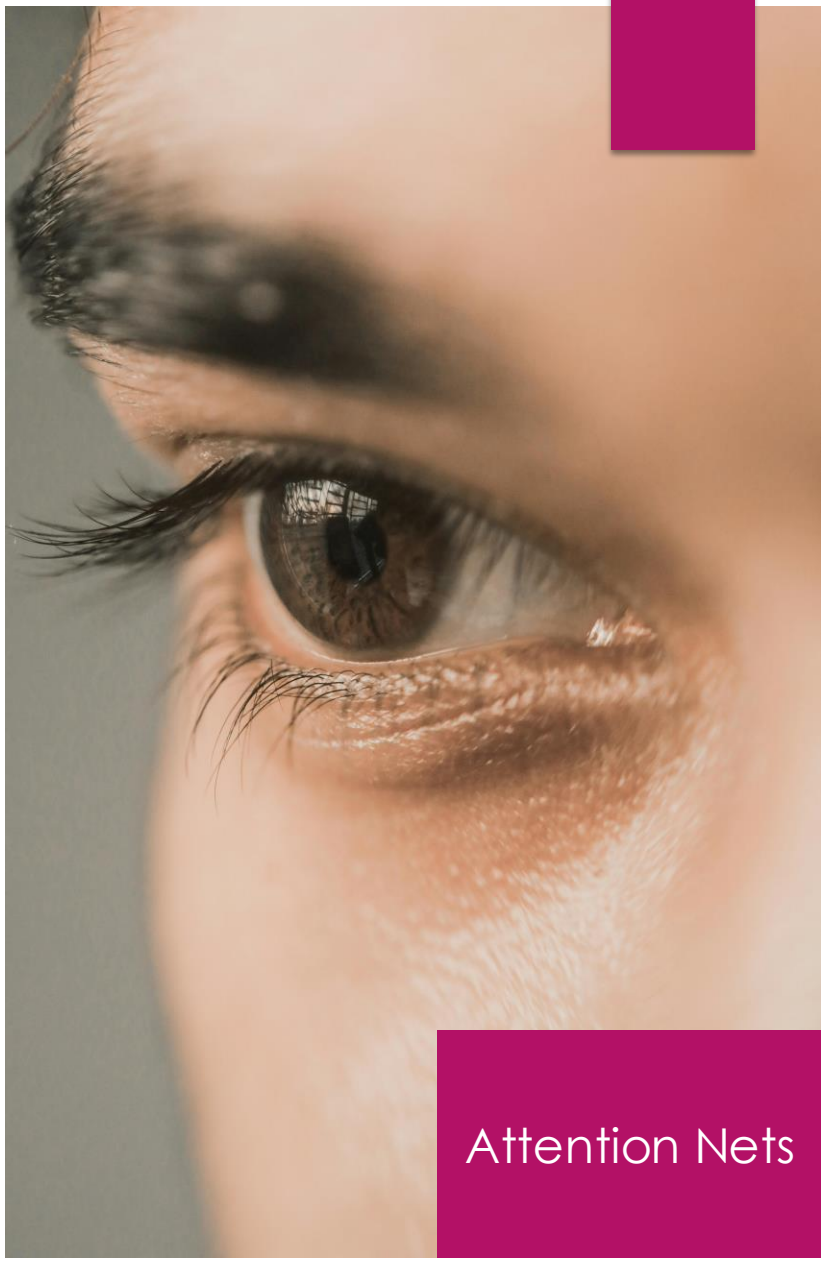
Ist das Schreiben / Analysieren von Unit Tests eine NLP Aufgabe?

- Nein!
- LLMs können Unit Tests generieren oder analysieren, weil sie auf riesigen Mengen an Programmier- und Textdaten trainiert sind
- Sie haben gelernt, den Aufbau von Code, Teststrukturen und gängigen Programmierpraktiken zu verstehen

Ist das Schreiben / Analysieren von Unit Tests eine NLP Aufgabe?

- Können durch Mustererkennung Codevorschläge, Testszenarien und Fehleranalysen erstellen
- Basierend auf ihrem Verständnis von Programmiersprachen und Softwareentwicklungspraktiken
- Ersetzten keine Testwerkzeuge (Testrunner, Assertion Libraries) => sie dienen nur der Unterstützung!

Transformer-Architekturen



Attention Nets

Transformer-Architekturen

- Encoder-Architektur
  - Inhalt verstehen
  - Fokus auf Bedeutung von Wörtern im Kontext
  - Beispiel: BERT

Transformer-Architekturen

- Decoder-Architektur
  - Unidirektional
  - Autoregressiv
  - Wörter vorhersagen
  - Texte generieren
  - Basierend auf vorherigem Text
  - Beispiel: GPT-Modelle

Transformer-Architekturen

- Encoder-Decoder-Architektur (Seq2Seq)
  - Encoder versteht und fasst zusammen
  - Decoder nimmt Input und baut darauf auf
  - Beispiel: T5

Aufmerksamkeit (Attention-Mechanismen)

Genutzt vom Encoder

- Um den Kontext eines Wortes im Satz erfassen

Beispiel: Die **Bank** hat viel Geld. Lina sitzt auf der **Bank**.

Genutzt vom Decoder

- Um das vorherige Wort zu berücksichtigen, während es das nächste Wort vorhersagt

Beispiel: Es sieht nach Regen aus, denn der Himmel ist **bewölkt**.

Multi-headed Attention

Self-Attention

KI-Architekturen

Architektur-Stil	Modell	Ersteller
Encoder	Bert, ALBERT, ELECTRA	Google
	RoBERTa	Facebook AI Research
	DistilBERT	Hugging Face
Decoder	GPT-3, GPT-4	OpenAI
	GPT-J, GPT-Neo	EleutherAI
	LLaMA	Meta (Facebook)
Encoder-Decoder	T5	Google
	BART, mBART	Facebook AI Research
	MarianMT, ProphetNet	Microsoft

Wie lernt ein transformer-basiertes LLM coden?

Pre-training

- Von Grund auf erlernen (allgemeine "Fitness")
- Lernt grundlegende Muster und Zusammenhänge
- Auf großen allgemeinen Datensätzen trainiert (z.B. Wikipedia, Common Crawl)

Finetuning

- bereits trainiertes Modell wird für eine spezielle Aufgabe oder Domäne weiter trainiert
- mit spezifischeren Daten (Text, Bilder, Audio, Video, Nutzerfeedback)
- Leistungsverbesserung für bestimmte Aufgaben oder Domänen

Wie kann ich transformer-basierte LLMs fürs Unit-Testen nutzen?

KI-Modelle nutzen

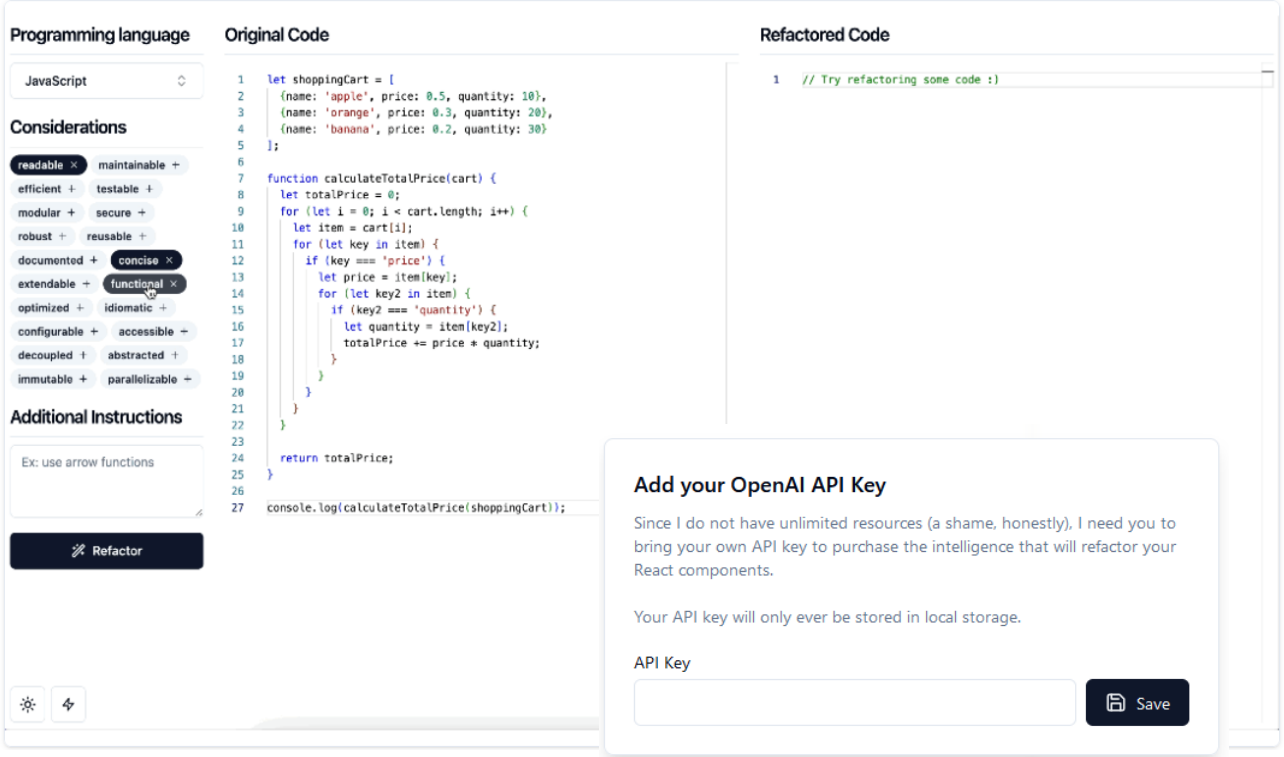
- 1) In IDE integrierte Tools zur Code-Vervollständigung
- 2) Modell selbst anbinden, um direkt aus der IDE zu prompten
- 3) Modell mit vorhandenem User Interface zum Prompten

Die Prompt-UI: DIY vs. tailored

Refactory:

- Benutzerfreundliche Oberfläche
- Verbesserte Prompts
- Ergänzungen

- **Hauptunterschied:** Wer entscheidet wie ein Prompt und der Output auszusehen hat?



