O'REILLY®

Deutsche Ausgabe

Prompt Engineering für Large Language Models

LLM-basierte Anwendungen entwickeln, steuern und optimieren

John Berryman &
Albert Ziegler

Übersetzung von Jens Olaf Koch

KAPITEL 8

Dialogbasierte Agenten

In Kapitel 3 ging es um den Übergang von Completion- zu Chatmodellen. Ein reines Chatmodell kennt nur die Informationen, die es während des Trainings gelernt hat, sowie das, was der Nutzer ihm im aktuellen Gespräch mitteilt. Es hat jedoch keine Möglichkeit, neue Informationen aus der Außenwelt zu beziehen oder eigenständig mit externen Systemen zu interagieren und im Namen des Nutzers Aktionen auszuführen.

Die LLM-Community macht große Fortschritte dabei, diese Beschränkungen durch dialogbasierte Handlungsfähigkeit (*Conversational Agency*) zu überwinden. *Handlungsfähigkeit* bezeichnet die Eigenschaft einer Entität, selbstständig Aufgaben ausführen und Ziele auf autonome Weise erreichen zu können. Die dialogbasierten Agenten, die wir in diesem Kapitel betrachten, bieten eine Erfahrung, die einem Chat ähnelt – einem wechselseitigen Dialog zwischen Nutzer und Assistent. Allerdings verfügen sie über zusätzliche Fähigkeiten: Sie können neue Informationen beschaffen und mit der realen Welt interagieren.

In diesem Kapitel stellen wir Ihnen mehrere hochmoderne Ansätze zur Entwicklung eines dialogbasierten Agenten vor. Wir zeigen, wie Modelle Werkzeuge nutzen können, um mit der Außenwelt zu interagieren, wie sie darauf konditioniert werden können, komplexe Probleme besser zu durchdenken, und wie sich der beste Kontext zusammenstellen lässt, um lange oder komplexe Interaktionen zu ermöglichen. Am Ende dieses Kapitels werden Sie in der Lage sein, Ihren eigenen KI-Agenten zu bauen – ein System, das nicht nur Gespräche führt, sondern auch selbstständig Aufgaben in der realen Welt ausführt.

Werkzeuge einsetzen

Ein Sprachmodell, das isoliert arbeitet, ist in seinen Möglichkeiten eingeschränkt. Ein Chatassistent ist faszinierend, da er in gewisser Weise den digita-

len Zeitgeist der Welt verkörpert. Er kann Informationen zu nahezu jedem Thema liefern, auf unterschiedliche Denkschulen zurückgreifen und beim Brainstorming helfen. Als Tutor ist das Modell großartig – sofern man gelegentliche Halluzinationen in Kauf nimmt. Doch eines kann es nicht: auf »unsichtbares« Wissen zugreifen – also auf Informationen, die ihm während des Trainings nicht bereitgestellt wurden.

Im beruflichen Alltag nutzen Sie regelmäßig vertrauliche Informationen – etwa Unternehmensdokumentationen, interne Memos, Chatnachrichten oder Code –, auf die ein Sprachmodell keinen Zugriff hat. Außerdem arbeiten Sie in der Gegenwart, nicht in der Vergangenheit. Daher können ältere Informationen weniger relevant oder sogar falsch sein. Wenn das Modell die neuesten API-Änderungen einer von Ihnen verwendeten Bibliothek oder aktuelle Nachrichtenereignisse nicht kennt, sind die Antworten möglicherweise irreführend oder fehlerhaft. Im Extremfall benötigen Sie sogar Informationen in Echtzeit. Wenn Sie beispielsweise eine Reise planen, müssen Sie wissen, welche Flüge *aktuell* verfügbar sind. Ein reines Chatmodell hat auf all das keinen Zugriff.

