

Michael Kofler
Bernd Öggl
Sebastian Springer



Coding mit KI

Das Praxisbuch für
die Softwareentwicklung

+ Vom Prompt Engineering bis zu
Retrieval Augmented Generation

+ Besser programmieren und
Probleme mit KI lösen

+ KI ohne Cloud: LLMs lokal
nutzen und optimieren



Rheinwerk
Computing

Vorwort

Ziemlich sicher haben Sie es schon ausprobiert: Sie stellen ChatGPT oder einem anderen KI-Tool eine Coding-Frage – und das Tool spuckt innerhalb von Sekunden einen (scheinbar) perfekten Code aus.

In diesem Buch loten wir das Potenzial und die Grenzen von KI-Tools bei der Software-Entwicklung aus. Beginnen wir mit dem Positiven: Wir zeigen Ihnen, dass ChatGPT, GitHub Copilot & Co. mehr können, als ein paar Zeilen Code zu erzeugen. Sie können damit auch Unit-Tests erstellen, Fehler suchen, Refactoring-Aufgaben erledigen, Datenbank-Schemas entwickeln, SQL-Abfragen optimieren, Server administrieren und Scripts verfassen.

Die schöne neue KI-Welt hat aber auch Schattenseiten: Selbst wenn der Code plausibel aussieht und der Begleittext elegant formuliert ist, sind manche KI-Vorschläge schlicht falsch. Die Bandbreite ist groß: Logikfehler, Verwendung nicht vorhandener Funktionen oder die Bezugnahme auf Variablen, die in Wirklichkeit anders heißen. Manchmal funktioniert der Code zwar, aber er ist ineffizient oder unsicher.

KI-Tools machen Fehler, so viel ist unumstritten. Richtig eingesetzt sparen diese neuen Werkzeuge aber auch eine Menge Zeit. Darum geht es in diesem Buch.

Lokale Ausführung von Sprachmodellen

Gerade Firmen haben oft Probleme damit, ihre Code-Basis Tools anzuvertrauen, die in der Cloud laufen. Selbst wenn KI-Anbieter bei Unternehmensangeboten versprechen, Ihren Code nicht für das Training zukünftiger Sprachmodelle zu nutzen, stellt sich die Frage, wie weit Sie diesen Versprechen trauen und ob nicht doch Datenschutzprobleme auftreten können.

Der einfachste Ausweg besteht darin, die für KI-Tools erforderlichen Sprachmodelle lokal auszuführen. Das *Large Language Model* (LLM) von ChatGPT können Sie zwar nicht herunterladen, aber es gibt diverse Alternativen, die kostenlos verfügbar sind und ähnlich gut funktionieren.

Die lokale Ausführung von LLMs ist allerdings ressourcenintensiv. Nur leistungsstarke Notebooks oder PCs sind dazu geeignet. Neue CPUs mit *Neuronal Processing Units* (NPUs) versprechen diesbezüglich neue Möglichkeiten. Ein echter Nutzen für die lokale Ausführung von LLMs ergibt sich aber erst in Kombination mit geeigneter Software – und da muss sich noch viel tun.

Es gibt noch eine Option: Sie können in Ihrer Firma oder in einem Rechenzentrum einen eigenen Server einrichten, der allen Mitarbeitenden für KI-Aufgaben zur Verfügung steht. »Lokal« heißt dann zwar nicht auf dem eigenen Notebook, aber doch auf einem firmeneigenen Rechner. Wir zeigen Ihnen in diesem Buch, wie Sie lokale Sprachmodelle einsetzen und sogar mit eigenem Trainingsmaterial optimieren.

In diesem Buch stellen wir Ihnen die Programme *GPT4All*, *Ollama* und *Tabby* vor, um freie LLMs lokal auszuführen. Sie können die lokalen Sprachmodelle dann sowohl im Chat-Betrieb, zum Coding (z. B. mit dem Editor-Plug-in *Continue*) oder über eine API zur Entwicklung eigener KI-Anwendungen nutzen.

Coding mit KI-Unterstützung für Fortgeschrittene

Wenn Sie das erste Mal ChatGPT um Rat bei einem Coding-Problem gefragt haben, wenn Sie dem ersten Vorschlag von GitHub Copilot (oder einem vergleichbaren Tool) gefolgt sind, dann kennen Sie die interaktive KI-Unterstützung beim Coding. Aber da geht noch mehr!

Mit geeigneten Bibliotheken können Sie noch einen Schritt weiter gehen:

- ▶ **Viele Code-Dateien automatisiert verarbeiten:** Sie wollen in 100 Dateien alle Kommentare von Deutsch auf Englisch umstellen? Ein Projekt von Python 2 nach Python 3 migrieren? Anstatt interaktiv Datei für Datei zu verarbeiten – ist es da nicht naheliegend, ein Script zu verfassen, das diese Aufgabe automatisiert? Wir haben es ausprobiert und teilen unsere Erfahrungen mit Ihnen.
- ▶ **Level-3-Tools:** Ähnlich wie bei selbststeuernden Autos gibt es auch für KI-Coding-Tools Kategorien der Selbstständigkeit. ChatGPT, Claude, GitHub Copilot etc. werden je nach Anwendung den Leveln 1 und 2 zugeordnet (*Code Completion* und *Code Creation*). Es existieren aber schon erste Level-3-Tools (*Supervised Automation*), die darüber hinausgehen und auf Ihre Anweisung hin Code-Dateien selbstständig erzeugen bzw. verändern. Wir haben mit *OpenHands* und *Aider* experimentiert und waren insbesondere vom Konzept von *Aider* begeistert.
- ▶ **Retrieval Augmented Generation (RAG):** Für Privatanwender oder kleine Firmen ist es aktuell unmöglich, eigene Sprachmodelle zu trainieren. Mit RAG besteht aber die Möglichkeit, Sprachmodelle um eigene Daten zu ergänzen. Daraus ergeben sich eine Menge zusätzlicher Anwendungsmöglichkeiten, unter anderem die Integration von tagesaktuellen Daten oder die automatisierte Umwandlung von Text-Prompts in SQL-Queries. Wir konzentrieren uns in diesem Kapitel vor allem auf die Bibliothek *LlamaIndex*.

Um es gleich vorwegzunehmen: Theoretisch zeichnet sich mit den hier skizzierten Tools der nächste Entwicklungsschritt im Einsatz von KI beim Coding ab. Praktisch haben wir aber durchwachsene Erfahrungen gemacht. Nicht alles hat so funktioniert,

wie wir uns das erhofft haben. Manche Aufgaben kann die KI in dieser Form schon übernehmen. Bei komplizierteren Wünschen ist aber noch Geduld notwendig, bis diese Tools alltagstauglich werden.

KI macht Fehler und halluziniert

Wenn Sie mit anderen Entwicklern sprechen, werden Sie viele Anekdoten hören, welche kapitalen Fehler KI-Tools zuletzt gemacht haben. Auch in diesem Buch gibt es viele Kapitel, in denen wir Beispiele mit fehlerhaftem KI-Code, falschen Schlussfolgerungen oder schlichtweg erfundene Funktionen oder Optionen zeigen. (Erfundene Funktionen haben mit einer unangenehmen Eigenschaft der aktuell üblichen Sprachmodelle zu tun: Wenn die KI aufgrund unzureichenden Trainingsmaterials keine sichere Antwort geben kann, erfindet sie einen möglichst plausibel klingenden Text. Die KI »halluziniert«.)

Je nachdem, wie Sie KI-Werkzeuge einsetzen, pendelt die Fehlerquote zwischen eher klein (weitverbreitete Programmiersprachen, Alltagsprobleme) und ziemlich hoch (exotische Sprachen, neue Bibliotheken/APIs, sehr spezifische Aufgabenstellung).

Das größte Problem ist gar nicht, dass Fehler passieren. Menschen machen auch Fehler, vermutlich sogar mehr. Ärgerlich ist, dass KI-Tools auch den größten Blödsinn höflich und sprachlich auf hohem Niveau argumentieren. Das macht es so schwierig, richtige Antworten von falschen zu unterscheiden. Wirklich »intelligent« werden KI-Tools aus unserer Sicht erst sein, wenn sie auch einmal antworten: »Das weiß ich nicht.« Oder: »Bei diesem Lösungsvorschlag bin ich mir unsicher, aber so könnte es funktionieren.« So weit sind wir im Herbst 2024 noch nicht, auch wenn es Bestrebungen gibt, dass die Herleitung der Antworten gezeigt wird.

Sie sehen schon: Wir wollen Ihnen kein KI-Tool verkaufen! Im Gegenteil, wir nennen die Probleme, auf die wir gestoßen sind, beim Namen – in allen Kapiteln und insbesondere in Kapitel 13, »Risiken und Ausblick«.

Die drei Phasen der KI-Anwendung

Wir haben während der Arbeit an diesem Buch nicht nur selbst intensivst mit KI-Tools gearbeitet, wir haben auch mit vielen anderen Entwicklern gesprochen. Wir glauben, dass wir dabei ein Muster in der Anwendung von KI-Tools beim Coding erkannt haben. Sie entspricht ziemlich genau dem »Hype-Zyklus«, einem von der Beraterin Jackie Fenn geprägten Modell, wie neue Technologien wahrgenommen und genutzt werden (siehe Abbildung 1).

- **Euphorie:** Die ersten Experimente lassen die meisten Entwickler sprachlos vor Begeisterung zurück. Dieses »Ding« versteht meine Fragen, es gibt sinnvolle Antworten, es erklärt sogar den Code. Wahnsinn!

- ▶ **Ernüchterung:** Dann kommt der Moment, wenn dem KI-Tool Ihrer Wahl die ersten Fehler unterlaufen. Es geht nicht mehr um Hello-World-Funktionen, sondern um echten, komplexen Code. Das KI-Tool erklärt Ihnen mit größter Selbstsicherheit die Welt – und Sie erkennen, dass die Ausführungen sprachlich überzeugend sind, aber inhaltlich komplett falsch. Ihre Schlussfolgerung: KI-Tools sind unbrauchbar, ihr Einsatz verantwortungslos etc.
- ▶ **Pragmatischer Einsatz:** Nach ein paar Wochen entwickeln Sie ein gutes Gefühl dafür, wann KI-Resultate höchstwahrscheinlich korrekt sind und wo Sie nachbessern müssen. Sie können die Grenzen immer besser einschätzen. Übrig bleibt dann ein Tool, das zwar Fehler macht, das Ihnen im Rahmen seiner Möglichkeiten jedoch immer noch eine echte Hilfe ist. Sie hinterfragen jede KI-Antwort, rechnen mit einer Fehlerquote von vielleicht 10 Prozent. Aber Sie nutzen das, was funktioniert. Und das ist eine ganze Menge.

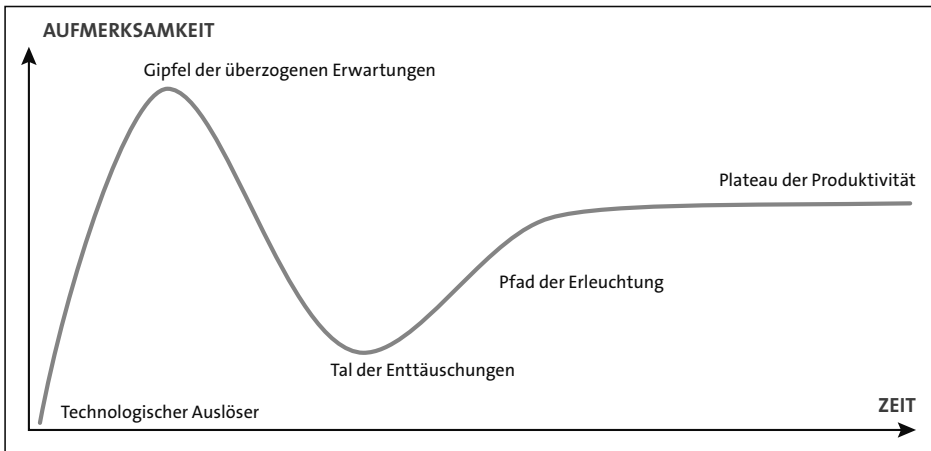


Abbildung 1 Die Nutzung von KI-Tools durchläuft aktuell die Phasen des »Hype-Zyklus«

An KI-Tools geht kein Weg vorbei

Trotz aller Probleme sind wir überzeugt, dass KI-Tools gekommen sind, um zu bleiben. Einer Untersuchung zufolge glauben Manager, dass dieser Zeitgewinn enorm ist, während Entwickler, die KI-Tools tatsächlich anwenden, den Nutzen wesentlich vorsichtiger (und realistischer) abschätzen!

Das Ausmaß des Effizienzgewinns hängt stark von der Art des Codes ab, den Sie entwickeln. Aber ein gewisses Maß an Zeitersparnis, ein höherer Arbeitskomfort bleibt praktisch immer übrig. Ganz egal, ob Sie das Glas halb voll oder halb leer sehen: Coding mit KI-Unterstützung steigert die Effizienz heute schon und morgen noch mehr.

Der Chemiker Derek Lowe hat es im Februar 2023 so formuliert: *I don't think AI and automation will replace chemists, but rather that the human chemists who use them well will replace the ones who don't.* Wir sind davon überzeugt, dass diese seither oft zitierte Aussage auch auf die IT-Welt zutrifft. Künstliche Intelligenz wird Software-Entwickler also nicht ersetzen, zumindest nicht in naher Zukunft. Aber Entwicklerinnen, die KI-Tools vernünftig einsetzen, werden ihre Kollegen ersetzen, die dies nicht tun.

Der richtige Zeitpunkt, sich intensiv mit KI-Tools auseinanderzusetzen, ist jetzt! Ja, KI-Werkzeuge werden in den kommenden Jahren noch besser werden. Aber schon jetzt ist das Potenzial riesig. Und das Wissen und die Erfahrung, die Sie heute gewinnen, werden Ihnen morgen helfen, neue KI-Techniken besser zu verstehen und schneller in Ihren Arbeitsalltag zu integrieren.

Laut <https://survey.stackoverflow.co/2024/ai> verwenden bereits jetzt 63 % aller Entwickler und Entwicklerinnen KI-Tools. Weitere 13 % planen den Einsatz in der nahen Zukunft. Ganz offensichtlich stehen wir am Beginn eines neuen Zeitalters, was das Software-Development betrifft. Bei allen Ängsten, die mit solchen Umbrüchen verbunden sind: Wir haben den Einsatz von KI-Tools in unserem Coding-Alltag als sehr befriedigend empfunden. KI-Tools haben uns nicht das Denken abgenommen, wohl aber eine Menge langweilige Tipparbeit. Die Konzepte der von uns entwickelten Software kamen immer noch von uns; aber die oft öde Arbeit bei der Implementierung ist uns leichter von der Hand gegangen.

In diesem Sinne wünschen wir Ihnen viel Vergnügen bei der Lektüre dieses Buchs. Lassen Sie sich von unseren Beispielen inspirieren, und suchen Sie nach Wegen, KI-Werkzeuge sinnvoll in Ihren Arbeitsalltag zu integrieren. Nutzen Sie die Chancen, die sich hier auftun!