Produkte für Unit Testing mit Jest

Tool	Modell	Interface Typ	Trainiert/ fine-tuned	Datensätze
Workik	Eigenständiges Modell	prompt-ui	trainiert	Allgemeine Programmier- und Testdatensätze
TestGPT (Feature in CodiumAI)	GPT-3.5-turbo / GPT-4	prompt-ui	fine-tuned	Codeschnipsel, Dokumentation, reale Codebasen
Airtest		ide-integration	trainiert	Große Datensätzen für die Testgenerierung in Typescript, Python und anderen Frameworks
GitHub Copilot	GPT-basiert (OpenAI Codex)	ide-integration	fine-tuned	Öffentlichen Code-Repositories, Open-Source-Projekte, Dokumentation von Rigor und Workik
CodiumAI	TestGPT model	ide-integration	trainiert	Diverse Code Repositories for various languages including JavaScript, TypeScript, and Python

Produkte für Unit Testing mit Jest

Tool	Modell	Interface Typ	Trainiert/Fine-tuned	Datensätze
Tabnine	GPT-basiert (customized)	ide-integration	Fine-tuned	öffentlich zugänglicher Code, mit Optionen für Training mit privaten Repositories
testRigor	Eigenständiges Modell	ide-integration	trainiert	funktionale Testdaten, benutzergenerierte Testfälle und natürlichsprachliche Befehle
AskCodi	GPT-3.5 / Benutzerdefiniert	prompt-ui	Fine-tuned	Programmiercode-Beispiele, Dokumentation, Hinweise für Entwickler
AI-Powered Unit Test Generator	OpenAI GPT-3.5 / 4	prompt-ui	Fine-tuned	Coding-Standards, Beispiele für Testfälle

Limited

Produkte für Unit Testing mit Jest

Tool	Modell	Interface Typ	Trainiert/Fine-tuned	Datensätze
Devin von Cognition	GPT-3.5	prompt-ui	trainiert	Allgemeine Code- und Testmuster
Bito.ai	GPT-3.5	prompt-ui	Fine-tuned	Code-Repositories, Entwicklerdokumentation
CodeGen	GPT-3.5	prompt-ui	Fine-tuned	Verschiedenste Code-Ressourcen und Standards
ChatGPT-3	OpenAI GPT-3	prompt-ui	trainiert	Breiter Textkorpus einschließlich Code und Dokumentation
Amazon CodeWhisperer	Eigenständiges Amazon-Modell	ide-integration	Trainiert (speziell für die AWS-Welt)	Code repositories, AWS Dokumentation, echte Code-Beispiele

Limited

Kosten-los

Produkte für Unit Testing mit Jest

Tool	Modell	Interface Typ	Trainiert/Fine-tuned	Datensätze
Devin von OpenAI	GPT-4	prompt-ui	trainiert	Allgemeine Code- und Testmuster
CodePal Unit-Tests Writer	OpenAI-Modelle	Prompt-ui / IDE-integration	trainiert	Verschiedene Quellen (unklar)
replit	Basiert wahrscheinlich auf OpenAI-Modellen	IDE-integration	unbekannt	unbekannt
HuggingChat	verschiedene	Prompt-ui	unbekannt	unbekannt

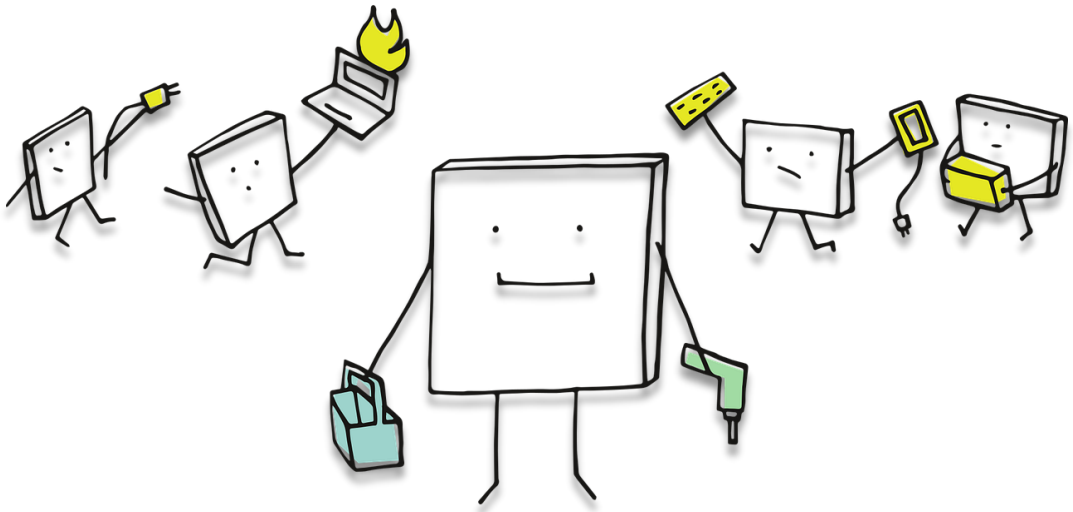
limited

kostenlos

Wie funktioniert das bei Tools, die ich in meine IDE integriere?

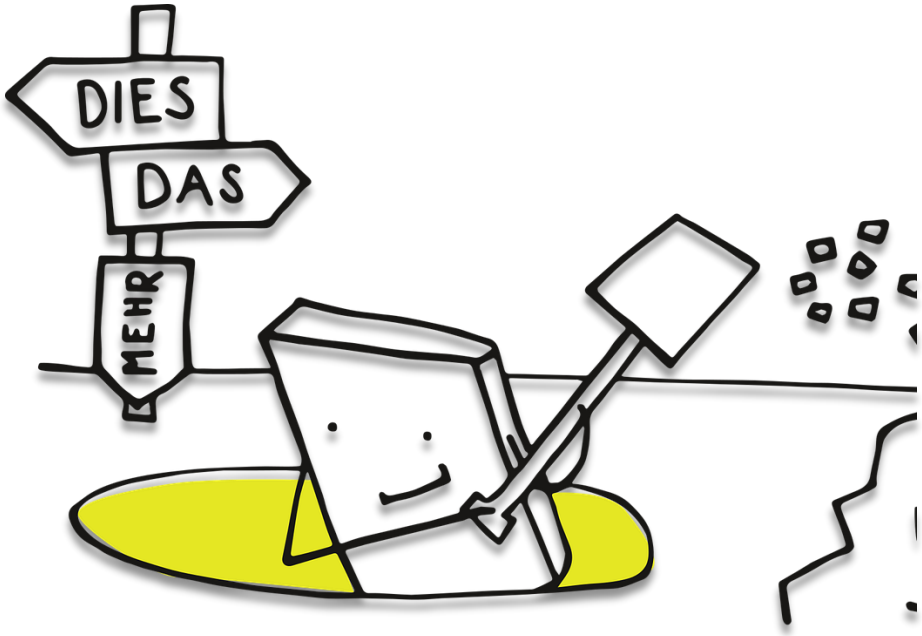
GitHub Copilot – Was ist das?

- ein KI-gestütztes Werkzeug, das beim Coden hilft
- Basiert auf OpenAI Codex
- speziell darauf trainiert, Code zu verstehen und zu schreiben
- Mit Milliarden von Codezeilen aus öffentlich zugänglichen Code-Repositories trainiert



GitHub Copilot – Wie funktioniert das?

- **Schritt 1:** Analsysiert die Texteingabe
  - Welche Programmiersprache?
  - Zweck des Codes?
- **Schritt 2:** Macht Vorhersagen
- **Schritt 3:** Präsentiert Vorschläge
  - Nutzer: Annehmen oder ablehnen?
- **Schritt 4:** Lernt und passt sich an
  - lernt Stil und Kontext des Projektes kennen



GitHub Copilot: Gut zu wissen

Der Kontext ist relevant

Domänenwissen: Medizin, Bank etc.

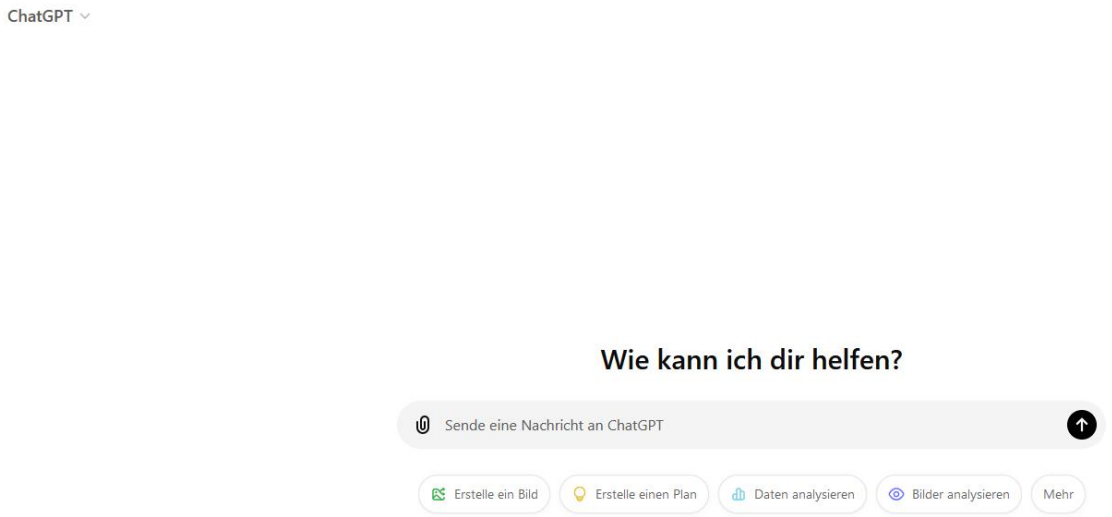
Gute Dokumentation bringt Vorteile

Dokumentations-dateien, API-Doku, Schnittstellen-beschreibungen, technische Handbücher etc.