Abgesehen davon, dass ihnen wichtige Informationen fehlen, sind Sprachmodelle bei bestimmten Aufgaben schlichtweg nicht besonders gut – am deutlichsten zeigt sich das bei Mathematik. Wenn Sie ChatGPT mit einer einfachen Rechenaufgabe konfrontieren, liefert es häufig die richtige Antwort – weil es sich solche Aufgaben gewissermaßen »gemerkt« hat. Doch je größer die Zahlen werden oder je komplexer die Berechnung ist, desto ungenauer werden die »Schätzungen« des Modells. Noch problematischer ist, dass diese Fehler oft mit großer Überzeugung als wahr präsentiert werden.

Darüber hinaus führen Chatmodelle auch keine Aktionen aus – sie reden lediglich! Die einzige Möglichkeit, wie sie eine Veränderung in der realen Welt bewirken können, besteht darin, den Nutzer zu einer Handlung aufzufordern. Sprachmodelle können keine Flugtickets buchen, keine E-Mails versenden und keine Raumtemperatur ändern.

Um all diese Probleme zu lösen, setzt die LLM-Community zunehmend auf den Einsatz von Werkzeugen. Durch solche *Tools* erhalten Sprachmodelle Zugriff auf aktuelle Informationen, können nicht sprachliche Aufgaben ausführen und mit ihrer Umgebung interagieren. Die Grundidee ist einfach: Man informiert das Modell darüber, welche Tools ihm zur Verfügung stehen, wann es sie verwenden darf und wie es das tun soll – und das Modell nutzt diese Tools dann, um externe APIs anzusprechen. Es ist Aufgabe der Anwendung, Toolaufrufe in der Completion zu erkennen, Anfragen an reale APIs weiterzuleiten und die erhaltenen Informationen in künftige Prompts für das Modell einzubinden.

Für den Einsatz von Tools feingetunte LLMs

Im Juni 2023 stellte OpenAI ein neues Modell vor, das speziell für Toolaufrufe feingetunt wurde. Inzwischen sind mehrere konkurrierende Anbieter diesem Beispiel gefolgt. Werfen wir einen Blick darauf, wie OpenAI das Thema Tools angeht.

Tools definieren und nutzen

Zunächst definieren wir die tatsächlichen Funktionen, die mit der realen Welt interagieren, Informationen sammeln und Änderungen an der Umgebung vornehmen. Die Implementierung ist hier nur angedeutet, aber wenn Sie möchten, finden Sie ohne Weiteres eine Python-Bibliothek, mit der sich ein echter Thermostat ansteuern lässt:

```
import random

def get_room_temp():
    return str(random.randint(60, 80))

def set_room_temp(temp):
    return "DONE"
```

Als Nächstes stellen wir beide Funktionen als JSON-Schema (*https://oreil.ly/rZsdN*) dar, damit OpenAI sie im Prompt verwenden kann.

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "get_room_temp",
            "description": "Get the ambient room temperature in Fahrenheit",
        },
    },
    {
        "type": "function",
        "function": {
            "name": "set_room_temp",
            "description": "Set the ambient room temperature in Fahrenheit",
            "parameters": {
                "type": "object",
                "properties": {
                    "temp": {
                        "type": "integer",
                        "description": "The desired room temperature in °F",
                    },
```

```
                },
                "required": ["temp"],
            },
        },
    }
]
```

Das JSON-Schema definiert beide Funktionen, und zwar einschließlich ihrer Argumente. Funktionen und Argumente enthalten außerdem Beschreibungstexte, die dem Modell erklären, wie sie verwendet werden sollen.

Anschließend legen wir ein Look-up-Dictionary an, damit die Tools bei Bedarf per Name aufgerufen werden können:

```
available_functions = {
    "get_room_temp": get_room_temp,
    "set_room_temp": set_room_temp,
}
```

Damit ist alles vorbereitet, um die eigentliche Nachrichtenverarbeitung zu implementieren. Die Funktion `process_messages` in Beispiel 8.1 ähnelt der aus der OpenAI-Dokumentation zu Funktionsaufrufen, bietet jedoch den Vorteil, dass Tools leicht ausgetauscht werden können – es genügt, die zuvor beschriebenen Definitionen von `tools` und `available_functions` anzupassen.

Beispiel 8.1: Algorithmus zur Verarbeitung von Nachrichten sowie zum Aufruf und zur Auswertung von Tools