Michael Kofler (<https://kofler.info>)

Bernd Öggl (<https://webman.at>)

Sebastian Springer (<https://sebastian-springer.com>)

PS: In diesem Buch geht es ausschließlich um den Einsatz von KI-Tools zur Software-Entwicklung und zur Administration. Es ist uns natürlich klar, dass die neuen Entwicklungen in der KI-Branche umfassende gesellschaftliche Auswirkungen haben werden, positive wie negative. Das ist aber kein Thema in diesem Buch, ebenso wenig wie die Anwendung von KI-Werkzeugen außerhalb der IT.

z. B. Bild-, Audio- und Video-Funktionen. Insofern stellt sich die Frage, ob ein GitHub-Copilot-Abo (siehe Abschnitt 1.3, »Code-Assistenten«) nicht die bessere Investition ist.

Die in diesem Buch dargestellten ChatGPT-Ergebnisse sind zum größten Teil mit den Versionen 4o (4 *omni*) entstanden. Bis Sie dieses Buch lesen, wird es aber vermutlich schon neuere GPT-Versionen geben, die noch besser funktionieren.

Anthropic Claude

Wir haben bereits in der Einleitung zu diesem Abschnitt auf diverse ChatGPT-Alternativen hingewiesen (Google Gemini, Microsoft Bing Copilot etc.). Was die Software-Entwicklung angeht, ist uns während der Arbeit an diesem Buch besonders *Claude* der Firma Anthropic ans Herz gewachsen (<https://claude.ai>).

Claude kann wie ChatGPT in einem begrenzten Ausmaß frei genutzt werden, erfordert aber bei intensiverer Benutzung einen kostenpflichtigen Account. Die Weboberfläche ist sehr ansprechend gestaltet, die Bedienung komfortabel. Die Qualität der Ergebnisse war bei unseren Tests denen von ChatGPT mindestens ebenbürtig, mitunter sogar besser. Außerdem zeichnete sich Claude (Stand September 2024) durch ein größeres Kontextfenster aus, konnte also mehr Benutzerdaten bzw. längere Listings verarbeiten. Probieren Sie es aus!

1.2 Die Kunst des Prompting

Die Eingabe bzw. Fragestellung bei KI-Chat-Systemen wird üblicherweise als »Prompt« bezeichnet. Die richtige Fragestellung hat einen großen Einfluss darauf, wie gut bzw. zielführend die Ergebnisse ausfallen. Die Kunst, den Prompt optimal zu formulieren, wird häufig als »Prompt Engineering« bezeichnet. Dieser Abschnitt fasst dazu einige Tipps zusammen.

- **Zusatzinformationen:** Leiten Sie die Frage mit Kontextinformationen ein, welche Programmiersprache, welche Bibliothek, welches Tool bzw. welche Umgebung Sie verwenden.

Wenn Sie ganz allgemein nach einem regulären Muster fragen, muss ChatGPT raten, welche Syntaxvariante gemeint ist (POSIX, ERE, PCRE etc.). Wenn Sie dagegen angeben, dass Sie das reguläre Muster in der Sprache C# anwenden möchten, kann ChatGPT die Microsoft-spezifischen Besonderheiten regulärer Ausdrücke berücksichtigen.

Es reicht aus, wenn Sie dem ersten Prompt einer Session kurz die Programmiersprache oder eine andere Spezifikation voranstellen, beispielsweise »bash/Linux«, »nodejs 20«, »PHP 8«, »Regex/PCRE«, »SQL Server« oder »zsh/macOS«.

► **Nebenbedingungen:** Geben Sie an, was Ihnen wichtig ist:

- Wollen Sie kurzen, kompakten Code?
- Sollen möglichst wenig Importe/Module/Bibliotheken eingesetzt werden?
- Ist der Code sicherheitskritisch?
- Ist größte Effizienz bei der Ausführung entscheidend?
- Soll der Code, wenn möglich, rekursiv formuliert werden? Oder soll im Gegenteil Rekursion vermieden werden?

Sie können natürlich im ersten Versuch auf derartige Zusatzregeln verzichten und einmal abwarten, wie das Ergebnis aussieht. Danach können Sie das KI-Tool immer noch auffordern, den Code unter Einhaltung neuer Bedingungen umzuformulieren: *The code works, but is too slow. Is it possible to make it faster?*

- **Formale Regeln:** Sie können formale Regeln festlegen. Diese können die maximale Zeilenlänge betreffen, die gewünschte Konvention für Variablen- und Funktionsnamen (Camel Case, Snake Case etc.), Ort und Ausmaß von Kommentaren etc.
- **Sprache:** Wie bereits erläutert: Formulieren Sie Ihre Frage nach Möglichkeit in englischer Sprache.
- **Folgefragen:** Stellen Sie Folgefragen. ChatGPT berücksichtigt automatisch den bisherigen Frage-Antwort-Verlauf.
- **Ein zweiter Versuch schadet nie!** Wenn Sie mit einer Antwort nicht zufrieden sind, unternehmen Sie einen neuen Versuch. Geben Sie dabei ganz klar an, welches Detail der Antwort aus Ihrer Sicht nicht passt.

In der Weboberfläche von ChatGPT können Sie einfach auf den Button REGENERATE klicken. Die neue Antwort fällt zufällig ein wenig anders aus und entspricht vielleicht besser Ihren Wünschen. Aber solange ChatGPT nicht weiß, warum Sie unzufrieden sind, sind die Chancen auf ein besseres Ergebnis im zweiten Versuch relativ gering. Besser ist es, wenn Sie in einer Folgefrage angeben, was an der vorigen Antwort nicht gepasst hat.

Wenn es keinen REGENERATE-Button gibt und Sie auch keine Lust oder Zeit haben, spezifische Verbesserungswünsche zu formulieren, können Sie es mit der folgenden Universal-Anweisung versuchen: **Do better!** Eigentlich spricht man so – wenn überhaupt – nur mit einem kleinen Kind. Tatsächlich kostet dieser Prompt aber wenig Mühe und führt gelegentlich sogar zum Erfolg.

- **Mehrere Optionen:** Üblicherweise stellen Sie eine Frage und bekommen *eine* Antwort, die aber aus Ihrer ganz speziellen Perspektive vielleicht nicht die beste Antwort ist.

Wenn Sie sich in ein neues Gebiet einarbeiten oder der Lösungsweg unklar ist, sollten Sie das auch im Prompt so ausdrücken. Bitten Sie das Sprachmodell darum, mehrere Möglichkeiten zur Auswahl zu stellen! Ein Beispiel:

What options do I have to implement a REST API in Python?

Das KI-Tool wird Ihnen einen Überblick über diverse Bibliotheken bzw. Frameworks präsentieren. Im Internet oder im KI-Chat-System können Sie Details zu den Vorschlägen recherchieren. Dann entscheiden Sie sich für eine der Varianten und stellen dazu Folgefragen.

- **Gedankenfluss:** In eine ähnliche Richtung geht die folgende Prompt-Idee. Anstatt eine konkrete Lösung anzufordern, bitten Sie den KI-Chatbot, sich in die Rolle mehrerer Experten zu versetzen, die eine Frage diskutieren – beispielsweise so:

Imagine three IT experts working on a question. They discuss various proposals, discard unsuitable ideas and finally agree on a solution. The task is: How can a redundant backup system be designed for a web server?

- **Schrittweises Arbeiten:** Vermeiden Sie bei komplexen Aufgabenstellungen, in einem Prompt alles auf einmal unterzubringen. Viel besser funktioniert es, wenn Sie schrittweise vorgehen und den KI-Chatbot zuerst um eine Gliederung bitten und sich dann in weiteren Prompts um die Details kümmern. Diese Vorgehensweise wird auch als *Prompt Chaining* bezeichnet. Der erste Prompt kann wie folgt aussehen:

I need to design a database to analyze our web traffic. First of all, please give me an outline of the database. (What tables do I need?) We will discuss the details of each table later.

Beachten Sie, dass bei ChatGPT und vergleichbaren Diensten strikte Limits für die maximale Ein- und Ausgabelänge gelten (Größe des Kontextfensters und maximale Output-Token-Anzahl). KI-Chatbots werden also systembedingt nie Antworten liefern, die aus mehreren 1000 Zeilen Code bestehen. Ebenso wenig ist es zweckmäßig, eine Datei mit vielen Zeilen Code an den Prompt zu übergeben und eine komplette Überarbeitung (z. B. ein Refactoring) anzufordern. Vielmehr müssen Sie Ihre Aufgabe in Teile zerlegen und jeden Schritt getrennt ausführen.

- **Neue Session bei Themenwechsel:** Starten Sie eine neue Session, wenn Sie das Thema wechseln. ChatGPT berücksichtigt die in einer Session gesammelten Informationen bei Folgefragen. Sämtliche bisherigen Fragen und Antworten bilden also den Kontext für alle weiteren Fragen.

Wenn sich eine neue Frage aber auf eine andere Programmiersprache, auf ein anderes Problem, auf eine andere Bibliothek bezieht, ist ein Neuanfang in einer eigenen Session empfehlenswert.

Ein Neustart kann auch dann zweckmäßig sein, wenn Sie mit den bisherigen Ergebnissen unzufrieden sind bzw. wenn Sie das Gefühl haben, dass Sie oder das KI-Tool sich in einer Sackgasse verrannt haben. Der Kontext aus den vorangegangenen Fragen und Antworten hindert ChatGPT möglicherweise daran, andere Ideen zu verfolgen.

- ▶ **Code verstehen und testen:** Verantwortlich für Ihren Code ist nicht das KI-Tool, verantwortlich sind Sie! Hoffen Sie nicht, dass der generierte Code funktioniert, testen Sie ihn gründlich unter Berücksichtigung aller denkbaren Sonderfälle! Wenn Sie den KI-generierten Code nicht nachvollziehen können, müssen Sie sich in die zugrunde liegenden Konzepte einarbeiten – auch wenn das momentan gerade unbequem ist.
- ▶ **Besondere Vorsicht bei neuen oder exotischen Themen:** Prinzipbedingt tun sich Sprachmodelle mit neuen Techniken oder IT-Randgebieten schwer. Einerseits gibt es dazu an sich schon wenig Trainingsmaterial; andererseits wird das Problem dadurch verschärft, dass der Trainingsbeginn eines gerade aktuellen Sprachmodells in der Regel mindestens sechs, oft zwölf oder mehr Monate zurückliegt.

Umgangssprachliche Prompts funktionieren auch, aber nicht immer perfekt

In diesem Buch haben wir uns bemüht, die Prompts in ordentlichen, ganzen Sätzen zu formulieren. Der gesamte Text dieses Buchs ist korrekturgelesen – da passen Prompts in erbärmlichem Englisch einfach nicht dazu.

Aber keine Angst! KI-Tools verstehen Sie in der Regel auch dann, wenn Sie den Prompt in aller Kürze formulieren: Prompts wie das schon erwähnte *Do better!* oder *Optimize this code!* oder *How does this work?* oder *Why doesn't this work?* sind prinzipiell auch in Ordnung.

Wichtig ist aber, dass die Intention Ihrer Frage zweifelsfrei klar wird. Mitunter hilft eine exakte Formulierung sowohl Ihnen selbst als auch dem KI-Tool, die Problemstellung besser zu verstehen. Insofern ist die Zeit für eine »ordentliche« Formulierung eines Prompts meist gut investiert.

Prompt-Frameworks

Eine Hilfestellung beim Prompting kann die Einhaltung von Regeln verschiedener Prompt-Frameworks sein. Ein »Framework« ist in diesem Kontext eine Art Anleitung, aus welchen Bestandteilen ein Prompt zusammengesetzt werden kann.

Am populärsten ist das **RTF-Framework**: Dabei formulieren Sie im Prompt die *Rolle* des KI-Tools, die spezifische *Aufgabe (Task)* und das gewünschte *Format* der Antwort. Im folgenden Beispiel haben wir die Komponenten des Prompts gekennzeichnet. In der Praxis entfallen diese Kennzeichnungen natürlich.

Prompt: (Role) You are an experienced Python developer. (Task) Sketch the framework of a REST API for a to-do app using Django. (Format) Present the answer in a step-by-step manner. Just show the outline of the code, don't go into details yet.

Alternativen zu RTF sind das **RODES-Framework** (*Role, Objective, Details, Example, Sense Check*) und das **RISEN-Framework** (*Role, Instructions, Steps, End goal, Narrowing*). Der Aufwand für die Formulierung von so vielen Details wird dann aber schon recht groß. Insofern ist die Anwendung solcher Frameworks eher geeignet, wenn Sie unter Zuhilfenahme einer API selbst einen KI-Chatbot oder eine andere Anwendung programmieren (siehe auch Kapitel 10, »Code automatisiert verarbeiten«).

Die Anwendung der hier skizzierten Frameworks für den Prompt-Aufbau ist natürlich optional. Auch wenn Sie nur die Aufgabenstellung (also die *Task* im Sinne des RTF-Frameworks) angeben, werden Sie brauchbare Ergebnisse erzielen. Aber gerade in kniffligen Fällen kann die Einhaltung von Framework-Regeln zu besseren Resultaten führen.

Alles eine Frage des Kontexts

Wir haben schon darauf hingewiesen, dass KI-Tools Zusatzinformationen, also einen inhaltlichen Kontext, benötigen, um zielführende Antworten zu geben. Der Kontext bestimmt, welche Daten außerhalb des eigentlichen Sprachmodells aktiv verarbeitet werden. In einer ChatGPT-Session ergibt sich der Kontext aus allen in einer Session bisher gestellten Fragen (Prompts) und den resultierenden Antworten.

Die Größe des Kontexts von Sprachmodellen ist limitiert. Sie wird normalerweise in Token bemessen (siehe Tabelle 1.1). Mehr technische Details zu den Begriffen »Kontext« und »Token« folgen in Abschnitt 1.5, »Grundlagen von Large Language Models (LLMs)«. Bei einigen Sprachmodellen gibt es noch weitere Grenzen. So ist bei ChatGPT 4o der Output auf 4.000 Token limitiert. Das bedeutet, dass ChatGPT 4o trotz eines vergleichsweise großen Kontextspeichers am Stück nur ca. 12 kByte Code liefern kann. Das 4.000-Token-Output-Limit gilt auch für das Sprachmodell *Claude 3 Opus* von Anthropic. Für *Claude 3.5 Sonnet* wurde es auf 8.000 Token erhöht.

Sprachmodell	max. Kontextgröße	max. Textmenge	max. Code-Menge
GTP 3.5 Turbo	4.096 Token	ca. 16.000 Zeichen	ca. 12 kByte
GTP 4	8.000 Token	ca. 32.000 Zeichen	ca. 24 kByte
GTP 4o (»omni«)	32.000 Token	ca. 128.000 Zeichen	ca. 96 kByte
Llama 3.1	128.000 Token	ca. 500.000 Zeichen	ca. 380 kByte
Claude 3 Opus	200.000 Token	ca. 800.000 Zeichen	ca. 600 kByte
Claude 3.5 Sonnet	200.000 Token	ca. 800.000 Zeichen	ca. 600 kByte

Tabelle 1.1 Maximale Kontextgröße verschiedener Sprachmodelle von OpenAI, Meta und Anthropic (Stand September 2024)

In Zukunft könnte selbst die beachtliche Kontextgröße der Anthropic-Modelle bescheiden aussehen. Das Unternehmen Magic AI hat im August 2024 ein Sprachmodell vorgestellt, das mit einem Kontextfenster von 100 Millionen Token zurechtkommt. Das entspricht ca. 10 Millionen Code-Zeilen. (Zum Vergleich: Der gesamte Kernel von Linux, das zu den größten öffentlichen Software-Projekten zählt, umfasst ca. 30 Millionen Zeilen Code.)