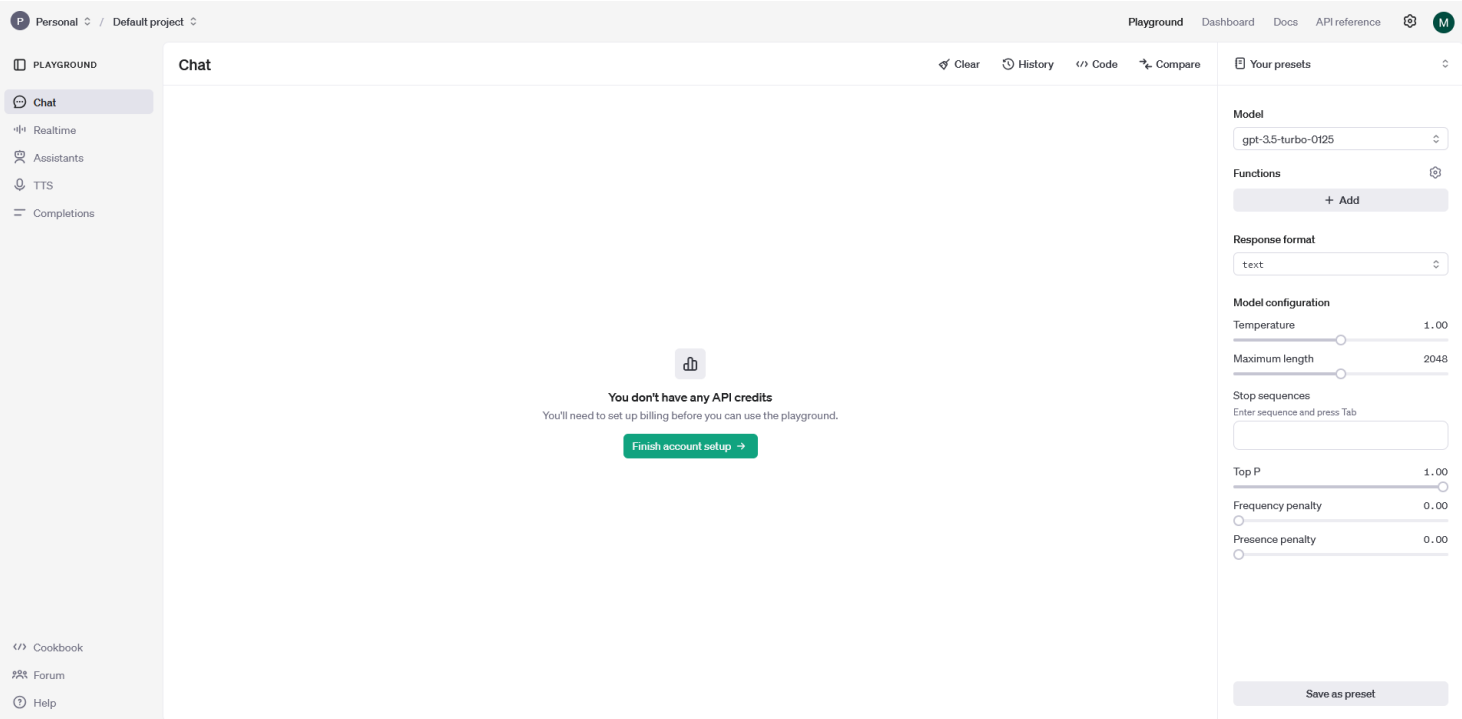
Kommentare und Docstrings (bspw. JSDoc) sind hilfreich

Wie funktioniert das bei Tools, die mir eine Prompt-UI zur Verfügung stellen?

Prompt-UI

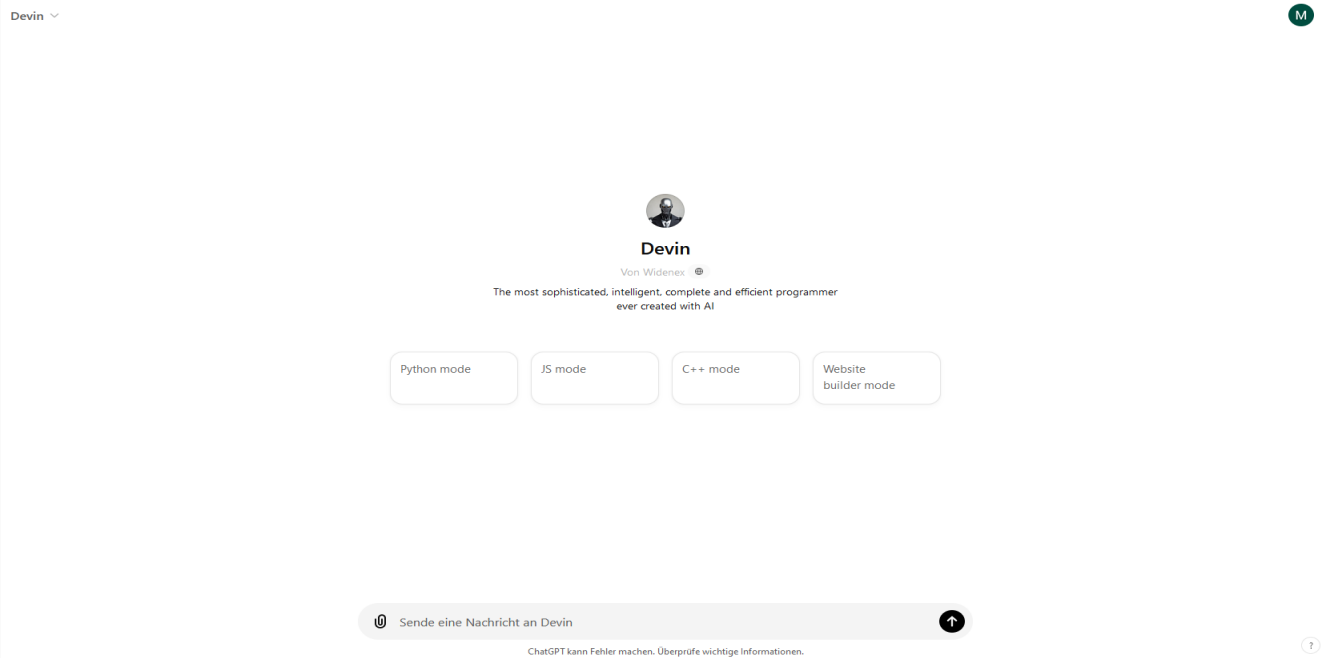

ChatGPT
Wie kann ich dir helfen?
Sende eine Nachricht an ChatGPT
Erstelle ein Bild
Erstelle einen Plan
Daten analysieren
Bilder analysieren
Mehr

Playground-UI: ChatGPT-3.5



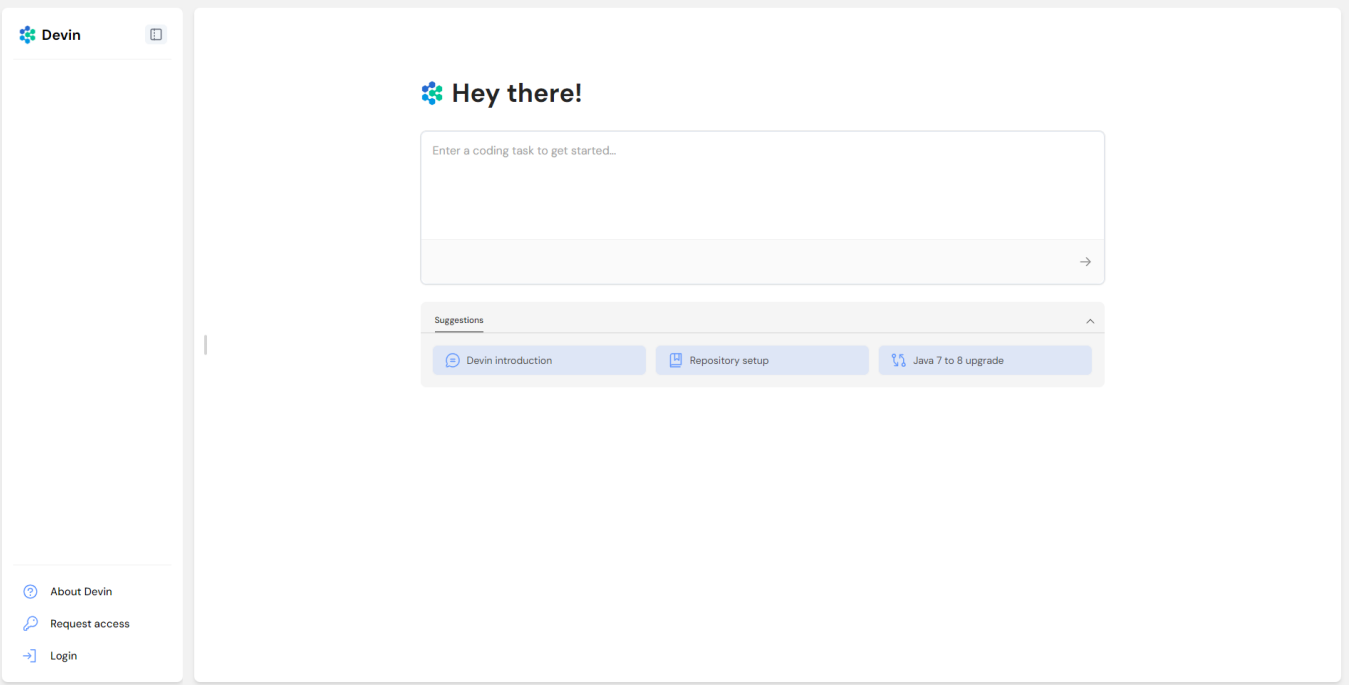
Custom GPTs: Devin von Widenex

- https://gpts.widenex.com/



Devin - the first AI software engineer

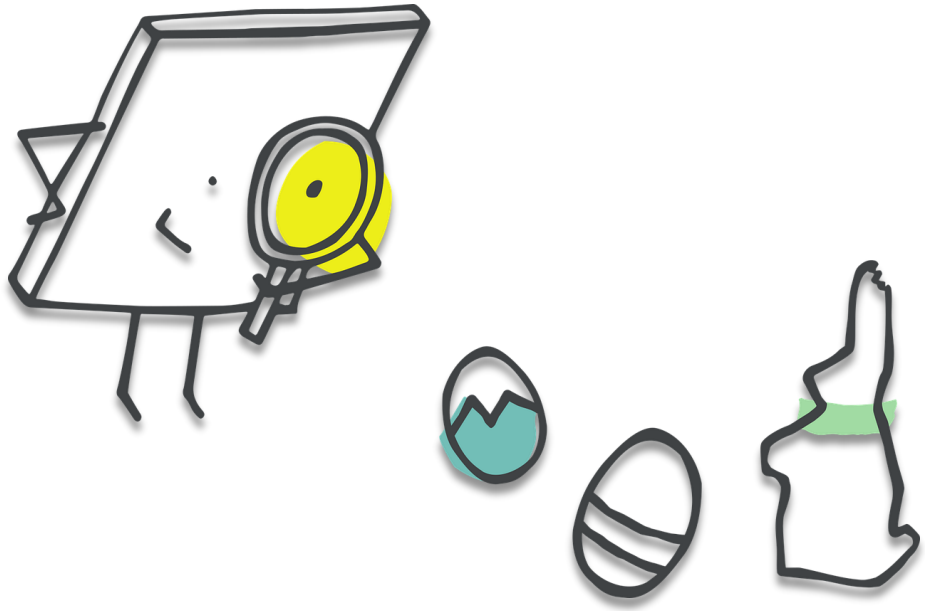
- Devin (the Developer)