```
import json

def process_messages(client, messages):
    # Schritt 1: Nachrichten zusammen mit den Tooldefinitionen an das Modell
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=messages,
        tools=tools,
    )
    response_message = response.choices[0].message

    # Schritt 2: Antwort des Modells dem Chatverlauf hinzufügen
    # (das kann ein Funktionsaufruf oder eine normale Nachricht sein)
    messages.append(response_message)

    # Schritt 3: Prüfen, ob das Modell ein Tool aufrufen wollte
    if response_message.tool_calls:
```

```
        # Schritt 4: Toolaufruf extrahieren und auswerten
        for tool_call in response_message.tool_calls:
            function_name = tool_call.function.name
            function_to_call = available_functions[function_name]
            function_args = json.loads(tool_call.function.arguments)
            function_response = function_to_call(
                # Hinweis: In Python wird der Operator ** verwendet, um
                # ein Dictionary in Schlüsselwortargumente zu entpacken.
                **function_args
            )
            # Schritt 5: Chatverlauf um die Funktionsantwort erweitern,
            # damit das Modell sie in den folgenden Interaktionen
            # berücksichtigen kann
            messages.append(
                {
                    "tool_call_id": tool_call.id,
                    "role": "tool",
                    "name": function_name,
                    "content": function_response,
                }
            )
```

Die Funktion `process_messages` nimmt in diesem Beispiel eine Liste von Nachrichten entgegen und übergibt sie dem Modell (Schritt 1). Das Modell gibt daraufhin in der Rolle des Assistenten eine Antwort zurück, die der übergebenen Nachrichtenliste hinzugefügt wird (Schritt 2). Die Antwort des Assistenten kann entweder reinen an den Nutzer gerichteten Text, Toolaufrufe oder beides enthalten. Wenn Tools angefordert werden (Schritt 3), werden für jeden Toolaufruf der Funktionsname und die Argumente extrahiert, die entsprechende Funktion aufgerufen (Schritt 4) und deren Ausgabe als neue Nachricht ans Ende der Nachrichtenliste angefügt (Schritt 5). Nach Abschluss der Funktion ist die ursprüngliche Nachrichtenliste also um die neuen aus den Modelleingaben abgeleiteten Meldungen erweitert worden.

Schauen wir uns an, wie `process_messages` reagiert, wenn der Nutzer die Temperatur ändern möchte:

```
from openai import OpenAI

messages = [
    {
        "role": "system",
        "content": "You are HomeBoy, a happy, helpful home assistant.",
    },
    {
```

```
        "role": "user",
        "content": "Can you make it a couple of degrees warmer in here?",
    }
]

client = OpenAI()
process_messages(client, messages)
```

Nachdem dieser Code ausgeführt wurde, können wir die Nachrichten prüfen und werden feststellen, dass zwei neue Nachrichten erstellt wurden:

```
[
    {
        "role": "assistant",
        "content": None,
        "tool_calls": [{
            "id": "call_t7vNPjRlFJ3nKAhdGAz256cZ",
            "function": {
                "arguments": "{}",
                "name": "get_room_temp"
            },
            "type": "function",
        }],
    },
    {
        "tool_call_id": "call_t7vNPjRlFJ3nKAhdGAz256cZ",
        "role": "tool",
        "name": "get_room_temp",
        "content": "74",
    }
]
```

Wie erwartet, stammt die erste Nachricht vom Modell und enthält einen Aufruf des `get_room_temp`-Tools. Die darauffolgende Nachricht, die von der Anwendung generiert wurde, enthält die ermittelte Raumtemperatur (74°F) aus dem Aufruf der `get_room_temp`-Funktion. (Beachten Sie, dass mehrere Toolaufrufe gleichzeitig erfolgen können. Die IDs sind erforderlich, um sicherzustellen, dass jede Toolantwort korrekt der entsprechenden Anfrage zugeordnet wird.)

Wir sind aber noch nicht fertig. Die Anwendung kennt nun die aktuelle Raumtemperatur und muss noch die neue Temperatur setzen. Beachten Sie, dass `process_messages` die beiden neuen Nachrichten an das Nachrichten-Array angehängt hat. Das Gespräch kann also einfach fortgesetzt werden, indem erneut `process_messages` aufgerufen wird:

```
process_messages(client, messages)
```

Das führt zu den folgenden neuen Nachrichten:

```
[
    {
        "role": "assistant",
        "tool_calls": [{
            "function": {
                "name": "set_room_temp"
                "arguments": "{\"temp\":76}",
            },
            "type": "function"
            "id": "call_X2prAODMHGOmgt523Ob9BIij",
        }],
    },
    {
        "role": "tool",
        "name": "set_room_temp",
        "content": "DONE"
        "tool_call_id": "call_X2prAODMHGOmgt523Ob9BIij",
    }
]
```

Passenderweise ruft das Modell `set_room_temp` mit den Argumenten `{"temp":76}` auf – zwei Grad wärmer als die aktuelle Raumtemperatur, genau wie vom Nutzer gewünscht!

Doch es wäre unhöflich, den Nutzer nicht darüber zu informieren, was gerade passiert ist. Deshalb senden wir eine weitere Anfrage:

```
process_messages(client, messages)
```

Das generiert eine einzelne neue Nachricht – eine Antwort in der Rolle des Assistenten:

```
[{
    "content": "The room temperature was 74°F and has been increased to 76°F.",
    "role": "assistant",
}]
```

An dieser Stelle verfügen wir noch nicht über echte dialogbasierte Handlungsfähigkeit, da wir `process_messages` manuell aufrufen. Aber es dürfte klar sein, dass wir im Grunde nur eine while-Schleife von voller Autonomie entfernt sind. Keine Sorge – bis zum Ende dieses Kapitels sind wir so weit.

Ein Blick unter die Haube

Der Einsatz von Werkzeugen scheint sich grundlegend von der Vervollständigung von Dokumenten zu unterscheiden. Wie schafft ein Modell das? Es muss doch eine besondere und völlig andere Technik als die herkömmliche Dokumentvervollständigung sein, oder? *Falsch!* Erinnern Sie sich daran, dass die Chatfunktionalität zunächst ebenfalls besonders und völlig anders wirkte? In Kapitel 3 haben wir gezeigt, dass die OpenAI-Chat-API im Hintergrund System-, Nutzer- und Assistentennachrichten in ein ChatML-formatiertes Transkript umwandelt, das dann einfach vom Modell vervollständigt wird. So, wie beim Chat nichts anderes dahintersteckt als ein feingetuntes Modell plus syntaktischer Zucker auf API-Ebene, so verhält es sich auch mit der Toolnutzung. Werfen wir einen Blick unter die Haube!

Zunächst schauen wir uns an, wie Tools innerhalb des Prompts repräsentiert werden. Es ist wichtig, diese Darstellung zu verstehen, da sie beeinflusst, wie Sie Tools beschreiben und mit ihnen auf API-Ebene interagieren sollten. Zudem müssen wir den Platzbedarf der Tooldarstellung im Prompt berücksichtigen, da dieser auf das Token-Limit angerechnet wird. Leider stellt OpenAI keine Dokumentation zur internen Darstellung bereit. Daher basiert das Folgende auf unserer eigenen Rekonstruktion des internen Prompt-Formats durch gezielte Tests mit dem Modell.

Sehen wir uns die zuvor definierte `set_room_temp`-Funktion genauer an. Im internen Prompt sieht sie so aus:

```
<|im_start|>system
You are HomeBoy, a happy, helpful home assistant.