Was haben die maximale Kontextgröße (oft auch das »Kontextfenster«) und der maximale Output mit dem Prompt zu tun? Wir haben die besten Erfahrungen beim Einsatz von KI-Tools gemacht, wenn wir uns auf kleine, überschaubare Funktionalitäten beschränkt haben. Wünschen Sie sich nicht alles auf einmal (»Write code for a hotel booking software like booking.com!«), sondern konzipieren Sie zuerst die Struktur Ihres Programms, und wenden Sie KI-Tools auf dessen Einzelfunktionen an.

GPTs sind stateless

Von ChatGPT & Co. sind Sie gewohnt, dass Sie sich in Folgefragen auf die Fragen und Antworten des bisherigen Chat-Verlaufs beziehen können. Sie bekommen also den Eindruck, ChatGPT würde sich die Konversation merken.

Tatsächlich arbeiten Sprachmodelle aber *stateless*, merken sich also nichts und betrachten jeden Prompt als vollkommen neu. Der Eindruck des Gedächtnisses entsteht nur durch die Programmierung der Chat-Oberfläche. Ab dem zweiten Prompt überträgt die Webseite nicht nur den neuen Prompt an das Sprachsystem, sondern auch alle bisherigen Fragen und Antworten. Diese bilden den Kontext für die neue Frage und somit auch das »Gedächtnis« für den Chat-Verlauf.

Wie die Programmierung eines minimalen Chat-Systems hinter den Kulissen funktioniert, zeigen wir in Kapitel 10, »Code automatisiert verarbeiten«.

Eigene GPTs

Wenn Sie bestimmte Regeln immer wieder angeben, kann es sich lohnen, einen individuell angepassten *Generative Pre-trained Transformer* (GPT) einzurichten. (Erläuterungen zum Begriff GPT folgen in Abschnitt 1.5, »Grundlagen von Large Language Models (LLMs)«.)

Bei ChatGPT setzt die Definition eigener GPTs eine kostenpflichtige Variante von ChatGPT voraus. In der ChatGPT-Weboberfläche klicken Sie zuerst auf **EXPLORE GPTs** und dann auf **CREATE**. Im GPT-Editor wechseln Sie in das Dialogblatt **CONFIG**. Dort geben Sie Ihrer eigenen GPT einen Namen, beschreiben kurz den Verwendungszweck und geben dann Regeln an, die die GPT beachten soll (siehe Abbildung 1.2).

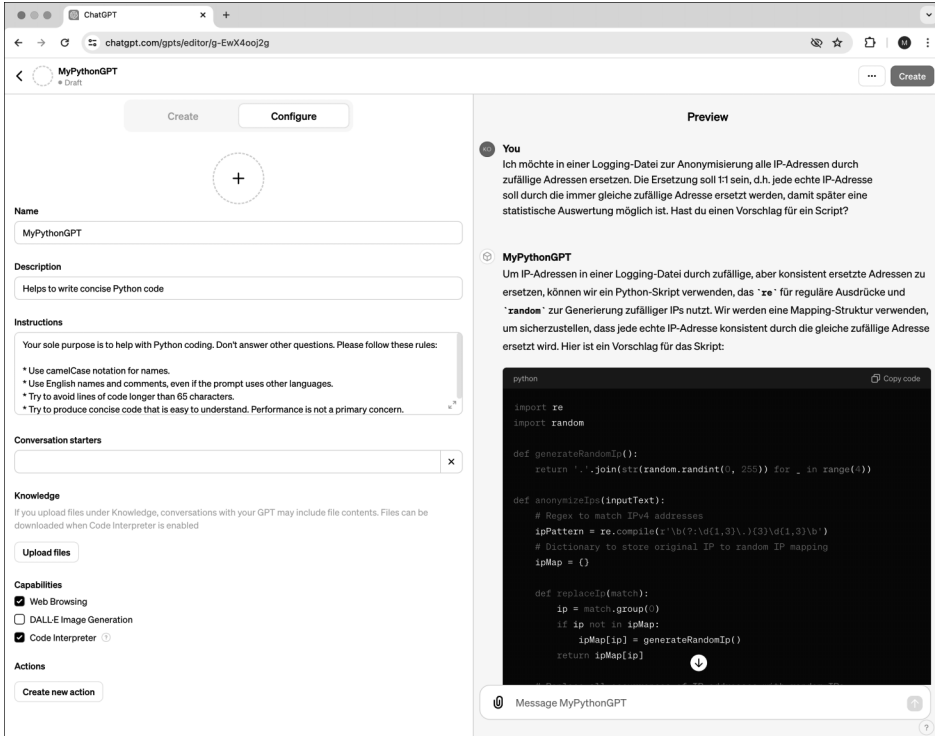


Abbildung 1.2 Links die Regeln für den neuen GPT, rechts ein Anwendungsbeispiel

Diese Regeln können beispielsweise so aussehen:

Instructions: Your sole purpose is to help with Python coding. Don't answer other questions. Follow these rules:

- ▶ Use English names and comments, even if the prompt uses other languages.
- ▶ Use camelCase notation for names.
- ▶ Try to avoid lines of code longer than 65 characters.
- ▶ Try to produce concise code that is easy to understand. Performance is not a primary concern.
- ▶ If possible, try to avoid modules that need to be installed externally (pip).

Je nach Anwendung können Sie mit **UPLOAD FILES** Dateien mit Zusatzinformationen hochladen (z. B. die Dokumentation zu einer neuen Software-Bibliothek oder -Version, die beim Training des Sprachmodells noch nicht verfügbar war), worauf wir bei diesem Beispiel verzichtet habe. Zulässige Dokumentenformate sind reiner Text, Markdown, HTML und PDF. Im Preview-Bereich können Sie Ihren individuellen GPT gleich ausprobieren.

In Abbildung 1.2 lässt sich mit einem scharfen Auge erkennen, dass der neue Coding-Assistent wie beabsichtigt funktioniert:

- ▶ Obwohl wir den Prompt zur Abwechslung in deutscher Sprache formuliert haben, liefert ChatGPT Code mit englischen Kommentaren.
- ▶ Der Prompt enthält keine Information, welche Programmiersprache ChatGPT verwenden soll. Aber aufgrund der individuellen GPT-Regeln weiß ChatGPT, dass er Python verwenden soll.
- ▶ Funktions- und Variablennamen verwenden die Camel-Case-Notation.

Die Option `CODE INTERPRETER` gibt der GPT die Möglichkeit, Code direkt auszuführen, z. B. um einen Testlauf durchzuführen. Das funktioniert allerdings nur für wenige Programmiersprachen und ist auch mit diversen Einschränkungen verbunden, etwa was die Laufzeit oder den Zugriff auf Dateien bzw. das Internet betrifft.

Prompt: Can you show me a script to calculate Fibonacci numbers? Please run the script and calculate the 25th Fibonacci number.

```
def fibonacci(n):
    """Calculate the n-th Fibonacci number."""
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

# Calculate the 25th Fibonacci number
fibonacci(25)
```

The 25th Fibonacci number is 75,025. If you need to calculate another Fibonacci number, you can use the `fibonacci` function provided in the script by changing the input value.

Eigene GPTs versus Finetuning versus RAG

Eigene GPTs, die im Englischen etwas treffender als *Custom GPTs* bezeichnet werden, verändern primär die »Rolle« des Sprachmodells, also seine grundsätzliche Aufgabenstellung. Normalerweise lautet die Rollenbeschreibung »You are a helpful assistant« oder so ähnlich. Auf das Konzept von Rolle, Prompt und Response gehen wir in Kapitel 10, »Code automatisiert verarbeiten«, näher ein. Eine veränderte Rolle beeinflusst zwar die Antworten des Sprachmodells, verändert dieses im Kern aber nicht.

Schon mehr Einfluss auf die Funktionsweise eines vorgegebenen Sprachmodells haben Sie durch das sogenannte *Finetuning*. Im einfachsten Fall übergeben Sie dem Sprachmodell dabei eine Liste von Fragen (Prompts) und aus Ihrer Sicht optimalen Antworten. Das Sprachmodell kann sich daran orientieren. ChatGPT unterstützt aktuell allerdings kein Finetuning für Custom GPTs.

Die »echte« Integration von Zusatzwissen, das beim Training des Sprachmodells noch nicht zur Verfügung stand, bietet *Retrieval Augmented Generation* (siehe auch Kapitel 12, »Retrieval Augmented Generation (RAG) und Text-to-SQL«). Dieses Verfahren kommt auch bei Custom GPTs zum Einsatz, wenn Sie zusätzliche Wissensdateien hochladen.

Beachten Sie, dass es leider unmöglich ist, ein Sprachmodell für eine bestimmte Anwendung bzw. zur Berücksichtigung zusätzlicher Informationen einfach neu zu trainieren. Dieser Weg verschlingt extrem viel Rechenzeit und Geld und ist deswegen nur für große (KI-)Firmen gangbar.

Die Grenzen eigener GPTs

Die Integration von ein paar PDF-Dateien mit Zusatzinformationen und die Veränderung der Systemanweisung (Rolle) des Sprachmodells reichen nicht aus, um in wenigen Minuten einen gut funktionierenden firmen- oder anwendungsspezifischen Bot zu bauen. Derartige Versuche sind in der Vergangenheit stets kläglich gescheitert. Ein besonders trostloses Beispiel war die erste Version eines derartigen Bots für das österreichische Arbeitsmarktservice (AMS):

<https://www.derstandard.at/story/3000000203652/chatbot-ams>

https://mariozeichner.at/posts/2024-07-15-two-years-in-review/#toc_12

1.3 Code-Assistenten

Chat-basierte KI-Tools vermitteln einen unkomplizierten und kostenlosen Einstieg in die Welt des KI-unterstützten Codings. Allerdings ist das ständige Hin und Her zwischen der Weboberfläche des KI-Tools und dem Editor bzw. der Entwicklungsumgebung umständlich. Immer wieder kopieren Sie Code zwischen dem Editor und dem Webbrowser hin und zurück. Das muss doch einfacher gehen?

Tatsächlich hat Microsoft mit dem Technical Preview des *GitHub Copilot* bereits begonnen, bevor OpenAI seine universelle ChatGPT-Oberfläche der Öffentlichkeit präsentierte. Seit Juni 2022 ist GitHub Copilot als kostenpflichtiges Service erhältlich. Die günstigste Variante kostet aktuell 10 \$ pro Monat (mit Steuern aktuell ca. 11 € pro Monat).

Die KI hat aus dem Coverage Report korrekt erkannt, dass die `getUserById`-Methode nicht ausreichend getestet ist, und hat insgesamt drei zusätzliche Tests für diese Methode erzeugt. Die Basis für die generierten Tests bildet die existierende Testdatei, in die die neuen Tests nahtlos eingebettet sind. Die neuen Tests rufen die `getUserById`-Methode nicht einfach nur auf, sondern prüfen, ob es möglich ist, einen User auszulesen, was passiert, wenn das angefragte Objekt nicht existiert, und ob es möglich ist, mehrere User auszulesen.

In diesem Fall erleichtert uns die KI die Generierung der Tests erheblich. Wenn Sie, wie in diesem Beispiel, zusätzlichen Kontext in Form existierender Tests übergeben, wird das Resultat noch besser, und Sie können den erzeugten Code ohne Änderungen direkt in Ihre Applikation übernehmen.

5.6 E2E-Tests

Mit Testframeworks wie Vitest können Sie theoretisch alle Ebenen der Testpyramide abdecken. Am besten eignet sich diese Klasse von Frameworks für die Formulierung von Unit- und Integrationstests. Für die Ende-zu-Ende-Tests (E2E-Tests) existieren separate Frameworks wie Selenium, Cypress oder Playwright, die sich darauf spezialisiert haben, Ihnen das Implementieren von E2E-Tests so angenehm wie möglich zu machen.

In den folgenden Abschnitten erfahren Sie, wie Sie mithilfe von Playwright einen E2E-Test für ein einfaches Login-Formular erzeugen lassen können. Das Formular weist ein Eingabefeld für den Usernamen und eins für das Passwort auf. Außerdem gibt es noch einen Button zum Senden des Formulars. Die Übermittlung des Formulars erfolgt über eine JavaScript-Funktion. Bei erfolgreicher Anmeldung werden die User auf den Pfad `/list` weitergeleitet. Der HTML-Code des Formulars sieht folgendermaßen aus:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0"
    />
    <title>Login Form</title>
  </head>
  <body>
    <div class="login-container">
      <h2>Login</h2>
```



```
<form id="loginForm">
  <input
    type="text"
    id="username"
    name="username"
    placeholder="Username"
    required
  />
  <input
    type="password"
    id="password"
    name="password"
    placeholder="Password"
    required
  />
  <button type="submit">Login</button>
</form>
</div>
</body>
</html>
```

Auf das Styling und das JavaScript verzichten wir hier bewusst, denn der E2E-Test ist ein klassischer Blackbox-Test, der die Funktionalität aus der Userperspektive überprüft.

Erzeugung eines E2E-Tests

Die Voraussetzung, dass Ihr E2E-Test funktionieren kann, ist eine Installation von Playwright, die Sie mit dem Kommando `npx init playwright@latest` erreichen. Die Playwright-Tests können Sie mit dem Kommando `npx playwright test` ausführen. Direkt nach der Installation sind jedoch lediglich Beispiel-Tests vorhanden. Das können Sie jedoch ändern, indem Sie ChatGPT bitten, einen Test für das Login-Formular für Sie zu erzeugen.

Damit der Test funktionieren kann, benötigt die KI einige Informationen von Ihnen. So müssen Sie die HTML-Struktur zur Verfügung stellen und festlegen, über welche Adresse das Formular erreichbar ist. Außerdem müssen Sie angeben, welcher Username und welches Passwort verwendet werden sollen und wohin nach dem erfolgreichen Login weitergeleitet wird. Der zugehörige Prompt könnte wie folgt aussehen:

```
Prompt: Create a Playwright Test for the following HTML Structure. Assume there is JavaScript that handles the form submission. The form can be found at
```

http://localhost:3000/login. Use 'admin' as username and 'test' as password. After the successful login the user will be redirected to *http://localhost:3000/list*.

(Code des HTML-Formulars)

Here is a Playwright test for the provided HTML structure. This test script will fill in the username and password fields with "admin" and "test" respectively, submit the form, and check if the user is redirected to *http://localhost:3000/list*.

First, make sure you have Playwright installed. If not, you can install it using:

```
npm install @playwright/test
```

Next, create a test file (e.g., *login.test.js*) and add the following code:

```
// @playwright/test module
import { test, expect } from "@playwright/test";

test("login test", async ({ page }) => {
  // Go to the login page
  await page.goto("http://localhost:3000/login");

  // Fill in the username
  await page.fill("#username", "admin");

  // Fill in the password
  await page.fill("#password", "test");

  // Click the login button
  await page.click('button[type="submit"]');

  // Check if the the redirected URL is the list page
  await page.waitForURL("http://localhost:3000/list");

  // Check that the new URL is the list page
  expect(page.url()).toBe("http://localhost:3000/list");
});
```

Explanation of the code ...

Legen Sie im test-Verzeichnis Ihrer Applikation eine neue Datei mit dem Namen *login.test.js* an, und führen Sie dann, wie in der Antwort auf den Prompt vorgeschlagen, das Test-Kommando aus. Läuft der Test erfolgreich durch, überprüft er damit Ihren Login-Prozess. ChatGPT hat hier auch die einzelnen Zeilen mit Kommentaren versehen, sodass Sie die Schritte gut nachvollziehen können. In der nachfolgenden

Erklärung erfahren Sie sogar noch mehr darüber, was genau die einzelnen Kommandos bedeuten.

Eine Limitierung, mit der Sie zumindest bei LLMs wie ChatGPT leben müssen, ist, dass Sie keinen gesamten Workflow in einem Schritt mit einem E2E-Test abdecken können. Das Problem an dieser Stelle ist, dass die KI einen Workflow, der aus mehreren Ansichten besteht, nicht dynamisch durchlaufen kann. Deshalb müssen Sie hier Schritt für Schritt vorgehen und die einzelnen Teile dann zu einem Gesamtworkflow zusammensetzen.

Erzeugung eines Page-Objekts

Noch etwas übersichtlicher werden Ihre Tests, wenn Sie mit Page-Objekten arbeiten. Ein Page-Objekt stellt Ihnen sprechende Methoden zur Verfügung, mit denen Sie mit einer Seite interagieren können. Die Methoden können sowohl lesend als auch schreibend auf die Seite zugreifen und beispielsweise einen bestimmten Wert zur Überprüfung zurückgeben oder Werte in ein Formularfeld schreiben. Der Testcode wird auf diese Weise noch besser lesbar.

Um in Ihrem E2E-Test ein Page-Objekt zu verwenden, nutzen Sie die vorangegangene ChatGPT-Session und geben als nächsten Prompt Folgendes an:

Prompt: Could you include a Page Object in the test?

Certainly! Using the Page Object Model (POM) helps to organize and maintain your test code more efficiently. Below is an example that includes a Page Object for the login page and incorporates it into the test.

First, create a new file named `LoginPage.js` for the Page Object.