Praktische Übungen

Test Driven Development (TDD)

- **Das Henne-Ei-Problem:** Wer ist zuerst da? Test-Code oder Produktivcode?
- **Ziel:** kontinuierliches Refactoring und Verbesserung
- **Vorteile:** Qualität, Wartbarkeit, Sicherheit



Test Driven Development mit KI

Copilot Chat: Code Generation with Test-Driven Development

How CopilotChat Works

Follow

Developer 1. Define Test Cases	LLM 2. Code Generation	Copilot Chat 3. Validation
• Define inputs and expected outputs. • Provide an optional requirement description.	• LLM generates code based on the test cases and requirement description.	• Copilot Chat validates the generated code. • If a test case fails, Copilot Chat iteratively interacts with LLM to refine the code until all test cases pass.

Define Test Cases Example 1 | Example 2 | Example 3 | Example 4 | Example 5 | Clear

```
input: "git+https://github.com/group1/name1.git"          expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+https://github.com/group1/name1"              expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+http://github.com/group1/name1"               expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+https://github.com/group1/name1.git"          expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+http://github.com/group1/name1.git"           expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+http://github.com/group1/name1.git#main1"     expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+http://github.com/group1/name1#main1"         expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+https://github.com/group1/name1.git#main2"    expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+https://github.com/group1/name1#main2"        expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git://github.com/group1/name1.git"                expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+ssh://git@github.com/group1/name1.git"        expectedOutput: { groupName: "group1", projectName: "name1" }
input: "git+ssh://git@github.com/group1/name1"            expectedOutput: { groupName: "group1", projectName: "name1" }
```

Requirement Description

Parse github url into an object {group, project}

Generate JavaScript Function By AI

58% Iteration 1

100% Iteration 2

Code Generated By Copilot Copy Code

```
function parseGitUrl(input) {
    // Remove the protocol and any fragment identifiers
    const cleanedInput = input.split('#')[0];
```

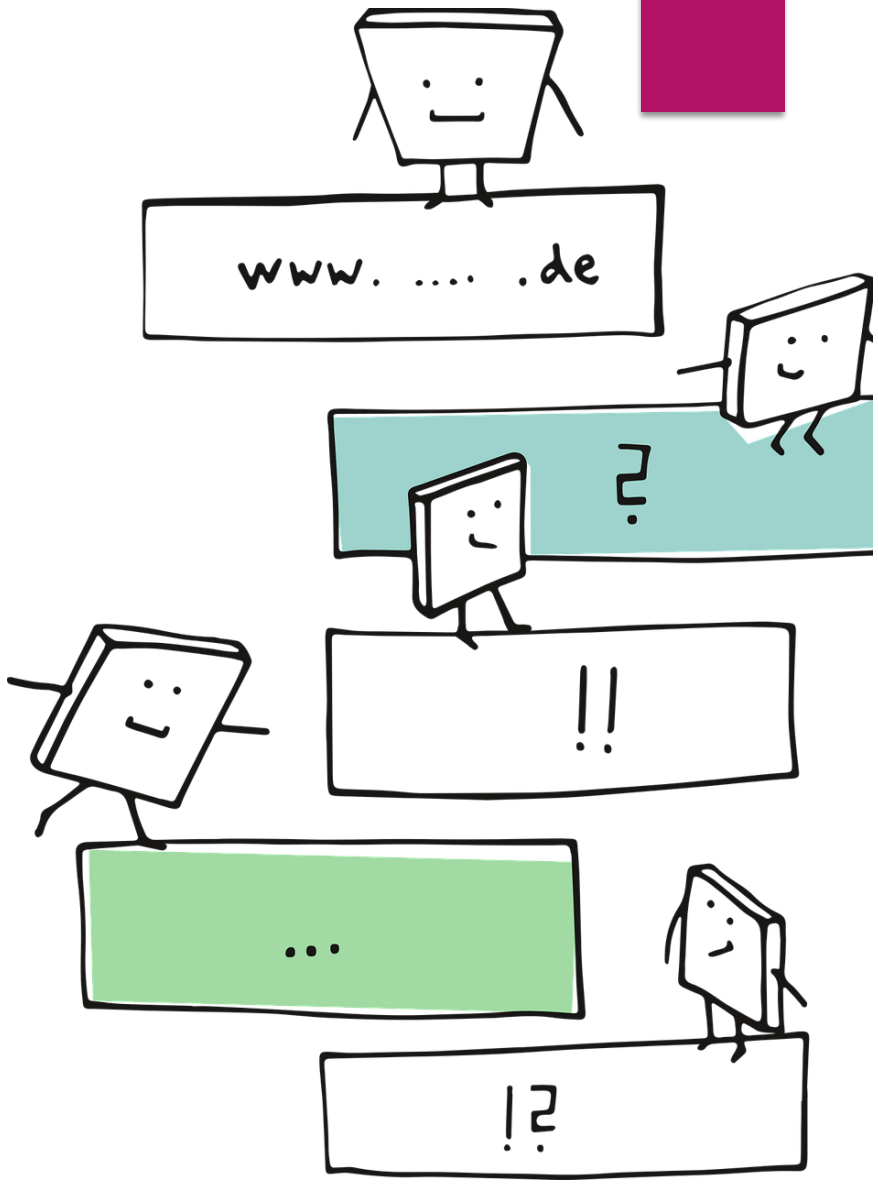
Übung: TDD mit KI

Beispiel in TDD.md

- **Palindrom-Prüfung**: Funktion, die prüft, ob ein String ein Palindrom ist.
- **Sortierung**: Implementiere eine Funktion, die eine Liste von Zahlen aufsteigend sortiert (z.B. Bubble Sort oder Selection Sort).
- **Primzahlen**: Funktion, die prüft, ob eine Zahl eine Primzahl ist.

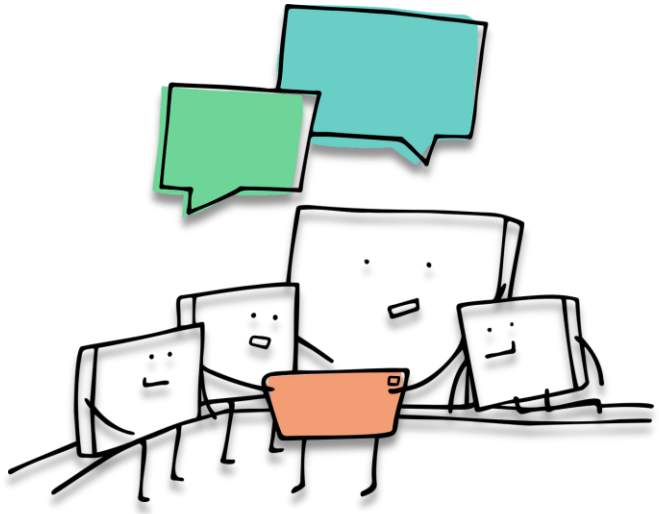
In-Context Learning

- Fähigkeit aus bereitgestellten Eingaben bei Interaktion zu lernen
- Keine Anpassung des Modells notwendig (Training oder Finetuning)
- Agent verwendet Kontext des Gesprächs



Prompt Engineering – Was ist das?

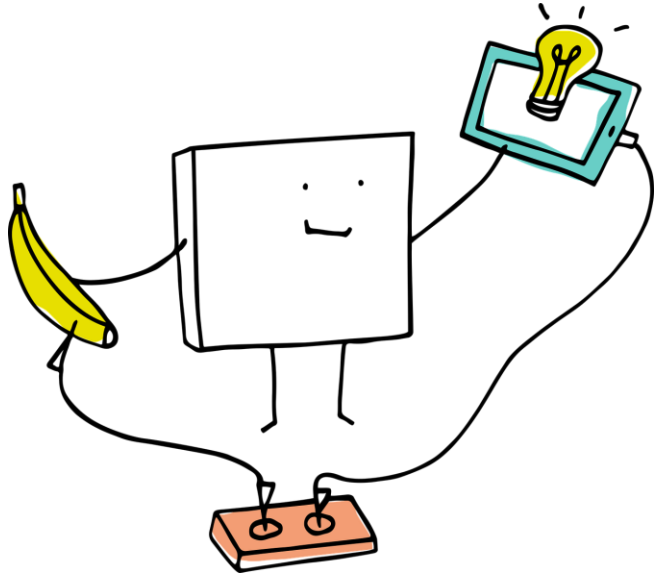
- Gestaltung und Optimierung von Eingaben (Prompts) für KI-Modelle
- Techniken zur gezielten Anleitung von LMs
- Ziel: Antworten sollen präziser und relevanter werden



Beispiel

```
add(a: number, b: number) {
   return a + b;
}
```

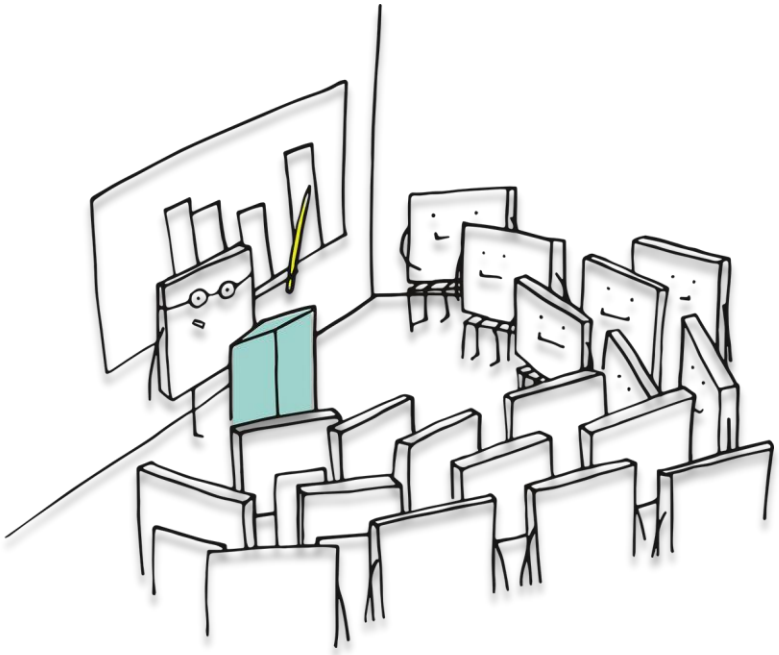
Lass ein KI-Modell diese Funktion vollständig testen.
Was würdest Du tun?