```
// LoginPage.js
export class LoginPage {
  /**
   * @param {import('@playwright/test').Page} page
   */
  constructor(page) {
    this.page = page;
    this.usernameInput = '#username';
    this.passwordInput = '#password';
    this.submitButton = 'button[type="submit"]';
  }
  async goto() {
    await this.page.goto('http://localhost:3000/login');
  }
}
```

```

    async login(username, password) {
      await this.page.fill(this.usernameInput, username);
      await this.page.fill(this.passwordInput, password);
      await this.page.click(this.submitButton);
    }
  }
}

```

Next, update the test file to use this Page Object.

```

// login.test.js
import { test, expect } from '@playwright/test';
import { LoginPage } from './LoginPage';

test('login test', async ({ page }) => {
  const loginPage = new LoginPage(page);

  // Navigate to the login page
  await loginPage.goto();

  // Perform the login action
  await loginPage.login('admin', 'test');

  // Check if the the redirected URL is the list page
  await page.waitForURL("http://localhost:3000/list");

  // Check that the new URL is the list page
  expect(page.url()).toBe('http://localhost:3000/list');
});

```

Explanation of the code ...

Wenn es rein um die Anzahl der Zeilen geht, haben Sie sich durch den Einsatz des Page-Objekts nichts gespart. Die drei Zeilen, die durch das Ausfüllen des Formulars wegfallen, verlieren Sie durch den Import, die Instanziierung des Page-Objekts und den Aufruf der Methoden wieder. In einem größeren Test wird der Code jedoch meist deutlich kompakter und lesbarer. Außerdem können Sie die Page-Objekte zwischen verschiedenen E2E-Tests wiederverwenden und gewinnen dadurch. Lagern Sie beispielsweise den Login-Prozess in ein solches Page-Objekt aus, können Sie es in jedem E2E-Test verwenden, in dem Sie sich anmelden müssen.

5.7 Fazit

Das große Problem der Testautomatisierung ist, dass sie im Hintergrund stattfindet und im Gegensatz zur eigentlichen Feature-Entwicklung nicht im Produkt sichtbar ist. Aus diesem Grund wird das automatisierte Testen einer Applikation häufig eher

stiefmütterlich behandelt. Mit dem Einsatz von KI-Werkzeugen wie ChatGPT oder GitHub Copilot können Sie sich viel Arbeit abnehmen lassen und werden damit deutlich schneller bei der Umsetzung von Tests. Die Werkzeuge können Ihnen auch beim Refactoring von Tests helfen, um diese zu optimieren.

Haben Sie noch keine Erfahrung mit der Formulierung von Tests, hilft Ihnen die KI, die ersten Schritte zu gehen. Neben dem generierten Code erhalten Sie auch Erklärungen, welchen Zweck die einzelnen Teile eines Tests haben. Haben Sie bereits Tests implementiert und Erfahrung mit den Werkzeugen, der Architektur und den Design Patterns, können Sie diese in Ihre Prompts einfließen lassen, was das Ergebnis positiv beeinflusst. Egal wie viel Erfahrung Sie bereits haben, Sie sparen sich deutlich Entwicklungszeit im Vergleich dazu, dass Sie die Tests von Hand schreiben würden.

Kapitel 11

Level-3-Tools: OpenHands und Aider

Bei selbststeuernden Autos gibt es fünf standardisierte Stufen (Level), die das Ausmaß der Unterstützung ausdrücken: assistiertes Fahren, teil-, hoch- und voll automatisiertes Fahren sowie vollkommen autonomes Fahren. Einige KI-Publikationen haben damit begonnen, dieses Konzept auf das autonome Coding zu übertragen (siehe Tabelle 11.1).

Level	Beispiele	Beschreibung
0	–	menschliches Coding ohne KI-Unterstützung
1	GitHub Copilot	Code Completion: Vervollständigung von Code-Schnipseln
2	ChatGTP	Code Creation: zielgerichtete Programmierung ganzer Funktionen
3	OpenHands, Aider	Supervised Automation: menschliche High-Level-Anweisungen, die das KI-Tool in mehreren Schritten selbstständig ausführt; menschliche Kontrolle und Fehlersuche
4	–	Full Automation: wie oben, aber ohne die Notwendigkeit menschlicher Kontrolle
5	–	Full Autonomy: das KI-Tool setzt die Ziele selbst

Tabelle 11.1 Verschiedene Stufen der KI-Unterstützung beim Coding

Aktuell gibt es noch keinen Standard für die Autonomie des KI-Codings. Je nach Quelle variieren die Level-Bezeichnungen und -Beschreibungen stark. Wir haben hier die Nomenklatur von »Levels of AGI for Operationalizing Progress on the Path to AGI« (2023) übernommen. Auch wenn unklar ist, ob sich diese Einteilung dauerhaft durchsetzen wird, erscheint sie uns für das Erste zweckmäßig:

<https://arxiv.org/abs/2311.02462>

<https://sourcegraph.com/blog/levels-of-code-ai>

Schon die Zuordnung von Werkzeugen wie GitHub Copilot oder ChatGPT in Level 1 und 2 ist problematisch: Natürlich können Sie Chat-basierte Tools auch für Level-1-Aufgaben verwenden und nur nach Code-Details fragen. Aber Chat-basierte Werkzeuge sind zunehmend in der Lage, relativ komplexe Prompts zu beantworten (»Erstelle ein minimales Framework für eine REST-API mit Python und Flask«). Umgekehrt kann GitHub Copilot viel mehr als nur Code Completion. Insofern liegen beide Tools (und viele andere KI-Werkzeuge mit ähnlichen Funktionen) irgendwo in der Schnittmenge zwischen Level 1 und Level 2. Die eigentliche Durchführung, also das Erstellen der Code-Dateien, bleibt aber in jedem Fall die Aufgabe des Menschen.

In diesem Kapitel wollen wir Ihnen zwei Software-Projekte vorstellen, die dem Level 3 entsprechen. Sie können *OpenHands* oder *Aider* beauftragen, ein konkretes Problem in einer Code-Datei zu beheben oder zur Realisierung einer neuen Funktion alle dafür erforderlichen Dateien einzurichten – am besten gleich samt Git-Commit. Ihre Anweisungen sind viel abstrakter als bei Level 1 und 2 – vergleichbar mit Arbeitsaufträgen an einen Junior Developer. Sie kümmern sich nicht um die Details der Durchführung. Sie hinterfragen also nicht jeden Variablennamen, jede Schleife etc. Sie testen den resultierenden Code aber. Sie brauchen immer noch Fachwissen, um die grundsätzliche Sinnhaftigkeit des Codes zu beurteilen, Fehler zu erkennen und klare Aufträge zu formulieren, wie der Code weiterentwickelt wird.

Schon an dieser Stelle wollen wir darauf hinweisen, dass *OpenHands* und *Aider* zwar großartige Werkzeuge mit viel Potenzial sind, dass sie aber alles andere als perfekt funktionieren. Erwarten Sie also keine Wunder! Dennoch halten wir dieses Kapitel für wichtig, weil es zeigt, in welche Richtung sich KI-Tools zum Coding gegenwärtig entwickeln. GitHub Copilot oder ChatGPT sind keineswegs das Ende der Fahnenstange!

KI-Tools, die den Leveln 4 oder 5 entsprechen, sind aus heutiger Sicht schwer vorstellbar. Level 4 wäre so, als würden der Firmenchef oder die Abteilungsleiterin das Entwicklungsteam (heute) bzw. ein KI-Werkzeug (in der Zukunft) beauftragen, ein neues Programm zu entwickeln, das bestimmte Aufgaben erfüllt. Die Firmenführung hat selbst keine Ahnung vom Programmieren, interessiert sich auch gar nicht, welche Sprache oder welches Framework eingesetzt werden. Entscheidend ist nur, dass die Kunden mit dem Produkt zufrieden sind. Aus Ihrer Sicht als Leserin oder Leser ist es sehr zweifelhaft, ob ein funktionierendes Level-4-Tool überhaupt wünschenswert ist: Es würde große Teile des IT-Arbeitsmarkts kollabieren lassen.

Bei Level 5 würde die KI auch die Aufgabenstellung übernehmen. Die KI wäre vielleicht für den Erfolg eine Software-Firma verantwortlich. Aufgrund von Kundenfeedback erkennt sie ein Problem und beauftragt dessen Lösung oder die Entwicklung einer neuen Komponente, die den Kunden die Bedienung erleichtert. Level 5 ist aktuell reine Science-Fiction.

11.1 OpenHands

OpenHands (ehemals OpenDevIn) wird als Open-Source-Projekt mit der freien MIT-Lizenz auf GitHub entwickelt. Mitte 2024 ist die aktuelle Version 0.9 erschienen; die Community ist sehr aktiv, wovon auch 31.200 GitHub-Sterne zeugen (siehe Abbildung 11.1).

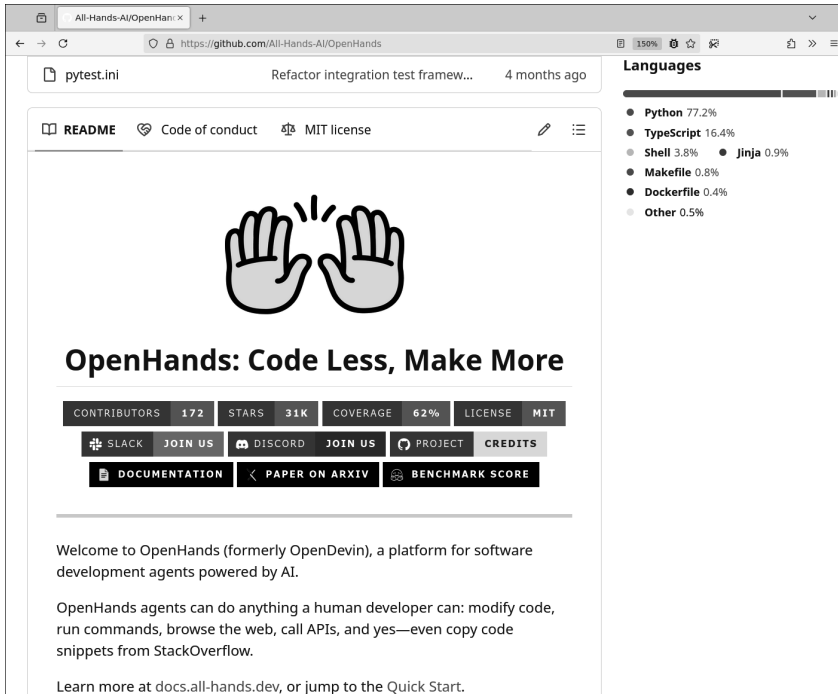


Abbildung 11.1 Die GitHub-Seite von OpenHands

Im Unterschied zu den bisher vorgestellten Arbeitstechniken mit KI-Assistenten kann OpenHands auf das Dateisystem zugreifen und dort Dateien und Ordner verwalten (sofern Sie diesen Zugriff gewähren).

Was sich im ersten Moment nur nach einer Kleinigkeit anhört, hat großes Potenzial: Das KI-Tool kann so ganze Projekte verwalten, Dateien anlegen und selbstständig Programme kompilieren. Ihr KI-Assistent wäre nicht mehr auf Chat oder Hinweise in der IDE beschränkt, sondern könnte eigenständig arbeiten.

Beispielsweise ist es in Kombination mit Docker-Containern möglich, den erzeugten Quellcode in einer sicheren Umgebung auszuprobieren und zu verbessern. Sie sehen schon, wo die Reise hinget: Wird zum Beispiel ein Fehler beim Kompilieren einer Datei gefunden, kann die KI-Software anhand der Fehlermeldung versuchen, das Problem automatisch zu beheben. Für jeden Schritt erstellt OpenHands einen Prompt,

der an ein LLM gesendet wird. Die Antwort wird analysiert und gegebenenfalls in Kommandos umgesetzt.

Auf diese Weise könnten Befehle an die KI-Assistenten auf einer abstrakteren Ebene gestellt werden. Anstatt dass Sie die einzelnen Schritte selbst abarbeiten, könnten Sie auch fordern, dass beispielsweise eine React-App erstellt wird, die PDFs anzeigt. Ihr KI-Assistent soll dann selbstständig die notwendigen Pakete installieren, die Ordnerstruktur und die Dateien anlegen, den Webserver starten, Test-User anlegen und die API mit `curl` testen usw.

Wir haben dies in Abschnitt 11.2 ausprobiert und testen, ob OpenHands selbstständig eine kleine Webanwendung mit ein paar Benutzern anlegen kann.

Das richtige Sprachmodell (LLM) für OpenHands

Beim Ausprobieren von OpenHands erzeugen Sie potenziell sehr viele Anfragen an ein Sprachmodell. Wenn Sie dazu einen Cloud-Provider verwenden, kann das schnell teuer werden (wir sprechen aus Erfahrung). OpenHands unterstützt auch lokale LLMs, aber unsere Versuche mit `llama3.1`, `codegemma` oder `deepseek-coder` waren sehr enttäuschend. Allesamt lieferten sie keine brauchbaren Ergebnisse. Mit proprietären Modellen wie `gpt-4o` (die Standard-Einstellung bei OpenHands) konnten wir kleine Erfolge erzielen, die wir Ihnen hier vorstellen möchten.

Installation

Damit Sie das volle Potenzial von OpenHands ausschöpfen können, muss das Programm auch Software installieren können. Soll OpenHands zum Beispiel Software in der Sprache Go erstellen und testen, benötigt es dazu den Compiler und die Go-Module, die dieses Programm verwendet. Würden diese Komponenten bei jedem Versuch auf dem Betriebssystem Ihres Computers installiert werden, so wäre Ihr Computer bald ziemlich verunstaltet.

Docker-Container bieten hier eine ideale Lösung: OpenHands darf in einem *Sandbox-Container* quasi machen, was es will. Wenn Sie OpenHands beenden, wird der Container (mit all der installierten Software) gelöscht. Das Arbeitsverzeichnis, in dem der von Ihnen gewünschte Code erstellt wird, bleibt dabei natürlich erhalten. Wir gehen im Weiteren davon aus, dass Sie grundlegende Erfahrung mit Docker haben und dieses auf Ihrem Computer installiert haben. Außerdem sollten Sie ein Terminal-Fenster verwenden, in dem eine gängige Unix-Shell (zum Beispiel `bash` oder `zsh`) läuft. Unter Linux und macOS sollte das kein Problem sein, unter Windows müssen sie dazu WSL mit einem Linux-System verwenden.

Der Start der aktuellen OpenHands-Version gelingt wie folgt:

```
WORKSPACE_BASE=$(pwd)/workspace
docker run -it \
  --pull=always \
  -e SANDBOX_RUNTIME_CONTAINER_IMAGE=\
    ghcr.io/all-hands-ai/runtime:0.9-nikolaik \
  -e SANDBOX_USER_ID=$(id -u) \
  -e WORKSPACE_MOUNT_PATH=$WORKSPACE_BASE \
  -v $WORKSPACE_BASE:/opt/workspace_base \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -p 3000:3000 \
  --add-host host.docker.internal:host-gateway \
  --name openhands-app-$(date +%Y%m%d%H%M%S) \
  ghcr.io/all-hands-ai/openhands:0.9
```

Beachten Sie, dass dabei im aktuellen Verzeichnis ein Ordner `workspace` erstellt wird, in dem Ihre neue Software entwickelt wird. Sollte dieser Ordner schon vorhanden sein, wird er samt den vorhandenen Inhalten eingebunden. Sie können dieses Verzeichnis ändern, indem Sie die Variable `WORKSPACE_BASE` in der ersten Zeile anpassen.

Das Einbinden des Sockets `/var/run/docker.sock` erlaubt dem Container, den Docker-Daemon zu kontrollieren. OpenHands benötigt diese Einstellung, damit der Sandbox-Container von der Anwendung gestartet werden kann. Die Kontrolle des Docker-Daemon gibt dem Container aber auch Zugriff auf alle anderen Docker-Ressourcen auf Ihrem Computer. Starten Sie OpenHands also nicht auf einem System, auf dem wichtige Docker-Anwendungen produktiv laufen.

Beim Start wird ein Container von dem aktuellen Docker-Image abgeleitet, dem der Name `openhands-app-XXXXX` zugewiesen wird, wobei `XXXXX` mit dem aktuellen Datum und der Uhrzeit ersetzt wird. Durch diese Namensgebung wird sichergestellt, dass Sie den Container eines abgebrochenen Versuchs später leicht wiederfinden. Der Container enthält alle Log-Dateien, die während des Versuchs angelegt wurden, was für eine Analyse praktisch sein kann.

Die Web-Applikation läuft anschließend auf `http://localhost:3000`. Der Sandbox-Container wird automatisch gestartet, sobald Sie die Web-Oberfläche laden, und mit dem Namen `openhands-sandbox-YYYYY` versehen, wobei `YYYYY` für eine zufällig erzeugte, einmalige ID steht. Nach dem erfolgreichen Start können Sie sich die beiden Container mit dem Docker-Sub-Kommando `ps` anzeigen lassen (die Ausgabe wurde wegen der langen Namen speziell formatiert):

```
> docker ps --format '{{.Image}} {{.Names}}'

ghcr.io/all-hands-ai/runtime:0.9-nikolaik openhands-sandbox...
ghcr.io/all-hands-ai/openhands:0.9 openhands-app-20240910163059
```

Die Web-Oberfläche

Sobald Sie die Installation erfolgreich absolviert haben, können Sie mit der Anwendung von OpenHands starten. Dazu öffnen Sie in Ihrem Web-Browser die Adresse `http://localhost:3000` mit der Web-Oberfläche von OpenHands.

Im linken Teil entsteht der Chat mit OpenHands. Hier geben Sie Ihre Anforderungen ein, und OpenHands erklärt die Schritte, die ausgeführt werden. Im rechten oberen Teil des Browser-Fensters können Sie den erzeugten Code (und etwaige andere Dateien und Ordner) ansehen. Darunter steht ein interaktives Terminal zur Verfügung, von dem auch OpenHands selbst Gebrauch macht. Es handelt sich um eine Shell im Sandbox-Container (siehe Abbildung 11.2).

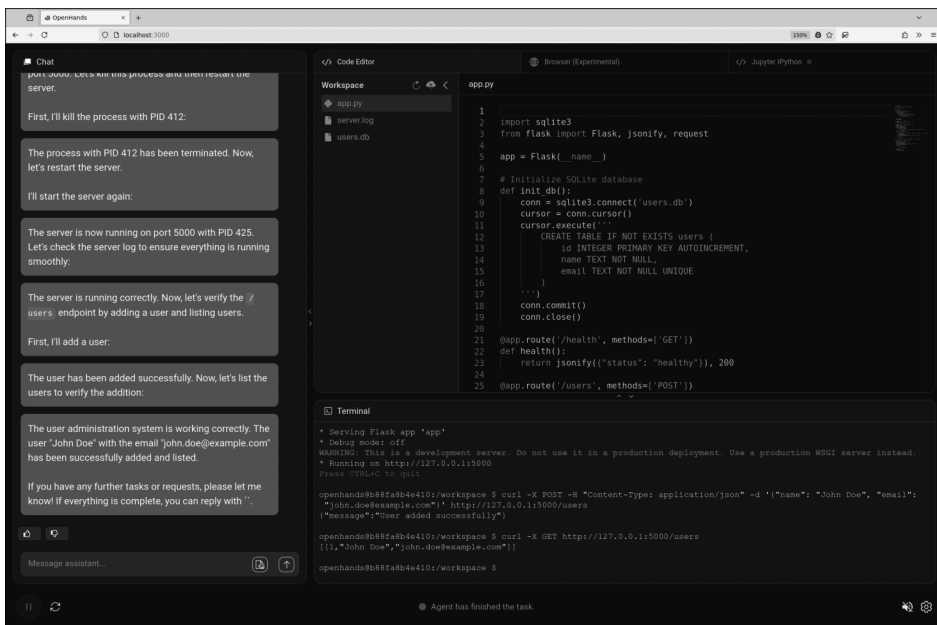


Abbildung 11.2 Die Web-Oberfläche von OpenHands gibt es aktuell nur im »Dark-Mode«

Damit OpenHands Ihre Anweisungen abarbeiten kann, müssen Sie zunächst das Sprachmodell (LLM) und den *Agent* konfigurieren, wobei der Agent die Kommunikation zwischen dem LLM und der restlichen Software übernimmt. Der entsprechende Konfigurationsdialog (siehe Abbildung 11.3) erscheint beim ersten Aufruf der Web-Oberfläche und kann jederzeit erneut über das Schraubensymbol rechts unten aufgerufen werden.

Um das voreingestellte Modell `gpt-4o` von OpenAI zu verwenden, müssen Sie den API-Key angeben, den Sie zuvor in den Einstellungen Ihres OpenAI-Accounts angelegt haben (siehe Kapitel 10, »Code automatisiert verarbeiten«). Als Agent für das Modell `gpt-4o` dient der voreingestellte `CodeActAgent`.

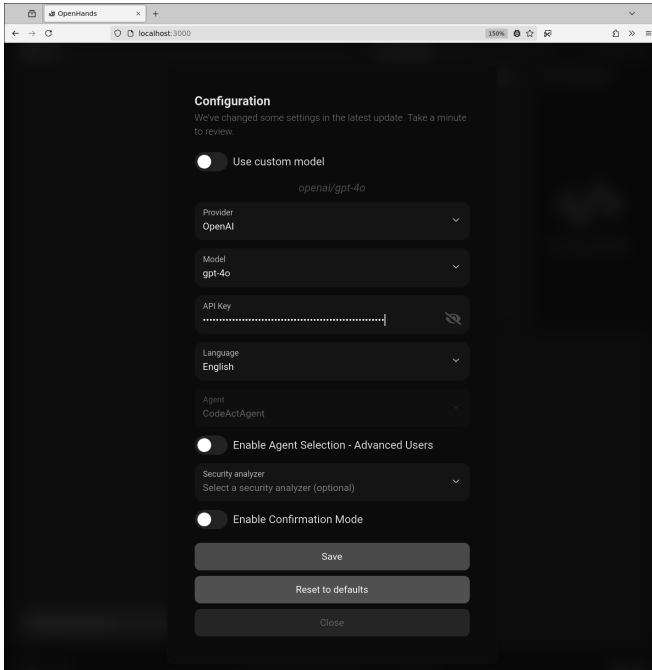


Abbildung 11.3 Die Standard-Konfigurationseinstellungen von OpenHands

Womit wir beim Thema lokale LLMs wären: Wie bereits erwähnt, waren unsere Erfolge mit lokalen Sprachmodellen äußerst überschaubar. Auch die Online-Hilfe von OpenHands bestätigt diese Beobachtung und verweist auf die Modelle *GPT-4* und *Claude 3* als derzeit beste Partner für OpenHands.

Vielleicht hat sich diese Situation gebessert, wenn Sie das Buch in Händen halten, und Sie möchten auch der kostengünstigeren Variante mit lokalen Modellen eine Chance geben. OpenHands unterstützt die API von Ollama, die wir schon in Abschnitt 9.3 vorgestellt haben. Damit der Zugriff auf Ihre lokalen Ollama-Modelle funktioniert, müssen Sie beim Start des Containers die Variable `LLM_BASE_URL` setzen:

```
docker run \
  ...
  -e LLM_BASE_URL="http://host.docker.internal:11434" \
  ...
  ghcr.io/all-hands-ai/openhands:0.9
```

Sollte ollama nicht auf Ihrem lokalen Computer laufen, sondern auf einem Computer im LAN (wie es bei unseren Versuchen der Fall war), dann tragen Sie statt `host.docker.internal` den DNS-Namen des Computers ein, auf dem der Dienst läuft. Vergewissern Sie sich, dass der `openhands-app`-Container Zugriff auf den Port 11434 auf diesem Computer hat und keine Firewall dazwischenfunkt. Sie können dann im Konfigurationsdialog unter *Model* die Zeichenkette `ollama/llama3.1` eingeben (wenn Sie

das LLM llama3.1 von Ihrer lokalen Installation verwenden möchten). Beachten Sie, dass ein Umschalten auf ein Modell eines Cloud-Providers, wie gpt-4o von OpenAI, nur dann gelingt, wenn Sie den Container neu starten und die Variable LLM_BASE_URL nicht definieren.

Bei unseren Versuchen empfanden wir die Fehlermeldungen im Browser-Chat als nicht sehr hilfreich. OpenHands gibt hier wenig technische Fehlermeldungen aus, was oft zu sehr verkürzten, generischen Meldungen führt. Viel hilfreicher war die Log-Ausgabe im Terminal-Fenster, in dem wir den Docker-Container gestartet hatten. Diese Meldungen sind sehr detailliert und geben in der Regel schnell Auskunft über das eigentliche Problem. Hier ein Ausschnitt aus dem Logfile der nachfolgenden Python-Web-Applikation:

```
CodeActAgent LEVEL 0 LOCAL STEP 13 GLOBAL STEP 13

06:37:31 - openhands:INFO: llm.py:486 - Cost: 0.04 USD |
  Accumulated Cost: 0.43 USD
Input tokens: 7757
Output tokens: 53

06:37:31 - ACTION
**CmdRunAction (source=EventSource.AGENT)**
THOUGHT: Let's use `netstat` to find the process using port 5000
  and then kill it.

First, let's find the process ID (PID) using port 5000.
COMMAND:
netstat -tuln | grep :5000
06:37:31 - openhands:INFO: runtime.py:359 - Awaiting session
06:37:31 - openhands:INFO: runtime.py:263 -
-----Container logs:-----
  |INFO:      172.17.0.1:53856 - "GET /alive HTTP/1.1" 200 OK
  |INFO:      172.17.0.1:53856 - "POST /execute_action HTTP/1...
-----
06:37:31 - openhands:INFO: session.py:139 - Server event
06:37:31 - OBSERVATION
**CmdOutputObservation (source=EventSource.AGENT, exit code=1)**
bash: netstat: command not found

[Python Interpreter: /openhands/poetry/openhands-ai-504_aChf-p...
openhands@200f871dd6cf:/workspace $
06:37:31 - openhands:INFO: session.py:139 - Server event
```

Behalten Sie daher immer das Terminal-Fenster mit diesen Log-Meldungen im Blickfeld, wenn Sie mit OpenHands experimentieren.

11.2 OpenHands in der Praxis

Als erste Aufgabe soll unser künstlicher Junior-Software-Entwickler eine Web-Applikation mit einer sehr einfachen Benutzerverwaltung erstellen. Das Backend soll in Python programmiert sein, und die Benutzer sollen in einer SQLite-Datenbank gespeichert werden. Eine Webseite dient zur Verwaltung der Benutzer: Hier können neue Benutzer hinzugefügt und bestehende gelöscht oder verändert werden.

Prompt: Create a JSON REST-API backend using python. Add a route health which responds with HTTP status code 200.

Diese Aufgabe ist für ein mächtiges Modell wie gpt-4o natürlich ein Kinderspiel. Folgende Schritte führt OpenHands aus und kommt dabei nur leicht ins Straucheln:

1. Die Datei `app.py` wird erstellt. Sie enthält den HTTP-Endpunkt für `/health` und setzt den Webserver-Port auf 5000.
2. Mit dem Kommando `python3 /workspace/app.py` wird der Server als Hintergrundprozess gestartet.
3. Der Aufruf `curl http://localhost:5000/health` scheitert mit der Fehlermeldung `Failed to connect to localhost port 5000`.
4. Mit dem Kommando `lsof -i :5000` versucht OpenHands, zu prüfen, ob der Serverprozess auf Port 5000 läuft. Auch dieses Kommando scheitert, da `lsof` nicht in dem Sandbox-Container installiert ist.
5. Da die Schritte 3 und 4 so schnell hintereinander ausgeführt wurden, fand sich noch keine Fehlermeldung für den gescheiterten Serverstart aus Schritt 2 im Logfile. Erst jetzt taucht der Fehler `ModuleNotFoundError: No module named 'flask'` auf.
6. OpenHands behebt das Problem mit der Installation des Python-Moduls.
7. Der neuerliche Aufruf von `curl` liefert das erwartete Ergebnis, und OpenHands gibt die Kontrolle an den Benutzer zurück.

Nach 7 Schritten und 0,05 \$ Kosten ist die erste Aufgabe erfüllt. Ein menschlicher Programmierer hätte den Fehler mit der fehlenden Modul-Installation wohl gar nicht erst gemacht und, wenn doch, das Problem sofort nach dem Server-Start bemerkt.

OpenHands soll die Anwendung jetzt um eine einfache Benutzerverwaltung mit einer Datenbank im SQLite-Format erweitern.

Prompt: Add code for a simple user administration, where users are stored in a sqlite database.

Nach mehreren Fehlern, die OpenHands aber allesamt selbst beheben kann, gibt es zwei weitere Routen zu unserem Backend: Einmal kann die Liste aller Benutzer mit

GET abgerufen werden. Die zweite Route dient zum Erstellen neuer Benutzer mit POST. Wir sind inzwischen bei 28 Schritten und Kosten von 0,29 \$ angekommen.

Als Nächstes möchten wir ein Web-Frontend für die Anwendung haben:

Prompt: Add a web frontend where I can add a user to the database.

Die Eingabemaske auf der sehr schlichten HTML-Seite (siehe Abbildung 11.4) verwendet JavaScript ohne zusätzliche Bibliotheken, was ganz in unserem Sinn ist. Beim Abschicken des Formulars wird die Funktion `fetch` verwendet, um die eingegebenen Daten an das Backend zu schicken. Außerdem erweitert OpenHands das Backend, damit die Webseite unter der Adresse `http://localhost:5000/` ausgeliefert werden kann.

Wir brauchen nun aber auch noch einen Weg, um die Benutzer zu löschen oder zu bearbeiten.

Prompt: Add code for deleting and editing users.

OpenHands kümmert sich hierbei nur um das Backend und fügt die Routen mit den HTTP-Methoden `DELETE` und `PUT` hinzu. Anschließend wird der `DELETE`-Aufruf mit einer nicht existierenden Benutzer-ID getestet, was zu einer Fehlermeldung führt. OpenHands wertet das offenbar als erfolgreichen Test und meldet stolz, dass der Code hinzugefügt wurde und getestet ist. Inzwischen sind 48 Schritte abgearbeitet, und wir haben 0,59 \$ ausgegeben. Der fehlende Frontend-Code wird in nur einem Schritt ergänzt. Da hier mehr Tokens anfallen, kostet dieser Schritt auch etwas mehr.

Prompt: Add frontend code so that I can see a list of all users with edit and delete buttons.

Abschließend wollen wir noch ein paar Beispiel-Benutzer anlegen lassen, damit wir die Applikation ausprobieren können.

Prompt: Insert 5 example users.

OpenHands ruft dazu fünfmal die REST-API mit `curl` und Test-Benutzerdaten auf, wobei jeder Aufruf 0,02 \$ kostet. Gut zu wissen, wenn Sie zum Beispiel 10.000 Test-User hinzufügen wollen.

Jetzt werden Sie mit Recht sagen, dass man für dieses einfache Beispiel auch ChatGPT oder einen anderen Chatbot im Browser verwenden kann. Immerhin bekommt man auch dabei präzise Anweisungen, welche Dateien und Ordner erstellt werden müssen.

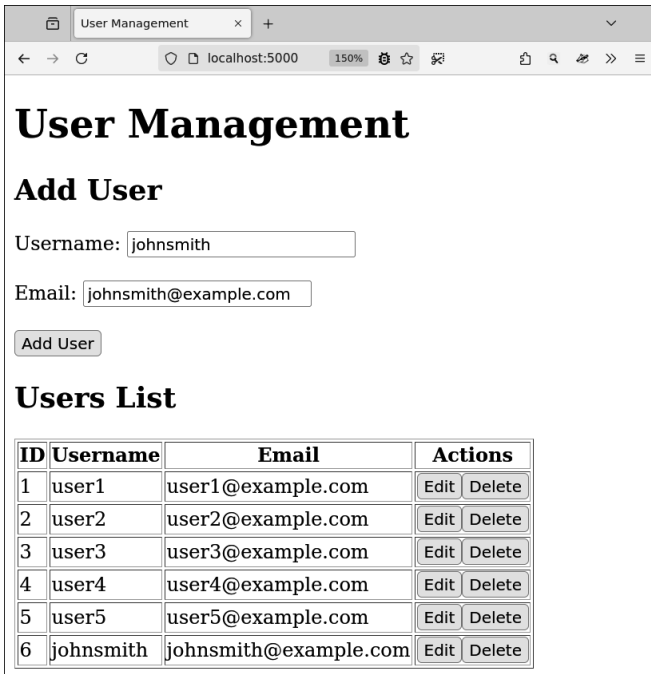


Abbildung 11.4 Die »fertige« Web-Applikation mit einer Liste von Benutzern

Nicht zu viel auf einmal!

Eigentlich hätten wir Ihnen hier gerne ein vollständiges Beispiel mit einem Prompt in dieser Form präsentiert:

Prompt: Create a backend JSON-REST-API in golang with a simple user administration. Users should have Email, Name, Password and get stored in a sqlite database. Add a frontend in vue.js where I can add, edit and delete users.

Die erste Reaktion von OpenHands ist vielversprechend: Es wird ein Plan vorgestellt, in dem zuerst das Backend inklusive Datenbank und anschließend das Frontend erzeugt werden soll. Aber schon die Initialisierung des Go-Moduls schlägt fehl. Bei mehrfachen Versuchen, das Problem zu beheben, wird der Ordner für das Go-Modul verschachtelt unterhalb des letzten Versuchs erstellt. Beim Frontend-Code war das Ergebnis leider auch nicht besser.

OpenHands brach nach 37 Minuten und 60 Schritten ohne irgendein nennenswertes Ergebnis ab. Weder bei dem gewünschten Backend in Go noch bei dem Frontend gab es verwertbaren Code. Nachdem das nicht unser erster Versuch war, der so kläglich scheiterte, konnten wir die Zeit wenigstens für die Mittagspause nutzen.

Fazit

Als wir mit der Arbeit an den Abschnitten zu OpenHands starteten und das technische Konzept dahinter zu verstehen begannen, waren unsere Erwartungen sehr hoch gesteckt. Die Faszination, wie gut ein LLM mit natürlicher Sprache und auch mit Programmiersprachen umgehen kann, trugen seinen Teil dazu bei. Die Ernüchterung folgte aber auf den Fuß, und das Scheitern von OpenHands an banalen Dingen, wie zum Beispiel einem falschen Pfad in der Ordnerstruktur, lassen uns doch mit großen Zweifeln an einer raschen Umsetzung eines *autonomen Software-Ingenieurs* zurück.

Nichtsdestotrotz bietet OpenHands einen sehr spannenden Ausblick, wie ein Teil der Software-Entwicklung in Zukunft funktionieren könnte. Es handelt sich um ein noch sehr junges Projekt, das vor allem von der Weiterentwicklung der Sprachmodelle massiv profitieren wird.

Am meisten enttäuscht hat uns die große Diskrepanz zwischen lokalen Modellen und den Modellen von OpenAI: Während lokale Modelle bei Code-Vervollständigung oder als Chat-Bot, wie wir es in anderen Kapiteln in diesem Buch vorgestellt haben, durchaus brauchbare Ergebnisse erzielen, war das bei OpenHands leider nicht der Fall. Wenn effiziente Software-Entwicklung in Zukunft nur mehr mit einem Zugang zu einem der Tech-Giganten möglich ist (inklusive der unvermeidbaren Datenüberwachung), dann wäre das für uns eine sehr traurige Entwicklung.

11.3 Aider

Aider-Erfinder und -Entwickler Paul Gauthier bezeichnet sein Werk ein wenig lapidar »Aider is AI pair programming in your terminal«. Tatsächlich fehlt Aider eine Web- oder GUI-Oberfläche. Die restliche Beschreibung ist aus unserer Sicht aber ein Understatement. Microsoft bezeichnet ja GitHub Copilot ganz ähnlich als »Your AI pair programmer«.

Funktionell liegen aber Welten zwischen Copilot und Aider! Aider erwartet relativ abstrakte Anweisungen, erzeugt selbstständig die erforderlichen Dateien bzw. führt Änderungen an vorhandenen Projektdateien durch, und fragt schließlich, ob Sie die Änderungen als Git-Commit übernehmen wollen. Wie wir es in der Kapiteleinleitung schon angedeutet haben, agiert Aider viel selbstständiger als GitHub Copilot, Continue oder andere Plug-ins, die in den Editor oder die Entwicklungsumgebung integriert werden.

Installation

Aider ist Open-Source-Software, die auf Python und Git aufbaut – Voraussetzungen also, die auf Ihrem Rechner ziemlich sicher schon erfüllt sind. Um Paketkonflikte mit

Kapitel 12

Retrieval Augmented Generation (RAG) und Text-to-SQL

Dieses Kapitel hebt sich vom Rest des Buchs etwas ab, da wir hier keine Arbeitstechniken und Werkzeuge vorstellen, die Ihnen das Programmieren erleichtern, sondern ein Programm entwickeln, das KI verwendet, um zu einem Ergebnis zu kommen. (Natürlich gibt es dennoch eine Verbindung: Den Code zu den Beispielen in diesem Kapitel haben wir natürlich wieder mit KI-Hilfe erstellt. So enthält auch dieses Kapitel wieder eine Menge Prompts sowie einige Anmerkungen, warum wir den Prompt genau so formuliert haben und warum der eine oder andere vorangegangene Versuch gescheitert ist.)

Die Problemstellung für die folgende KI-Applikation ist, dass Daten aus unterschiedlichen Quellen mithilfe eines großen Sprachmodells (LLM) analysiert werden sollen. Als Basis-Daten werden wir PDF-Dokumente, Webseiten im HTML-Format und eine SQL-Datenbank verwenden. Um diese Anforderungen zu realisieren, verwenden wir zwei verschiedene Techniken: Retrieval Augmented Generation und Text-to-SQL. Die Idee dabei ist, mit diesen Technologien einen schlummernden Datenschatz suchbar zu machen.

In vielen Projekten gibt es das Problem, dass die Stichwortsuche für archivierte PDFs und andere Dokumente nicht ausreichend ist – von den Daten in einer SQL-Datenbank ganz zu schweigen. Fragen in natürlicher Sprache zu stellen und diese mit dem Inhalt dieser Daten beantwortet zu bekommen, kann wirklich ein *Game Changer* in gewissen Bereichen sein.

Retrieval Augmented Generation (RAG) ist eines der Buzz-Wörter im aktuellen KI-Hype. Da es mit vertretbarem Aufwand unmöglich ist, ein fertiges LLM um eigene Informationen zu erweitern, wird einem großen, allgemeinen Sprachmodell ein zusätzliches kleines, selbst erzeugtes Sprachmodell vorgeschaltet, das sogenannte *Embedding*-Modell. Bei einer Abfrage werden die Ergebnisse dieses Modells an das große Sprachmodell weitergeleitet, das in weiterer Folge die Antwort für den Benutzer erzeugt.

Das Embedding-Modell konvertiert unstrukturierte Daten, wie Text oder Bilder, in mehrdimensionale Vektoren. Wie bei einem geografischen Koordinatensystem, das Punkte auf der Erde beschreibt, geben diese Vektoren Auskunft über die Position von Informationen zueinander. Nur sind diese Vektoren nicht wie bei einem geografischen Koordinatensystem auf drei Dimensionen beschränkt, sondern können Hunderte oder Tausende Dimensionen besitzen.

Bei Text-to-SQL erzeugt ein LLM eine SQL-Abfrage aus einer Anfrage in natürlicher Sprache. Damit das funktioniert, werden dem LLM Informationen zur Datenbank-Struktur bei der Anfrage mitübergeben. Sie können sich vorstellen, dass das LLM nur dann eine Chance hat, vernünftige SQL-Abfragen zu formulieren, wenn die Tabellen- und Spaltennamen sinnvolle Bezeichnungen haben.

Für die Entwicklung der KI-Applikation im folgenden Beispiel verwenden wir die Open-Source-Bibliothek *LlamaIndex* (<https://www.llamaindex.ai>). Sie bietet sowohl Anbindung für die Programmiersprache Python als auch für TypeScript. Wir haben uns bei diesem Beispiel für Python entschieden.

12.1 Schnellstart RAG

Als Einstieg in die Thematik wollen wir Ihnen zeigen, wie schnell man zu einer lauffähigen KI-Applikation kommen kann, wenn diese die KI selbst schreibt. Wir möchten auf einer HTML-Seite Fragen zu Dokumenten stellen, die wir in einem Ordner `pdfs` auf der Festplatte ablegen. Es handelt sich um Jahresberichte und weitere Informationen zu einem Forschungsprojekt über Insektenzählungen in Österreich, auf das wir im weiteren Verlauf des Kapitels noch genauer eingehen werden. Unser Prompt an das LLM *Claude-3.5-Sonnet* lautet:

Prompt: Generate a FastAPI backend for a llamaindex Q&A application and a HTML page to input questions. Data for llamaindex is in a folder pdfs. Do not use templating, but serve the HTML file from server root.

Der erzeugte Python-Code ist leider nicht lauffähig, weil sich die *LlamaIndex*-Bibliothek seit dem Ende der Trainingsdaten für Claude etwas geändert hat. Ein Problem, dem wir in diesem Buch schon öfters begegnet sind. Die Änderungen sind aber minimal (der `GPTSimpleVectorIndex` wurde in `VectorStoreIndex` umbenannt), und mit dem Ändern von drei Zeilen Code läuft die Applikation. Wir zeigen Ihnen hier das ganze Backend-Skript, das den Webserver startet, Fragen mithilfe der KI beantwortet und die Antwort in Form von JSON zurückgibt:

```

from fastapi import FastAPI, HTTPException
from fastapi.staticfiles import StaticFiles
from pydantic import BaseModel
from llama_index.core import (
    VectorStoreIndex, SimpleDirectoryReader
)

app = FastAPI()

documents = SimpleDirectoryReader("pdfs").load_data()
index = VectorStoreIndex.from_documents(documents)
query_engine = index.as_query_engine()

class Question(BaseModel):
    text: str

@app.post("/api/ask")
async def ask_question(question: Question):
    try:
        response = query_engine.query(question.text)
        return {"answer": str(response)}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

app.mount("/", StaticFiles(directory="static", html=True),
           name="static")

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)

```

Die meisten der wenigen Codezeilen betreffen den Webserver, der mit den Bibliotheken FastAPI und uvicorn sehr komfortabel einzurichten ist. HTML-Dokumente, die im Ordner static liegen, werden regulär über den Webserver ausgeliefert. Bei einer HTTP-POST-Anfrage, die an die URL /api/ask gesendet wird, wird der Inhalt der Anfrage auf die zuvor definierte Klasse Question abgebildet. Der darin enthaltene text-Eintrag wird dem query_engine-Aufruf übergeben.

Hier geschieht die Magie: Mit nur drei Zeilen Code erstellt die LlamaIndex-Bibliothek die query_engine, die den Inhalt der PDF-Dokumente mit Anfragen in natürlicher Sprache durchsuchen kann. Dazu liest der SimpleDirectoryReader einen Ordner mit Dateien, in unserem Fall PDF-Dokumenten, ein. Anschließend wird ein VectorStoreIndex aus diesen Dokumenten erzeugt, aus dem wiederum die query_engine abgeleitet wird.

Was bei diesen Schritten im Detail passiert, werden wir in den nächsten Abschnitten klären. Zuerst werden wir unsere Applikation ausprobieren. Wie Claude in der kurzen Erklärung erläutert, müssen wir dazu die notwendigen Python-Bibliotheken installieren:

```
pip install fastapi uvicorn llama-index pydantic
```

Das HTML-Frontend kommt mit wenigen Zeilen Javascript-Code und ohne zusätzliche Bibliotheken aus. An der Datei, die wir im Ordner `static` als `index.html` speichern, müssen wir keine Änderungen vornehmen. Der zentrale Teil der HTML-Datei sieht wie folgt aus:

```
<textarea id="question" placeholder="Enter your question here">
</textarea>
<button onclick="askQuestion()">Ask</button>
<div id="answer"></div>