Prompt-Bestandteile: Anweisung (Instructional)

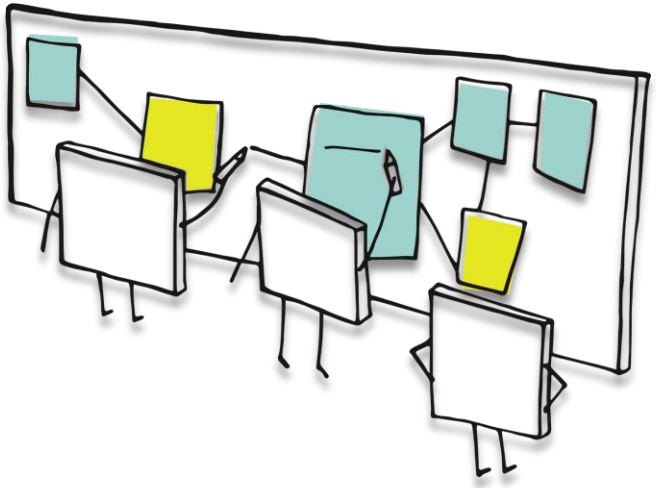
- Gibt an, was das Modell tun soll
- Klare und präzise Beschreibung der Aufgabe oder der Frage, die gestellt wird

- **Beispiel:** „*Erstelle einen Unit Test mit Jest für die Funktion add(a, b), die die Summe von a und b zurückgibt.*“



Prompt-Bestandteile: Kontext

- Bietet Hintergrundinformationen oder zusätzliche Details, die für die Aufgabe relevant sind
- Hilft dem Modell, die Anfrage besser zu verstehen
- **Beispiel:** „*Die Funktion add(a, b) ist eine einfache JavaScript-Funktion, die zwei Zahlen als Parameter entgegennimmt und deren Summe zurückgibt. Es ist wichtig, verschiedene Szenarien zu testen, einschließlich typischer Werte, Grenzwerte und falscher Eingaben.*“



Prompt-Bestandteile: Erwartung

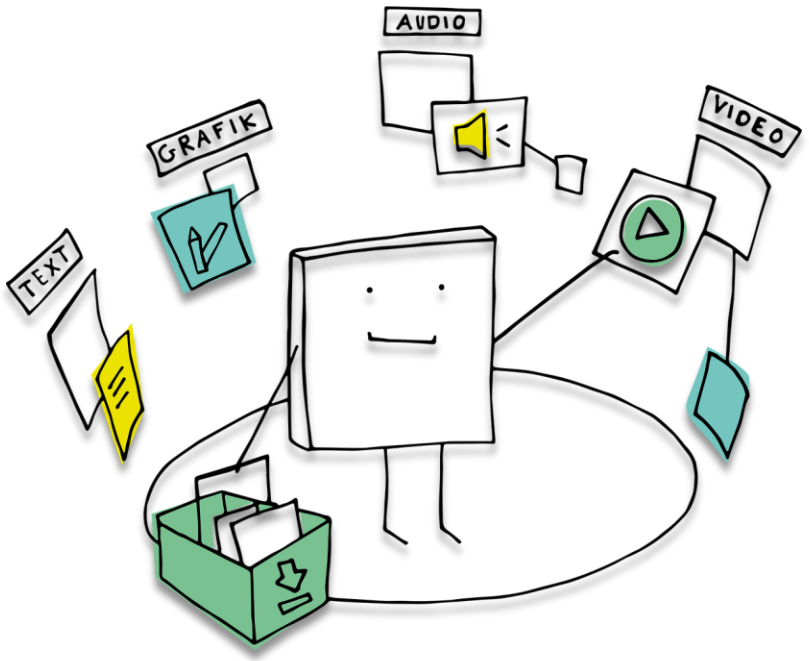
- Beschreibt, was man von der Antwort erwartet, einschließlich des Inhalts und der Art der Informationen
- Konzentriert sich auf den Inhalt und die Qualität der Antwort

- **Beispiel I:** *"Gib mir eine vollständige Testsuite, die mindestens die folgenden Szenarien abdeckt: Summe zweier positiver Zahlen, Summe zweier negativer Zahlen, Summe einer positiven und einer negativen Zahl, Umgang mit null als Eingabe, Umgang mit ungültigen Eingaben (z. B. Strings, undefined)"*
- **Beispiel II:** *"Ich erwarte eine Branch Coverage von 100%."*

Prompt-Bestandteile: Ausgabeformat

- Legt fest, wie die Antwort strukturiert oder präsentiert werden soll (Fließtext, JSON, YAML etc.)
- Das äußere Erscheinungsbild der Antwort

- **Beispiel:** „Präsentiere die Testfälle in Form von beschreibenden Testblöcken (describe und it-Blöcken). Testbeschreibungen sollen it 'should' starten, sodass jeder Test semantisch mit 'it should' startet.“



Prompt-Bestandteile: Einschränkungen

- Setzt Grenzen oder Bedingungen, die bei der Erstellung der Antwort beachtet werden müssen (z.B. Textlänge)

- **Beispiel:** *„Halte den Testcode einfach und übersichtlich. Verwende nur native Jest-Methoden ohne externe Bibliotheken. Testbeschreibungen dürfen aus maximal 10 Wörtern bestehen.“*

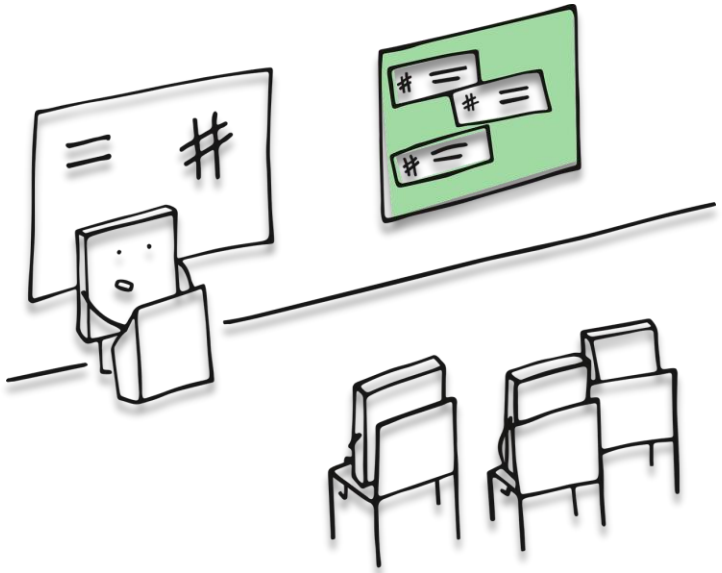
Prompt-Bestandteile: Zielgruppe

- Definiert, für wen die Antwort gedacht ist
- Dies kann helfen, den Ton oder Stil der Antwort anzupassen

- **Beispiel:** „*Die Tests werden später von strengen Reviewern geprüft, die die Qualität und Vollständigkeit der Tests bewerten werden. Sie erwarten, dass der Code klar kommentiert ist und die Absicht hinter jedem Testfall deutlich wird.*“

Prompt-Bestandteile: Beispiele (Shots)

- Zeigt der KI, wie eine Antwort aussehen könnte oder wie die Anfrage interpretiert werden soll
- Kann helfen, Missverständnisse zu vermeiden
- Besonders wichtig für In-Context Learning
- Anzahl der "Shots" in Prompts
  - Zero-Shot
  - One-Shot
  - Few-Shot



Bespiel

"Hier ist ein Beispiel für einen bestehenden Unit Test mit Jest:

Funktion:

```
multiply(a, b) { return a * b; }
```

Test:

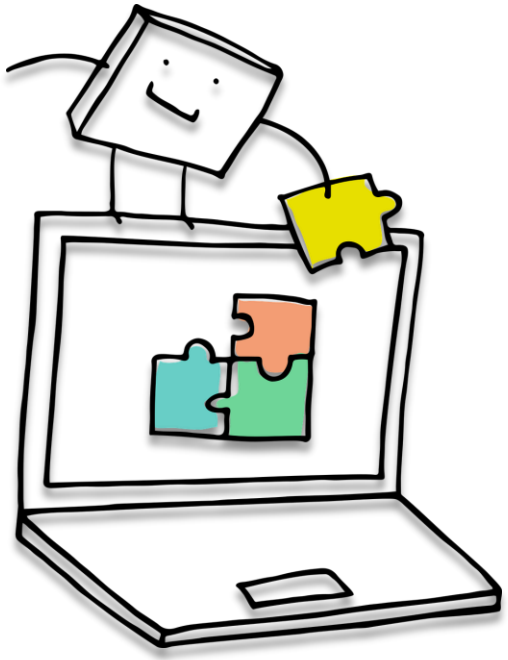
```
describe('multiply', () => {
    it('should return the product of two positive numbers', () => {
        expect(multiply(2, 3)).toStrictEqual(6);
    });
});
```

Bitte folge dieser Struktur für die Funktion: add(a, b): number { return a + b; }"

Prompt-Bestandteile: Trenner (Sectioning)

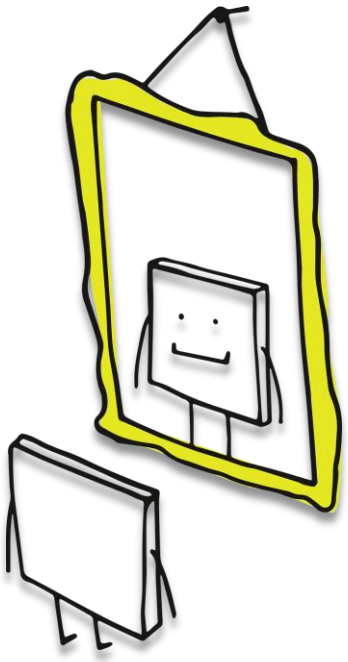
- Zerlegt den Prompt in sinnvolle Einheiten
- Modell kann Informationen leichter extrahieren
- KI-Modell kann besser den Fokus setzen
- Wahrscheinlichkeit von Missverständnissen wird geringer
- Nutzer kann eigenen Prompt besser lesen

- **Beispiel:** "### *Zielgruppe* ###"



Reflexion & Selbstvalidierung

- **Reasoning:** *"Erkläre bei jedem Schritt, was du tust."*
- **Andere Perspektive annehmen:** *"Ich bin nun eine Software Architektin und reviewe die Tests. Welche Verbesserungsvorschläge hätte ich?"*
- **Inhaltliche Validierung:** *"Nenne mir die F.I.R.S.T-Prinzipien. Halten alle Tests diese ein?"*



Prompt-Verkettung (Prompt-Chaining)

- Aufgabe in Unteraufgaben aufteilen
- Ergebnis vorherige Unteraufgaben als weiteren Input nutzen

- **Ich will eine Funktion testen. Was muss ich tun?**

Wir zerlegen den Prompt in einzelne Eingaben!

Vorteile

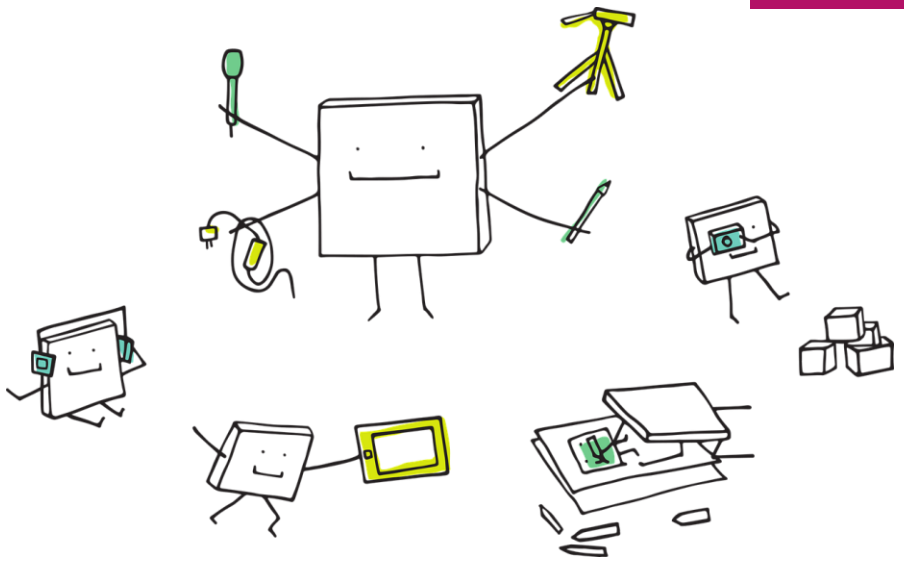
- **Komplexitätsreduktion:** Aufgaben werden in kleinere, leicht handhabbare Schritte unterteilt, wodurch die Lösung einfacher und präziser wird
- **Präzisionserhöhung:** Jeder Schritt im Prozess kann gezielt auf spezifische Details fokussiert werden, was zu relevanteren Ergebnissen führt
- **Iterative Verfeinerung:** Ergebnisse können schrittweise angepasst und verbessert werden, um optimale Lösungen zu erreichen

Chain-of-Thought Prompting

Modell zerlegt Prompt in Schritte!

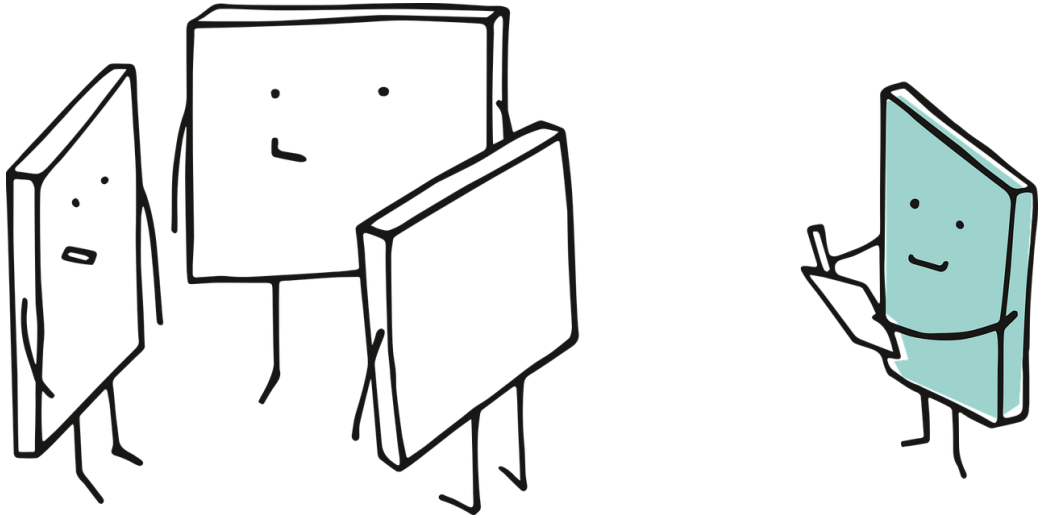
- **Schrittweise Problemlösung:** Modell bekommt strukturierte Aufforderung, die es dazu bringt, das Problem schrittweise zu lösen

- **Erklärung:** Das Modell erklärt die Schritte, die es nimmt, um die Lösung zu erreichen



Beispiel:

"Wie kann ich diese Datei testen? Bitte erkläre es Schritt für Schritt."



Vorteile

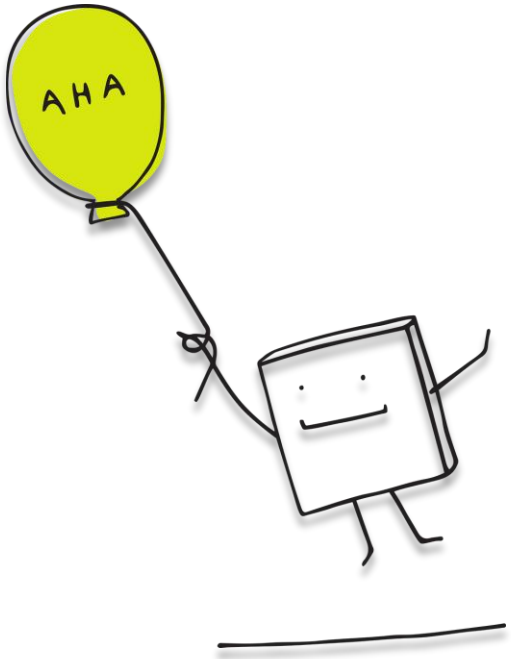
- **Schrittweises Denken:** Fördert Durchlaufen einzelner, klarer Schritte, was bei komplexen Problemen hilft, eine strukturierte Lösung zu entwickeln
- **Transparenz im Lösungsprozess:** Jeder Schritt wird explizit erklärt, was es einfacher macht, den Gedankengang nachzuvollziehen und zu lernen
- **Fehlerreduktion:** Durch die klare Aufteilung in Teilschritte wird die Wahrscheinlichkeit von logischen Fehlern oder Übersehen von Details reduziert

Generatives Wissens-Prompting

- Relevantes Hintergrundwissen vor der Aufgabe erzeugen oder abrufen
- Basiert auf der Idee, dass ein Modell, wenn es richtig mit Informationen und Kontext „gefüttert“ wird, präzise und nützliche Inhalte generieren kann

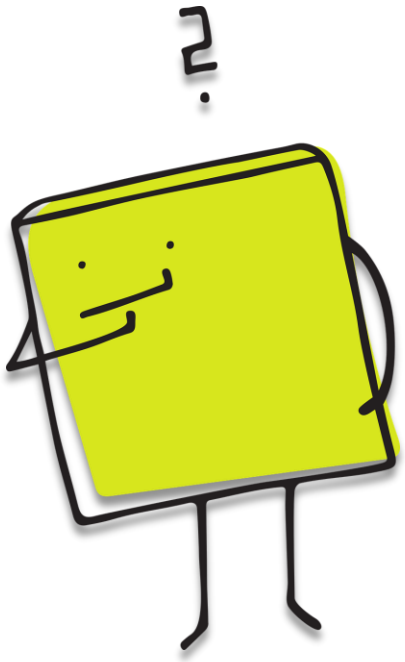
Wissen "füttern"

- **Ziel der Tests:** Wissen über Unit-Tests, Integrationstests oder andere
- **Test-Framework:** Wissen über Frameworks wie Jest, Mocha oder Jasmine
- **Best Practices:** Was macht gute Unit Tests aus?
- Testrunner, Assertion Libraries, Mocks & Spies etc.