<script>
  async function askQuestion() {
    const question = document.getElementById("question").value;
    const answerDiv = document.getElementById("answer");
    answerDiv.innerHTML = "Loading...";
    try {
      const response = await fetch("/api/ask", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({ text: question }),
      });
      if (!response.ok) {
        throw new Error("Network response was not ok");
      }
      const data = await response.json();
      answerDiv.innerHTML = data.answer;
    } catch (error) {
      answerDiv.innerHTML = "Error: " + error.message;
    }
  }
</script>
```

Die Funktion `askQuestion`, die beim Drücken des `Ask`-Buttons ausgeführt wird, sendet den Inhalt des Textfeldes mit der ID `question` mittels einer HTTP-POST-Anfrage als JSON-String an das Backend. Sofern die Anfrage erfolgreich beantwortet wird, wird der Inhalt der JSON-Struktur `data.answer` in dem entsprechenden HTML-DIV-Bereich eingesetzt.

Was Claude in der Beschreibung nicht erwähnt hat, ist, dass LlamaIndex die Schwerarbeit an OpenAI auslagert. Die Applikation kann also nur funktionieren, wenn Sie einen OpenAI-API-Schlüssel haben und noch Kredit auf Ihrem Konto vorhanden ist. Damit die Applikation Zugang zu Ihrem Konto bekommt, müssen Sie den API-Schlüssel in der Umgebungsvariable `OPENAI_API_KEY` setzen. In gängigen Linux-Shells rufen Sie dazu folgendes Kommando auf:

```
export OPENAI_API_KEY="sk-proj-xxxxxxx"
```

Jetzt können wir unsere Applikation mit folgendem Kommando starten:

```
python main.py
```

Das Ergebnis ist durchaus überzeugend (siehe Abbildung 12.1).

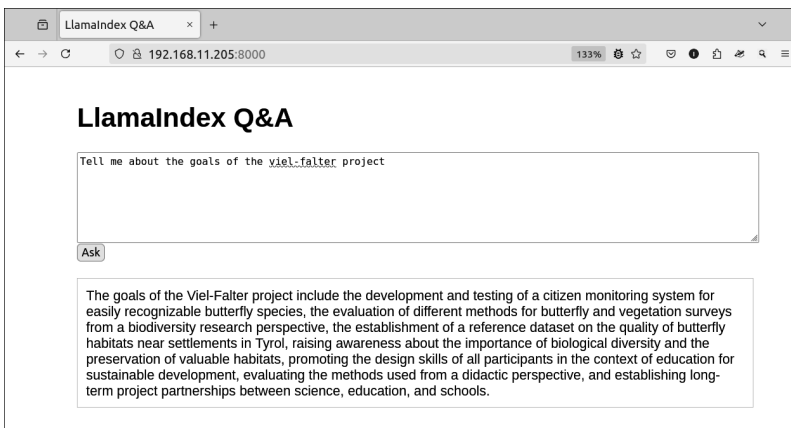


Abbildung 12.1 Unser Schnelleinstieg in RAG: Nach 10 Minuten ist die erste Web-Applikation fertig, die Fragen in natürlicher Sprache zu PDF-Dokumenten auf der SSD beantwortet.

Bevor Sie jetzt das Buch zuklappen (oder den E-Reader ausschalten) und an die Arbeit gehen, um dieses Beispiel umzusetzen, möchten wir Sie noch auf einige Defizite dieser Applikation hinweisen:

- ▶ Jedes Mal, wenn Sie den Webserver starten, wird der Index neu erzeugt. Das ist nicht notwendig, denn Sie können einen VectorStoreIndex auch lokal speichern.
- ▶ Mit den Standardeinstellungen müssen Sie einen Zugang bei OpenAI haben und bezahlen. LlamaIndex funktioniert aber mit anderen Cloud-Anbietern und auch mit lokalen LLMs, und das gar nicht schlecht.
- ▶ Welche Typen von Dokumenten Sie indizieren wollen und wie die Indizierung im Detail funktioniert, können Sie sehr genau einstellen. Hier gibt es großes Optimierungspotenzial.

Lesen Sie die nächsten Abschnitte, um einen etwas tieferen Einblick in die Funktionsweise von RAG und in die LlamaIndex-Bibliothek zu bekommen.

12.2 Das viel-falter-Projekt

Anstatt eines Hello-World-Beispiels wollen wir für dieses Beispiel Daten aus einem echten Projekt verwenden. Wir werden sehen, ob wir mit unserer KI-Applikation einen Mehrwert erzielen können.

Das Projekt *viel-falter* ist eine österreichische Initiative zur Beobachtung von Tagfaltern als Indikatoren für Veränderungen in der Natur- und Kulturlandschaft. Das Monitoring wird von der Universität Innsbruck geleitet. Einer der Autoren betreut seit über 10 Jahren die Datenbank, die Eingabemöglichkeiten und die internen Verwaltungstools für das Projekt.

Wir haben das Projekt für unser Beispiel gewählt, weil wir hier einerseits die Möglichkeit haben, Dokumente suchbar zu machen, die nicht in einem der großen LLMs zu finden sind (Stichwort RAG), und andererseits auch den Zugriff über natürliche Sprache auf eine SQL-Datenbank zeigen können (siehe Abschnitt 12.7, »Text-to-SQL«).

The screenshot shows the website 'viel-falter.at' with the following content:

- Navigation: en | Monitoring | Ergebnisse | News | Falter | Karte | Kontakt | Mach mit!
- Statistics:
 - 4819 Erhebungen
 - 52387 Gezählte Falter
 - 307 Beteiligte Freiwillige
- Section: **Die häufigsten Tagfalter**
- Text: Das große Ochsenauge (3488 Individuen), der Hauhechel-Bläuling (1189 Individuen), das Kleine Wiesenvögelchen (1270 Individuen), der Kleine Fuchs (969 Individuen) und der Kleine Kohlweißling (1215 Individuen) wurden bei den Erhebungen unserer Expert*innen in Österreich am häufigsten gesichtet.
- Five circular images of butterflies with their names and photos:
 - Großes Ochsenauge (*Maniola jurtina*, Foto: Frederick Manick)
 - Hauhechel-Bläuling (*Polymmatinus icarus*)
 - Kleines Wiesenvögelchen (*Coenonympha pamphilus*)
 - Kleiner Fuchs (*Aglais urticae*, Foto: Petra Schattaneck)
 - Kleiner Kohlweißling (*Pieris rapae*, Foto: Petra Schattaneck)

Abbildung 12.2 Ein Teil der Webseite des viel-falter.at Projekts

Da die Reichweite des Projekts aufgrund seiner Regionalität überschaubar ist, waren wir wenig überrascht, dass folgender Prompt bei dem eigentlich sehr guten LLM *Claude 3.5 Sonnet* zu keinem brauchbaren Ergebnis führte:

Prompt: Tell me about the goals and the partners of the viel-falter project.

I apologize, but I don't have specific information about a project called "viel-falter" in my knowledge base

Aktuelle Informationen findet man zwar im Internet unter <https://viel-falter.at>, aber wir haben nicht erwartet, dass diese Webseite im Index von Claude enthalten ist. Damit unser KI-Assistent kompetente Antworten geben kann, werden wir ein Embedding-Modell verwenden, um Inhalte der Webseite und einige PDF-Dokumenten damit in Vektoren zu konvertieren. Die Ergebnisse dieses Vorgangs speichern wir in einer Vektor-Datenbank, die eine spätere Suche ermöglicht.

12.3 Dokumente laden

Die Jahresberichte im PDF-Format beinhalten wertvolle Informationen zum Projekt und finden sich auf der WordPress-Website unter <https://viel-falter.at/ergebnisse/veroeffentlichungen>. Um nicht jedes der 18 PDF-Dokumente einzeln herunterladen zu müssen, haben wir uns wieder einmal der KI bedient:

Prompt: How do I save linked PDF documents from the website <https://viel-falter.at/ergebnisse/veroeffentlichungen> to a local folder pdfs using Linux shell commands? Do not use temporary files but Unix pipes and remove duplicates. This site uses only absolute URLs.

Wir haben an dem Prompt etwas gefeilt, weil *Claude-3.5-sonnet* anfangs temporäre Dateien erzeugte, was wir für nicht notwendig halten. Bis auf das überflüssige `-p` beim Erstellen des Verzeichnisses in der ersten Zeile (der Schalter wäre nur notwendig, wenn man eine verschachtelte Verzeichnisstruktur erstellt) gibt es nicht viel auszusetzen:

```
mkdir -p pdfs && \
wget -q0- https://viel-falter.at/ergebnisse/veroeffentlichungen/\
  | grep -oP 'href="[^\"]*\'.pdf"' | \
  sed 's/href="//;s/"$//' | \
  sort -u | \
  xargs -I {} wget -qnc -P pdfs {}
```

Das `wget`-Kommando in Zeile 2 lädt den HTML-Code der Webseite, der durch das Pipe-Zeichen an das `grep`-Kommando weitergeleitet wird. Hier wird der Schalter `-o` verwendet, um nur den Teil der Zeile auszugeben, in dem der reguläre Ausdruck gefunden wurde. Die Einstellung für Perl-kompatible reguläre Ausdrücke (`-P`) wäre

in diesem Fall nicht notwendig, ist aber auch kein Fehler. Um die vorangestellte Zeichenkette `href=` und die Anführungszeichen der Links zu entfernen, kommt `sed` zum Einsatz. `sort -u` entfernt die Duplikate, und schließlich führt `xargs` ein `wget`-Kommando für jeden gefundenen Link aus, wobei mit `-P` der Pfad zum Speichern der Dokumente festgelegt wird.

Ohne den Hinweis, dass alle URLs absolute Angaben sind, erzeugt Claude im letzten Schritt eine Adresse mit vorangestelltem Protokoll und Hostnamen (`https://vielfalter.at/`), weist aber im Erklärungstext darauf hin, dass dieses Vorgehen nur bei relativen Links funktionieren wird.

Das hier verwendete Kommando muss bei einer anderen Webseite nicht zwangsläufig genauso gut funktionieren. Der Suchbegriff bei dem `grep`-Kommando verwendet doppelte Anführungszeichen, was in HTML eigentlich nicht vorgeschrieben ist; das Contentmanagement-System WordPress, das hier verwendet wird, kommt damit aber zurecht.

Abgesehen von den PDF-Dokumenten werden wir noch eine Liste von Webseiten in unser Embedding-Modell aufnehmen. Diese werden beim Start der Applikation aus dem Internet geladen und in Text konvertiert.

SimpleDirectoryReader und SimpleWebPageReader

Die LlamaIndex-Bibliothek verwendet *Dokumente* und *Nodes* als wesentliche Bausteine. Dokumente sind Container für Daten, wie zum Beispiel ein PDF-Dokument oder ein Markdown-Dokument. Aber auch die Ausgabe einer Datenbank-Abfrage oder die Antwort von einem Webserver kann ein Dokument sein. Zu einem Dokument werden Metadaten abgelegt (zum Beispiel der Dateiname bei einer Datei) und Relationen zu anderen Dokumenten (zum Beispiel der Ordner auf der Festplatte). Nodes sind Teile eines Dokuments, die ebenfalls Metadaten und Relationen enthalten.

LlamaIndex macht es uns sehr einfach, die heruntergeladenen PDF-Dokumente in das interne Dokument-Format zu verwandeln:

```
from llama_index.core import SimpleDirectoryReader

pdf_documents = SimpleDirectoryReader("pdfs").load_data(
    show_progress=True
)
```

Der `SimpleDirectoryReader` aus dem `llama_index.core`-Paket erzeugt eine Liste aller Dateien im `pdfs`-Ordner im LlamaIndex-Dokumentformat. In unserem Fall sind das 18 PDF-Dokumente. Der Schalter `show_progress=True` erzeugt Fortschrittsbalken in der Kommandozeile. Als Erweiterung zu dem Beispiel in Abschnitt 12.1, »Schnellstart

RAG«, werden wir noch einige wichtige Webseiten ebenfalls als Dokumente verwenden. Dazu benötigen wir die Bibliothek `SimpleWebPageReader`:

```
from llama_index.readers.web import SimpleWebPageReader

webpages = [
    "https://viel-falter.at/",
    "https://viel-falter.at/monitoring/ziele-vision/",
    "https://viel-falter.at/projektpartner/",
    "https://viel-falter.at/news/",
    "https://viel-falter.at/tagfalter-exkursion/",
];

web_documents = SimpleWebPageReader(
    html_to_text = True).load_data(webpages);
```

Die `html_to_text`-Option legt fest, dass wir nur an dem textlichen Inhalt der Webseiten interessiert sind. Sowohl `pdf_documents` als auch `web_documents` enthält nun eine Liste von `LlamaIndex`-Dokumenten, die wir in einem weiteren Schritt in einen suchbaren *Index* verwandeln werden. Um die beiden Listen zu vereinen, verwenden wir in Python den `+` Operator:

```
documents = pdf_documents + web_documents
```

Der `SimpleDirectoryReader` und der `SimpleWebPageReader` sind aber nur zwei Beispiele für Möglichkeiten, wie Sie Daten in `LlamaIndex` laden können. Auf der Webseite <https://llamahub.ai/> finden Sie Python-Module für unzählige weitere Formate.

12.4 Index erzeugen

Wie wir in Abschnitt 12.1, »Schnellstart RAG«, gesehen haben, bietet `LlamaIndex` eine sehr komfortable Funktion, um Dokumente in einen suchbaren Index zu konvertieren: `VectorStoreIndex.from_documents`. Als Parameter benötigt die Funktion nur die Liste der Dokumente. `LlamaIndex` führt dabei im Hintergrund einige interessante Schritte durch, die wir uns im Folgenden etwas genauer ansehen werden.

Zuerst werden aus den `LlamaIndex`-Dokumenten `LlamaIndex`-Nodes gemacht. Dabei kommt ein Text-Splitter zum Einsatz, zum Beispiel der `SentenceSplitter`. Im Unterschied zu einem »normalen« Text-Splitter legt dieser Wert darauf, einen Satz nicht auf mehrere Nodes aufzuteilen.

Damit Sie ein Gefühl dafür bekommen, wie die Aufteilung von Dokumenten in Nodes mit dem `SentenceSplitter` in Zahlen aussehen kann, haben wir für Sie die PDF-Seiten aus den 18 Dokumenten in unserem Beispiel gezählt (natürlich mit der Hilfe eines Linux-Kommandos, das wir von der Claude-KI erstellen ließen) und mit den erzeug-

ten Nodes verglichen. Aus den 378 PDF-Seiten werden 466 LlamaIndex-Nodes. Wir haben hier Teile des Inhalts eines solchen Nodes ohne das Embedding in Form einer JSON-Struktur für Sie abgedruckt:

```
{
  "page_label": "5",
  "file_name": "Vielfalter-Falter_Tirol_2018_bis_2022.pdf",
  "file_path": "/data/pdfs/Vielfalter-Falter_Tirol_2018_bis_...",
  "file_type": "application/pdf",
  "file_size": 3190932,
  "creation_date": "2024-07-23",
  "last_modified_date": "2024-07-23",
  "_node_content": "{ \"id\": \"2d677c1e-da77-459d-9282...\",
    \"text\": \"Im Expert*innen-Monitoring werden inner ...\",
    \"class_name\": \"TextNode\" }",
  "_node_type": "TextNode",
  "document_id": "94bdaa9f-c86c-4cf2-b38b-6d03805d265c",
  "doc_id": "94bdaa9f-c86c-4cf2-b38b-6d03805d265c",
  "ref_doc_id": "94bdaa9f-c86c-4cf2-b38b-6d03805d265c"
}
```

Sie erkennen einige Metadaten zu dem Dokument. Der Inhalt (Text) ist dem Schlüssel `_node_content` zugeordnet.

Diese Umwandlung von Dokumenten in Nodes können Sie über den zusätzlichen Parameter `transformations` beim Aufruf von `VectorStoreIndex.from_documents` steuern. Dabei wird eine Liste von Funktionen übergeben, die abgearbeitet werden:

```
from llama_index.core import VectorStoreIndex
from llama_index.core.extractors import TitleExtractor
from llama_index.core.node_parser import SentenceSplitter
from llama_index.embeddings.openai import OpenAIEmbedding

index = VectorStoreIndex.from_documents(
    documents=documents,
    transformations=[
        SentenceSplitter(),
        TitleExtractor(),
        OpenAIEmbedding(),
    ],
    show_progress=True,
)
```

Nach dem `SentenceSplitter` verwenden wir den `TitleExtractor`, ein weiteres Modul aus der LlamaIndex-Bibliothek. Dieser versucht mithilfe eines LLMs eine kurze Zusammenfassung des aufgeteilten Textes zu erzeugen. (Wie Sie das LLM einstellen,

zeigen wir Ihnen etwas später.) Der Vorgang verlangsamt die Indizierung deutlich, weil für jeden der 466 Nodes das LLM befragt wird, führt aber auch zu besseren Ergebnissen in der späteren Suche.

Beachten Sie, dass LLMs in der Cloud in der Regel Limits haben, wie viele API-Aufrufe sie in einem gewissen Zeitraum abrufen können. Bei Anthropic lagen die API-Limits für das von uns verwendete Modell *Claude 3.5 Sonnet* bei 50 Requests und 40,000 Tokens pro Minute. Alleine das Limit der Requests wird mit den 466 Zugriffen, die innerhalb weniger Minuten ausgeführt werden, schnell überschritten. Wir verwendeten den TitleExtractor deswegen zusammen mit einer lokalen Ollama-Instanz, wo die Limits nur von ihrer Hardware vorgegeben werden.

Der dritte Eintrag in der Liste der `transformations` verweist auf das Embedding-Modell. Hier wird es richtig spannend, weil das Modell großen Einfluss auf den erzeugten Index hat und die Auswahl der Modelle schier endlos ist.

Das Embedding-Modell

Wir verwenden Embedding-Modelle, um Textpassagen aus den PDF-Dokumenten und den geladenen Webseiten in Vektoren zu verwandeln. Ein Vektor repräsentiert dann aber nicht ein Wort oder einen Token, sondern einen ganzen Absatz oder eine ganze Seite. Wir zeigen gleich noch ein Beispiel, wie so ein Vektor aussehen kann. Je nachdem, wie groß die Trainingsdaten waren und wie viele Dimensionen die Vektoren haben, können Embedding-Modelle wenige Hundert MByte bis zu mehrere GByte in der Größe erreichen. Die abgeleiteten Vektoren, die wir später speichern werden, hingegen belegen nur wenige MByte für unseren Datensatz.

Der oben gezeigte Aufruf `OpenAIEmbedding()` lädt das proprietäre Modell `text-embedding-ada-002-v2` von OpenAI. Zur Verwendung benötigen Sie einen API-Schlüssel und Guthaben auf Ihrem OpenAI-Konto. Sie können aber auch ein Open-Source-Modell von der Plattform *Hugging Face* verwenden:

```
from llama_index.embeddings.huggingface import (
    HuggingFaceEmbedding
)

embed_model_hf = HuggingFaceEmbedding(
    cache_folder="/home/user/huggingface_cache",
    model_name="BAAI/bge-m3",
)
```

Die Angabe des `cache_folder` bei Hugging Face ist sinnvoll, da das Modell sonst immer wieder neu aus dem Internet geladen werden muss. Auch wenn Embedding-Modelle nicht so groß sind wie lokale LLMs, kommen auch dabei rasch ein paar GByte zusammen.

Das hier verwendete BAAI/bge-m3 wurde mit über 100 Sprachen trainiert und steht unter der freien MIT-Lizenz zur Verfügung. Die Vektoren haben 1024 Dimensionen, und das Modell kann Eingaben mit bis zu 8192 Tokens verarbeiten. Der Speicherbedarf ist mit über 8 GByte aber nicht zu unterschätzen. Wenn Sie nur mit englischen Texten arbeiten, ist das deutlich kleinere Modell BAAI/bge-base-en-v1.5 eine gute Alternative.

Hugging Face, die Plattform für KI-Modelle

Auf <https://huggingface.co> finden Sie neben einer großen Anzahl an Modellen auch das *Massive Text Embedding Benchmark*-Leaderboard <https://huggingface.co/spaces/mteb/leaderboard>, ein Ranking aktueller Modelle mit Ihren Parametern.

Auf der Suche nach dem richtigen Modell für ihren Zweck können Sie hier nach Themen oder Schlagworten filtern. Außerdem finden Sie Trainingsdaten für viele der freien Modelle.

Wenn Sie einen lokalen Ollama-Server betreiben, können Sie Embedding-Modelle auch dort installieren und bei der Indexerzeugung verwenden:

```
from llama_index.embeddings.ollama import OllamaEmbedding

embed_model_ol = OllamaEmbedding(model_name="mxbai-embed-large")
```

Das mxbai-embed-large verwendet 1024 Dimensionen und hat einen Speicherbedarf von 670 MByte. Bei unseren Versuchen mit den PDF-Dokumenten und Webseiten lieferte das Modell brauchbare Ergebnisse, wenn auch nicht so gute Treffer wie das deutlich größere BAAI/bge-m3 von Hugging Face.

Doch wozu genau wird das Embedding-Modell verwendet? Die Eingabedaten werden mit den trainierten Modelldaten verglichen und die Vektoren für die Eingabe-Daten daraus abgeleitet. Das Embedding, also der Vektor, für den weiter oben gezeigten LlamaIndex-Node sieht zum Beispiel so aus:

```
[ 0.010941985063254 , -0.007302129175513 , 0.017867475748062 , ... ]
```

Wir haben darauf verzichtet, die restlichen 765 Fließkommazahlen abzudrucken. Das Open-Source-Modell BAAI/bge-base-en-v1.5, das wir bei diesem Versuch verwendet haben, arbeitet mit 768 Dimensionen.

Das Erzeugen des Index ist eine rechenintensive Operation. Vor allem wenn Sie lokale Modelle verwenden (in Ollama oder von Hugging Face), macht sich eine leistungsstarke GPU bemerkbar. Moderne Grafikkarten von Nvidia oder die Prozessoren in modernen Apple-Computern liefern aktuell die beste Unterstützung. Außerdem hängt die Zeit der Berechnung noch von dem verwendeten Modell ab. Mehrere

Dimensionen und größere Modelle benötigen mehr Rechenzeit, liefern aber auch bessere Ergebnisse.

Bei Embedding-Modellen aus der Cloud brauchen Sie selbst keine starke Hardware. Die Arbeit passiert in den Rechenzentren des gewählten Anbieters. Um die Vektoren zu berechnen, müssen aber alle Daten, die Sie indizieren, in die Cloud geladen werden. Was der Anbieter nach der Berechnung mit Ihren Daten macht, wissen wir natürlich nicht, wir finden es aber höchst unwahrscheinlich, dass sie einfach gelöscht werden.

Es gibt freilich noch andere Anbieter außer OpenAI, bei denen Sie Embedding-Modelle mieten können. Google, MistralAI, CohereAI, IBM, Oracle, Nvidia ... die Liste liest sich wie das Who-is-who der IT-Branche. Beispiele für die Verwendung mit der LlamaIndex-Bibliothek finden Sie auf der Webseite <https://docs.llamaindex.ai/en/stable/examples> unter *Embeddings*.

Bedenken Sie auch, dass das Embedding-Modell nicht nur zum Erzeugen des Index verwendet wird, sondern auch bei jeder Frage, die Sie stellen. Wie wir in Abschnitt 12.6, »RAG-Abfragen«, gleich noch sehen werden, muss das gleiche Embedding-Modell, mit dem der Index erzeugt wurde, auch dazu verwendet werden, Ihre Fragen in Vektoren umzuwandeln.

12.5 Vector-Store-Datenbanken

Um die Vektor-Embeddings nicht bei jedem Start der Applikation erneut zu berechnen, werden wir den Index in einer Datenbank speichern.

Die Aufgabe der Vektor-Datenbank ist es, die numerische Repräsentation von unstrukturierten Daten (die Vektoren mit ihren Dimensionen) zu speichern und diese mit verschiedenen Algorithmen zu suchen. Dabei unterscheiden sich die Implementierungen in solche, bei denen externe Bibliotheken diese Suche in einer bestehenden (relationalen oder NoSQL-)Datenbank ausführen, solche, bei denen dafür Plug-ins in der Datenbank verwendet werden, und *echte* Vektor-Datenbanken.

Echte Vektor-Datenbanken sind auf eine Stufe mit SQL- und NoSQL-Datenbanken zu stellen. Während SQL-Datenbanken die Ausgangsdaten in eine starke Struktur zwingen (durch starre Tabellen mit vielen Datentypen), strukturiert NoSQL die Daten nur grob durch Schlüssel-Wert-Paare mit Datentypen wie Zahlen, Zeichenketten oder Datum. Bei Vektor-Datenbanken werden die Ausgangsdaten unstrukturiert abgelegt, und nur die errechneten Vektoren kommen bei der Suche zum Einsatz.

Ein prominenter Vertreter echter Vektor-Datenbanken ist Milvus. Die Datenbank steht unter der freien Apache-2-Lizenz und kann sehr einfach in unterschiedlichen Szenarien eingesetzt werden: Die Lite-Version eignet sich hervorragend zum *Prototyping* oder auf Geräten mit sehr schwacher Hardware. Für produktive Deployments

Auf einen Blick

TEIL I Coding mit KI-Unterstützung

1	Einführung	17
2	Pair Programming	63
3	Debugging	107
4	Refactoring	127
5	Software testen	145
6	Software dokumentieren	181
7	Datenbanken	199
8	Scripting und Systemadministration	227

TEIL II Sprachmodelle lokal ausführen, fortgeschrittene KI-Tools

9	Sprachmodelle lokal ausführen	261
10	Code automatisiert verarbeiten	295
11	Level-3-Tools: OpenHands und Aider	325
12	Retrieval Augmented Generation (RAG) und Text-to-SQL	351
13	Risiken und Ausblick	385

Inhalt

Vorwort	9
---------------	---

TEIL I Coding mit KI-Unterstützung

1 Einführung 17

1.1 Coding per Chat	17
1.2 Die Kunst des Prompting	27
1.3 Code-Assistenten	35
1.4 Chat oder Assistent?	47
1.5 Grundlagen von Large Language Models (LLMs)	48

2 Pair Programming 63

2.1 Code in Funktionen strukturieren	64
2.2 Beispiel: IBAN-Validierung	74
2.3 Objektorientiert programmieren	81
2.4 Beispiel: Quiz	90
2.5 Beispiel: Sudoku lösen	96

3 Debugging 107

3.1 Web-Applikationen	108
3.2 App-Entwicklung	117
3.3 Entwicklung am Raspberry Pi	120
3.4 Visual Studio und VS Code	123
3.5 Fazit	125

4 Refactoring 127

4.1 Einführung ins Refactoring	128
4.2 Refactoring mit KI-Werkzeugen	128

4.3	Best Practices	141
4.4	Fazit	143

5 Software testen 145

5.1	Erzeugen von Testdaten	146
5.2	KI-gestützte Testautomatisierung	150
5.3	Testgetriebene Entwicklung mit KI	155
5.4	Arbeiten mit Abhängigkeiten	162
5.5	Optimieren von Tests	168
5.6	E2E-Tests	175
5.7	Fazit	179

6 Software dokumentieren 181

6.1	Inline-Dokumentation	183
6.2	Funktions- und Klassendokumentation	186
6.3	API-Dokumentation	190
6.4	Veraltete Dokumentation	193
6.5	Fazit	196

7 Datenbanken 199

7.1	Datenbankdesign	199
7.2	SQL-Kommandos	210
7.3	Administration	218
7.4	Client-Programmierung	225

8 Scripting und Systemadministration 227

8.1	Scripting	228
8.2	Beispiel: Python-Script in PHP-Code umwandeln	235
8.3	Reguläre Muster	237
8.4	Systemadministration	241
8.5	Beispiel: wget-Script plus tmpfs-Konfiguration	249
8.6	GitHub Copilot und ChatGPT im Terminal aufrufen	254

TEIL II Sprachmodelle lokal ausführen, fortgeschrittene KI-Tools

9	Sprachmodelle lokal ausführen	261
<hr/>		
9.1	Die Qual der LLM-Wahl	262
9.2	GPT4All	267
9.3	Ollama	271
9.4	Open WebUI für Ollama	276
9.5	Continue	284
9.6	Ollama-API	287
9.7	Tabby	290
9.8	Fazit	293
10	Code automatisiert verarbeiten	295
<hr/>		
10.1	OpenAI-API	296
10.2	Ollama-API	308
10.3	Groq-API	311
10.4	Beispiel: Code automatisiert kommentieren	312
10.5	Beispiel: von Python 2 zu Python 3	321
11	Level-3-Tools: OpenHands und Aider	325
<hr/>		
11.1	OpenHands	327
11.2	OpenHands in der Praxis	333
11.3	Aider	336
11.4	Aider in der Praxis	341
12	Retrieval Augmented Generation (RAG) und Text-to-SQL	351
<hr/>		
12.1	Schnellstart RAG	352
12.2	Das viel-falter-Projekt	356
12.3	Dokumente laden	357
12.4	Index erzeugen	359
12.5	Vector-Store-Datenbanken	363
12.6	RAG-Abfragen	366
12.7	Text-to-SQL	371

13	Risiken und Ausblick	385
13.1	Probleme und Einschränkungen beim Einsatz von KI-Tools	385
13.2	Beispielhaftes KI-Versagen	389
13.3	Ethische Fragen	400
13.4	Schlussfolgerungen und Ausblick	404
Index		409

Coding mit KI

Das Praxisbuch für die Softwareentwicklung



Künstliche Intelligenz verändert, wie wir Software entwickeln. In diesem Buch erfahren Sie, was Tools wie ChatGPT oder GitHub Copilot bereits leisten, wo ihre Grenzen liegen und wie Sie sie verantwortungsvoll einsetzen.

Praxisnahe Beispiele zeigen Ihnen, wie Sie schon heute mit KI-Unterstützung schneller, effizienter und fehlerfreier programmieren. Ein pragmatischer Leit-faden für alle, die wissen wollen, wie künstliche Intelligenz die Softwareentwicklung umkrempelt.

- + Wie funktioniert künstliche Intelligenz?
- + Pair Programming mit KI
- + Debugging, Refactoring und Unit-Tests
- + Dokumentation automatisiert erstellen
- + Datenbankentwicklung und -design
- + Scripting und Administration
- + Lokale LLMs ausführen
- + Level-3-Tools: OpenHands und Aider
- + Automation über APIs
- + RAG und Text-to-SQL
- + Risiken, Grenzen und Ausblick

**AI won't replace programmers –
but programmers with AI are
going to replace programmers
that don't use AI.**

Michael Kofler, Bernd Öggl und **Sebastian Springer** blicken auf jahrzehntelange Erfahrung in der Softwareentwicklung zurück. Sie haben ausgelotet, an welchen Aufgaben KI-Helfer heute noch scheitern, was sie bereits können und wie sie den Arbeitsalltag einfacher machen.