Beispiel

- **Prompt 1:** *"Welche Jest Matcher gibt es und wann verwendet man welchen?"*
- **Prompt 2 (Anwendung des Wissens):** *"Hast du in allen Tests immer den passendsten Matcher eingesetzt?"*



Vorteile

- **Breites Fachwissen:** Bietet umfassendere, dynamischere Antworten, indem es auf das gespeicherte Wissen aus verschiedenen Bereichen zugreift

- **Kreativere Lösungen:** Liefert Vorschläge, die nicht nur die Problemstellung abdecken, sondern auch Optimierungen, Best Practices und alternative Ansätze beinhalten

- **Kontextuelle Tiefe:** Bietet Hintergrundinformationen und ermöglicht einen tieferen Einblick

Prompt – Gut zu wissen

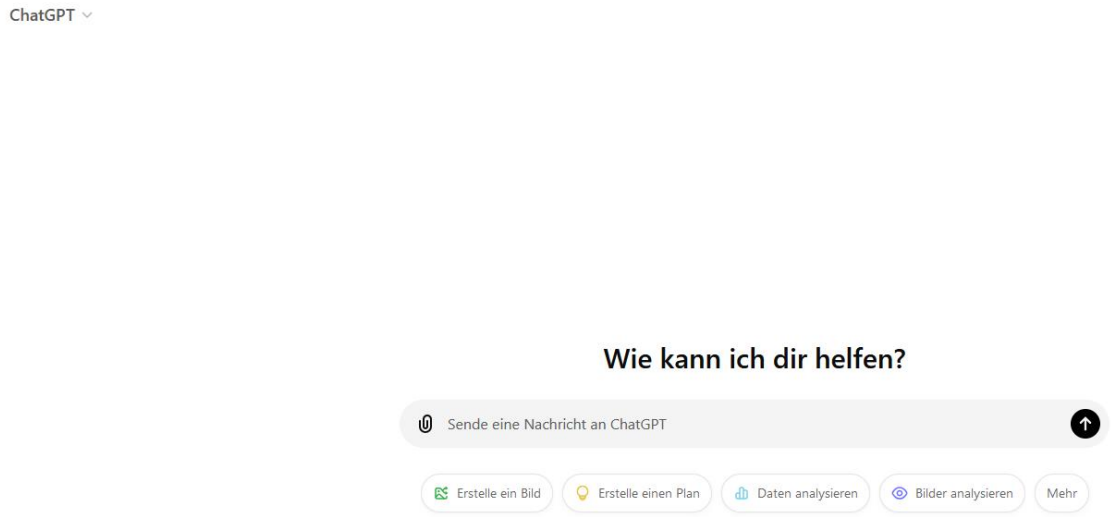
- Token
  - 100 Token ~ 75 Wörter in der englischen Sprache
  - Größte GPT-Modelle: 128.000 Tokens (Eingabe + Ausgabe) (GPT Turbo ~ 240 Seiten => Output auf max 3.000 Wörter begrenzt)
  - ChatGPT-3: 3000 Wörter
  - höhere Tokenzahlen für Kontexterhalt sind eher für längere komplexe Konversationen gedacht

Prompt – Gut zu wissen

- Eingrenzungen
  - Temperature (beim Testen niedrig! => 0,2)
  - Max-Length (keine Token verschwenden!)
  - Stop Sequences
  - Frequency & Presence Penalty (wir wollen Wiederholungen beim Testen!)
  - Top P (Nucleus Sampling)

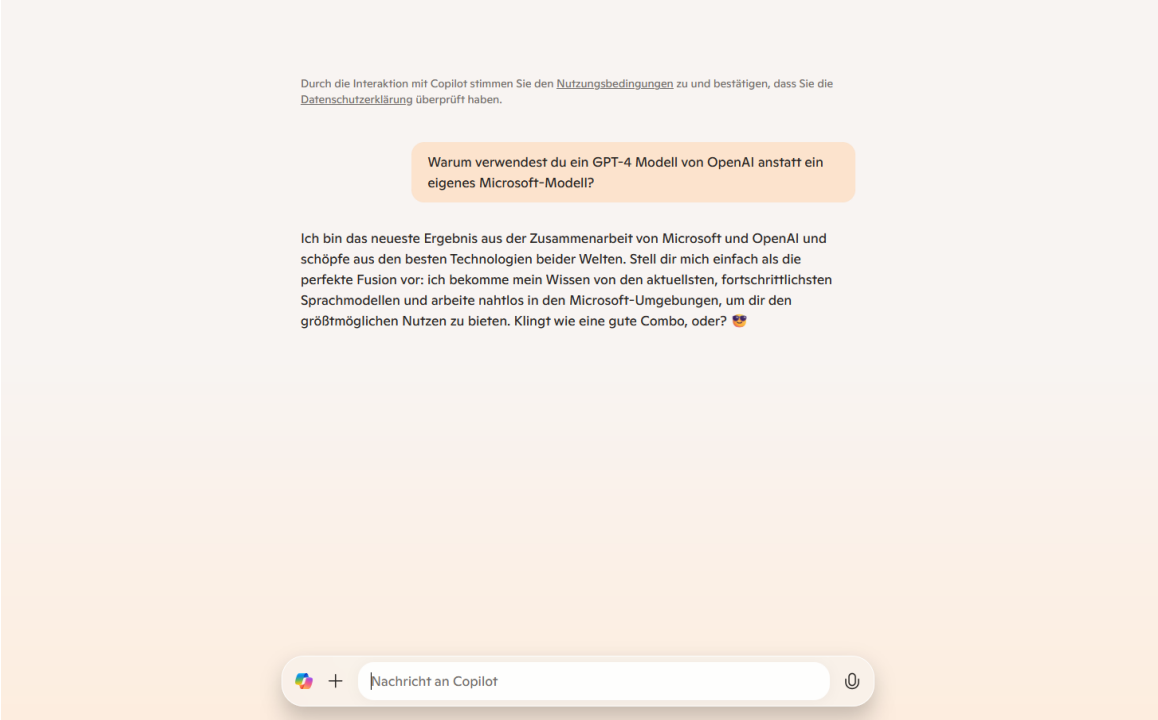
Angebote, die ihr kostenlos nutzen könnt

ChatGPT-3 (GPT-3)



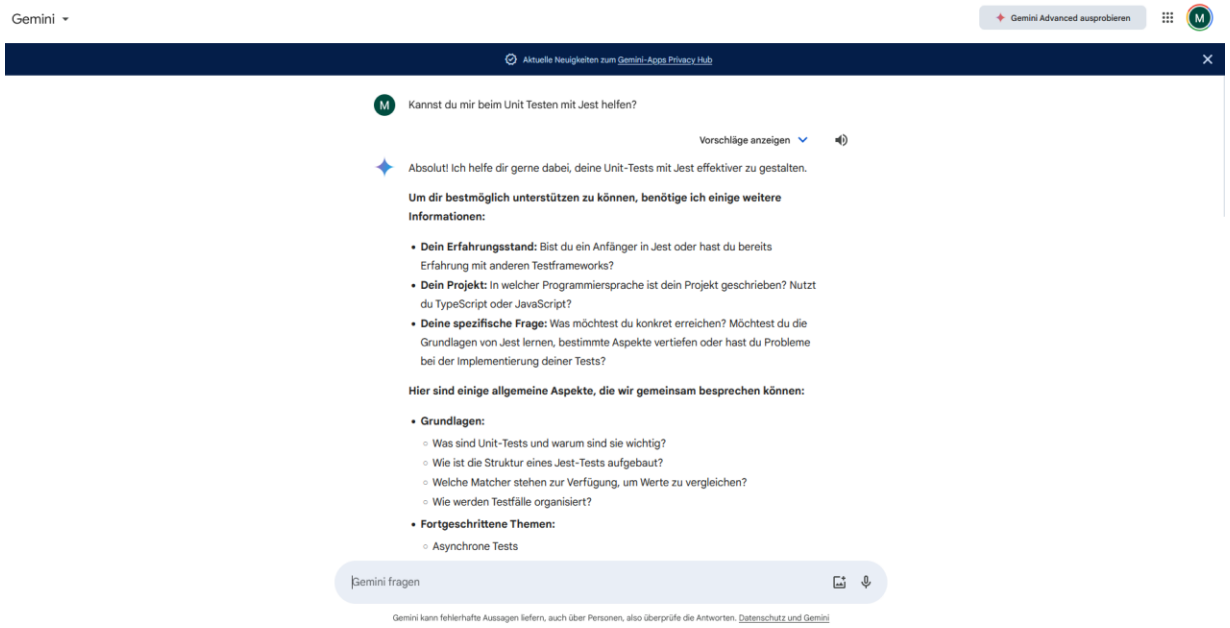
Microsoft Copilot (GPT-4)

- https://copilot.microsoft.com/



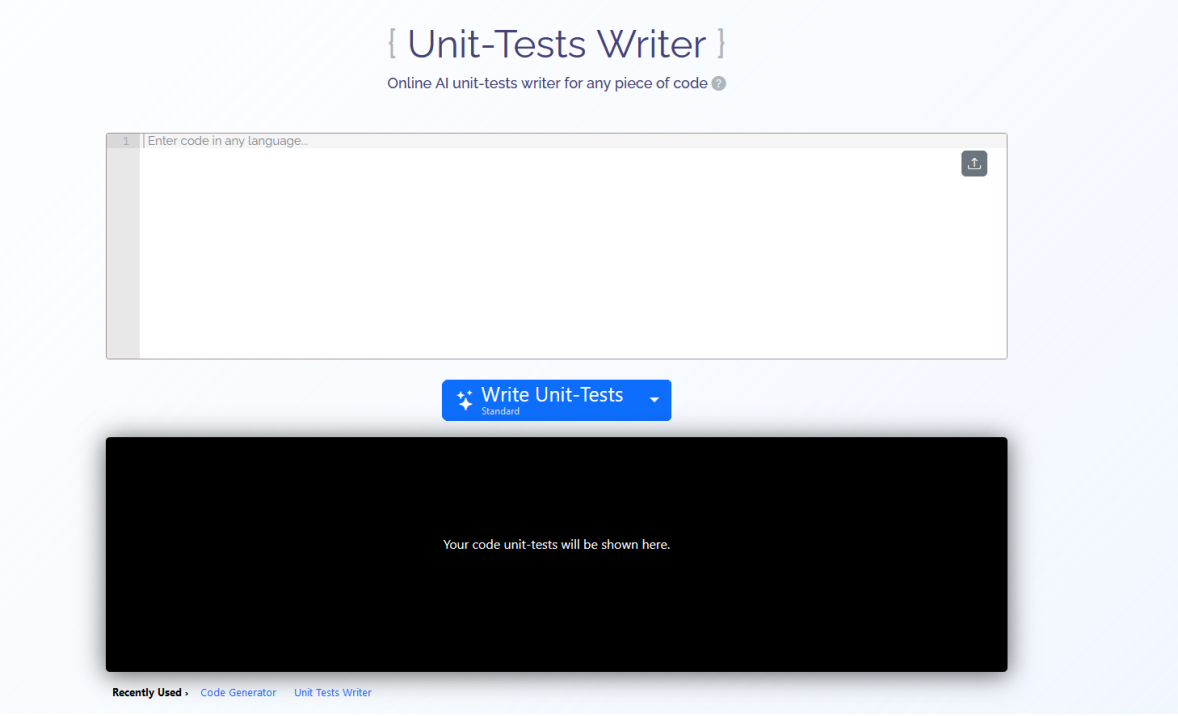
Gemini (LaMDA)

- https://gemini.google.com/app



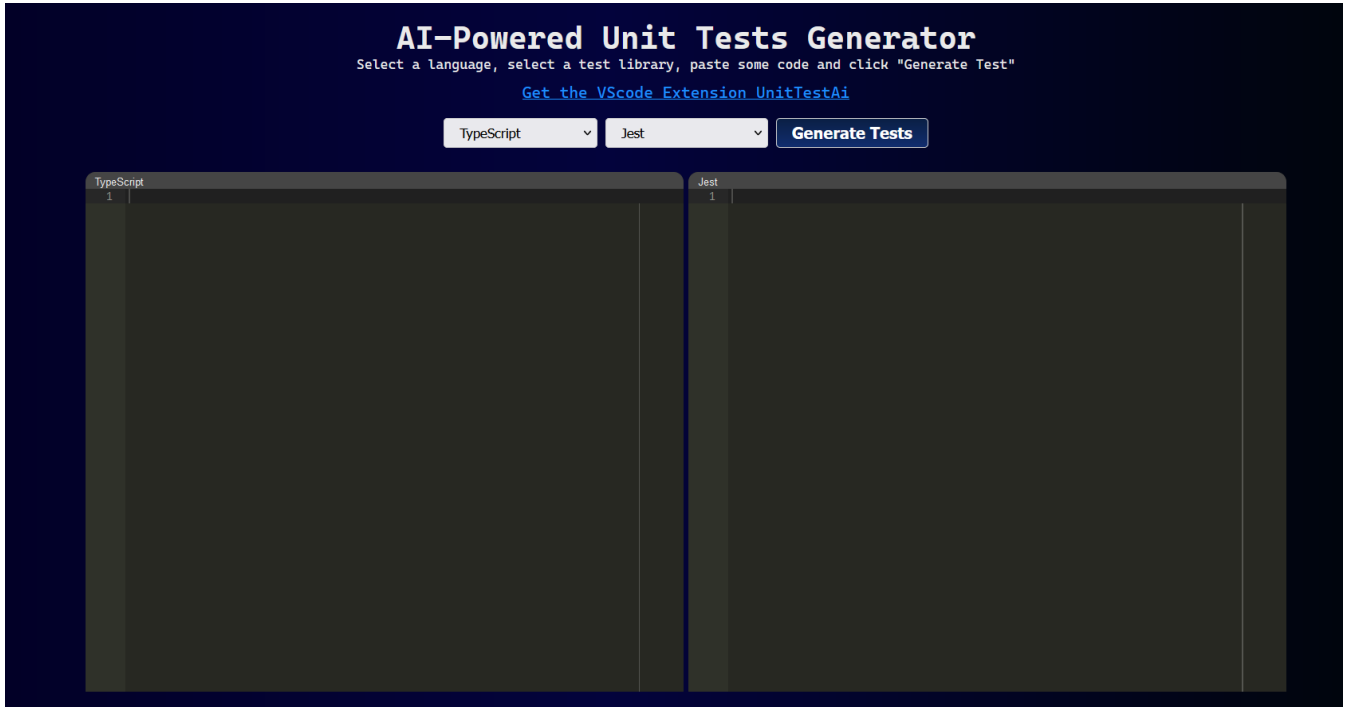
CodePal Unit-Tests Writer (GPT-Modelle)

- https://codepal.ai/unit-tests-writer



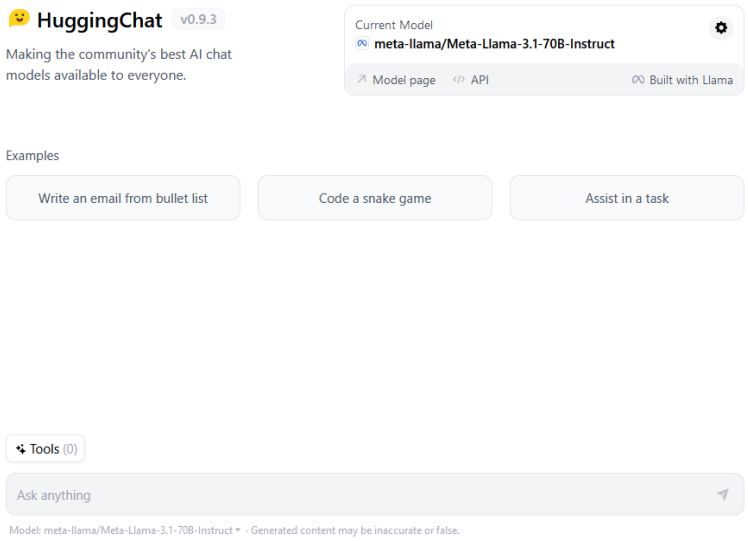
AI-Powered Unit Test Generator (GPT-3)

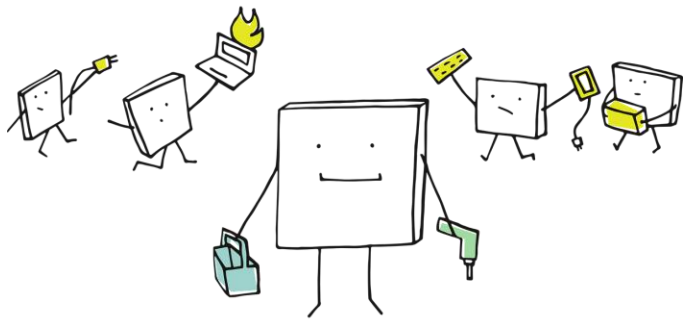
- Unit-test | Unit tests made easy with AI
- Finetuned von Paterson Anaccius



HuggingChat von HuggingFace

- https://codepal.ai/unit-tests-writer
- Kleine Demo...





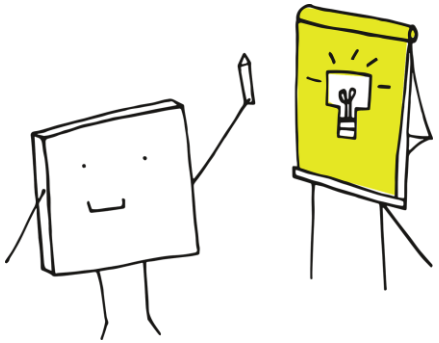
Wobei kann uns ein künstlicher Agent im Unit Test Prozess helfen?

Praktische Übungen

Gruppen-übung!

- Test schreiben
- Mehr über Jest lernen
- Testdaten schreiben
- Code testbar machen (Refactoring)
- Mocken (1)
- Testfälle identifizieren (2)
- Testsuite "aufräumen"
- Tests reviewen (3)

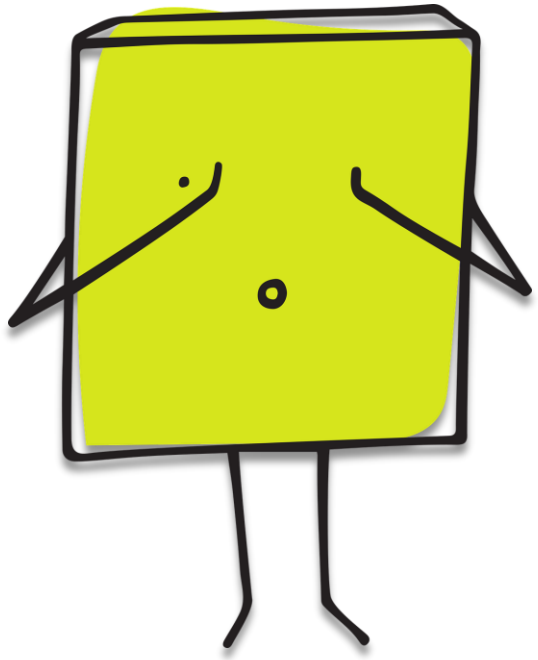
Praktische Tipps

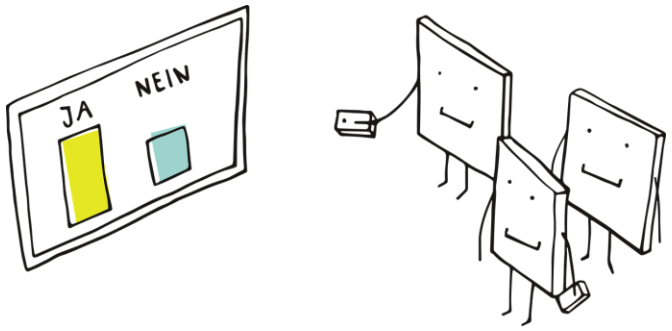


- Sammel gute Prompts und stelle sie anderen zur Verfügung
- Bleib up-to-date: lies Artikel zu den neusten Erkenntnisse im Prompt Engineering
- Um GenAI wirklich für Tests nutzen zu können, muss man Test-Theorie verstehen
- Um die richtigen Fragen zu stellen, muss man Wissen über Jest mitbringen
- Überprüfe alle generierten Ergebnisse immer nochmal manuell

Probleme

- Business-Code ist streng geheim
- Copyright – wem gehört der Code?
- Mehr Sicherheitslücken
- Fehler, für die keiner verantwortlich ist
- Nicht jeder Prompt funktioniert bei jedem Modell gleich gut
- Kosten vs. Profit





KI-Tools sind teuer. Lohnt sich das?

Prompt Engineering – hilfreiche Quellen

- https://www.promptingguide.ai/de
- https://gist.github.com/kostysh/dbd1dfb2181b96563754222903bf67e7
- Supercharging Your Unit Testing with Jest and ChatGPT: A Comprehensive Guide | by Priyanka Dhumane | Medium
- Unit Testing ChatGPT Prompts: Introducing Riteway for SudoLang | by Eric Elliott | JavaScript Scene | Medium
- ChatGPT - Prompts for Test Drive Development and Unit Testing - DEV Community
- https://platform.openai.com/playground
- AI For Testing: A Comparison of Unit Testing Tools | Zartis
- https://platform.openai.com/docs/overview
- https://kodora.ai/task/coding/

Quellen

- https://community.openai.com/t/cheat-sheet-mastering-temperature-and-top-p-in-chatgpt-api/172683
- https://community.openai.com/c/prompting/8
- [1706.03762] Attention Is All You Need