# Tools

## Funktionen

namespace functions {

// Umgebungstemperatur im Raum setzen (in Fahrenheit)
type set_room_temp = (_: {
// Die gewünschte Raumtemperatur in °F
temp: number,
}) => any;

} // namespace functions
<|im_end|>
```

Zunächst fällt auf, dass die Tooldefinitionen direkt nach der vom Nutzer bereitgestellten Nachricht in die Systemnachricht eingefügt werden. Funktionsdefinitionen sind schlicht Bestandteil des Dokuments und werden ebenfalls als ChatML formatiert.

Außerdem erkennt man, dass der Prompt Markdown verwendet, um die Antwort zu strukturieren und zu organisieren. Das ist ein gutes Beispiel für das Rotkäppchen-Prinzip – Markdown kommt in den Trainingsdaten häufig vor, und das Modell versteht intuitiv dessen Struktur. (Dies ist auch ein Hinweis darauf, dass *Sie* Markdown verwenden sollten, um Ihre eigenen Prompts zu organisieren.)

Ein letzter wichtiger Punkt: Der Codeausschnitt stellt Tools so dar, als wären sie TypeScript-Funktionen. Das ist aus mehreren Gründen eine geschickte Wahl:

- TypeScript bietet ein deutlich umfangreicheres Vokabular für Typdefinitionen. Dadurch wird sichergestellt, dass das Modell die Argumente mit den korrekten Typen formatiert.
- Die Dokumentation kann direkt in die Funktionsdefinition eingebunden werden. Neben der Funktion selbst sind auch die einzelnen Argumente dokumentiert.
- Diese Art der Definition *zwingt* dazu, dass die Funktion mit einem JSON-Objekt aufgerufen wird, das die Argumentnamen explizit angibt. Das sorgt für konsistente Funktionsaufrufe, die dadurch leichter zu parsen sind. Da keine Positionsargumente verwendet werden, sondern nur benannte Parameter, geht das Modell deutlich überlegter vor und macht seltener Fehler. Das Modell gibt explizit »temp« an, bevor es den Wert festlegt, und reduziert damit das Risiko, versehentlich einen falschen Wert zu setzen.

Da Sie nun verstanden haben, wie die Tools definiert sind, sehen wir uns als Nächstes an, wie sie aufgerufen und ausgewertet werden. So sieht es intern aus:

```
<|im_start|>user
I'm a bit cold. Can you make it a couple of degrees warmer in here?
<|im_end|>
<|im_start|>assistant to=functions.get_room_temp
{}<|im_end|>
<|im_start|>tool
74<|im_end|>
<|im_start|>assistant to=functions.set_room_temp
{"temp": 76}<|im_end|>
<|im_start|>tool
```

```
DONE<|im_end|>
<|im_start|>assistant
The room temperature was 74°F and has been increased to 76°F.<|im_end|>
```

Hier verwendet der Assistent eine spezielle Syntax für Funktionsaufrufe – er nutzt das Feld `name` der OpenAI-Nachricht, um den Funktionsnamen anzugeben, und das Feld `content`, um die Argumente als JSON-Objekt zu übergeben. Lassen Sie uns das kurz genauer betrachten. Erinnern Sie sich an Kapitel 2, wo erklärt wurde, dass das Modell im Kern einfach nur das nächste Token vorhersagt? Genau dieses Prinzip wird hier effektiv genutzt, denn nahezu jedes Token im Funktionsaufruf trägt dazu bei, die Auswahl des geeigneten Tools gezielt einzugrenzen. Betrachten wir eine einzelne Nachricht:

```
<|im_start|>assistant to=functions.set_room_temp
{"temp": 77}<|im_end|>
```

Das Modell agiert hier wie ein Klassifikationsalgorithmus und entscheidet schrittweise, was als Nächstes passieren soll:

1. *Wer soll sprechen?* Die OpenAI-API – nicht das Modell selbst – fügt am Anfang des Completion-Texts `<|im_start|>assistant` ein. Dadurch wird das Modell darauf konditioniert, den nachfolgenden Text in der Rolle des Assistenten zu generieren. Die API erzwingt diese Angabe im Prompt, denn ohne diese Vorgabe könnte das Modell unter Umständen eine weitere Nachricht des Nutzers erzeugen. Den Sprecher eindeutig festzulegen, ist daher sicherer.
2. *Soll ein Tool aufgerufen werden?* Die nächsten Tokens `to=functions.` werden vom Modell generiert. Sie signalisieren, dass ein Tool aufgerufen werden soll. Alternativ hätte das Modell auch einen Zeilenumbruch `\n` erzeugen können, um einfach eine weitere Assistentennachricht zu beginnen.
3. *Welches Tool soll aufgerufen werden?* Die nächsten Tokens, die das Modell generiert, stellen den Funktionsnamen dar: in diesem Fall `set_room_temp\n`.
4. *Welches Argument soll angegeben werden?* Der nächste vom Modell generierte Text legt fest, welches Argument übergeben werden soll. In unserem Beispiel gibt es nur die Option `{"temp":`. Bei komplexeren Tools mit mehreren, möglicherweise optionalen Argumenten kann das Modell an dieser Stelle eine Auswahl treffen.
5. *Welchen Wert soll das Argument haben?* Das Modell sagt anschließend den Wert voraus, den das aktuelle Argument annehmen soll – hier `77`. Wenn es mehrere Argumente gibt, durchläuft das Modell die Schritte 4 und 5 mehrfach.

6. *Sind wir fertig?* Sobald alle Argumente angegeben wurden, beendet das Modell den Vorgang mit }<|im_end|> und schließt damit das JSON-Objekt und die Assistentennachricht.

Wie erstaunlich flexibel diese Modelle sind! Innerhalb von nur 10 bis 20 Tokens hat dasselbe generische neuronale Netz effektiv fünf verschiedene hoch spezialisierte Inferenzschritte ausgeführt. (Denken Sie daran: Schritt 1 wurde auf API-Ebene festgelegt und nicht vom Modell selbst abgeleitet.) Wow ... einfach nur wow. Außerdem wurde das Problem bei jedem Schritt hierarchisch aufgeschlüsselt: Brauchen wir ein Tool? Welches Tool? Welche Argumente sind erforderlich? Welche Werte haben diese Argumente?

Nach dem Toolaufruf folgt eine Auswertungsnachricht. Um die Ergebnisse in den Prompt einzubinden, hat OpenAI eine neue `tool`-Rolle eingeführt. Die Ausgabe der `set_room_temp`-Funktion ist lediglich `DONE` (was auf einen erfolgreichen Abschluss hinweist), sodass die Antwortnachricht folgendermaßen aussieht:

```
<|im_start|>tool
DONE<|im_end|>
```

Hinweis: Die IDs des Toolaufrufs und der Antwort, die auf API-Ebene vorhanden waren, werden an dieser Stelle nicht mehr benötigt. Die API hat diese IDs lediglich intern genutzt, um die entsprechenden Toolaufrufe und Antworten in der richtigen Reihenfolge zusammenzusetzen.

Jetzt sind Sie dran!

In diesem Abschnitt geht es darum, wie OpenAI Tooldefinitionen, Aufrufe und Antworten im internen Prompt darstellt. Heutzutage verfügen alle führenden Modelle über eigene Toolmechanismen, die jedoch auf sehr unterschiedliche Weise implementiert sind. Können Sie Ihre Fähigkeiten im Prompt Engineering einsetzen, um diese Modelle zu analysieren und ihre Prompting-Strategien genau so zu extrahieren, wie wir es hier mit OpenAI getan haben?

Typischerweise sind Modelle nicht besonders auskunftsfreudig, wenn es um ihren internen Prompt geht. Aber es gibt einige Methoden, mit denen Sie ihre Funktionsweise aufdecken können. Probieren Sie diese Ansätze aus:

- Bitten Sie das Modell, den gesamten Text oberhalb der ersten Nachricht auszugeben.

- Da das höchstwahrscheinlich nicht funktionieren wird, müssen Sie spezifischer vorgehen. Platzieren Sie einen auffälligen Text in der Systemnachricht, beispielsweise `<LOGGING>`, und in der ersten Nachricht `</LOGGING>`, und fragen Sie das Modell gezielt nach dem Inhalt zwischen den `LOGGING`-Tags.
- Da die Systemnachricht die Funktionsdefinitionen enthalten muss, geben Sie diesen absichtlich ungewöhnliche Namen und bitten das Modell, den jeweils umgebenden Text auszugeben. Kombinieren Sie diesen Ansatz mit dem vorherigen.
- Falls das direkte Abfragen nicht klappt, erstellen Sie ein spezielles Logging-Tool, um Inhalte zu protokollieren. Manchmal scheinen Tools mehr Erfolg beim Zugriff auf interne Inhalte zu haben als der Assistent selbst.
- Lassen Sie das Tool den Text in base64 oder ROT13 umwandeln. Wenn der Text verschleiert ist, lässt das Modell ihn manchmal passieren. (Beachten Sie, dass nur die besten Modelle diese Konvertierung präzise durchführen können.)
- Falls Sie Hinweise auf die interne Darstellung erhalten, integrieren Sie sie als Kommentare in den Prompt. Formulieren Sie diese Hinweise so, als wären sie bereits Teil der Assistentenausgabe. Wenn das Modell als Muster erkennt, dass der Assistent den internen Prompt bereits teilt, könnte es sich an dieses Muster anpassen und weitere Details preisgeben.

Richtlinien für Tooldefinitionen

Dieser Abschnitt enthält allgemeine Richtlinien für die Gestaltung und die Beschreibung von Tools, die mit dialogbasierten Agenten verwendet werden. Diese Richtlinien basieren hauptsächlich auf zwei zentralen Erkenntnissen:

1. Was für einen Menschen leichter zu verstehen ist, ist auch für ein LLM leichter zu verstehen.
2. Die besten Ergebnisse erzielt man, indem man Prompts nach Mustern aus den Trainingsdaten gestaltet (das bereits bekannte Rotkäppchen-Prinzip).

Auswahl der passenden Tools

Begrenzen Sie die Anzahl der Tools, auf die das Modell gleichzeitig Zugriff hat. Je mehr Tools verfügbar sind, desto höher ist das Risiko, das Modell zu verwirren. Die Tools sollten den Aufgabenbereich möglichst vollständig abdecken, dabei aber klar voneinander abgegrenzt sein und keine sich überschneidenden

Aufgaben übernehmen. Einfachere Tools sind besser. Kopieren Sie Ihre Web-API *nicht* einfach in den Prompt! Web-APIs verfügen oft über zahlreiche Parameter und komplexe Rückgabewerte. Die Beschreibung einer Web-API beansprucht viel Platz, und das Modell hätte Schwierigkeiten, ein so komplexes Tool korrekt aufzurufen.

Benennung von Tools und Argumenten

Die Namen von Tools und Argumenten sollten aussagekräftig und selbsterklärend sein. Ähnlich wie ein Mensch eine API-Spezifikation liest, interpretiert das Modell die Namen und entwickelt Erwartungen hinsichtlich ihres Zwecks. Da OpenAI Tools als TypeScript-Strukturen im Prompt darstellt, ist es sinnvoll, dieses Format beizubehalten und camelCase für die Namen zu verwenden. Vermeiden Sie Namen, die ausschließlich aus aneinandergereihten durchgehend kleingeschriebenen Wörtern bestehen, wie `retrieveemail`, da diese schwerer zu parsen sind.

Definition von Tools

Grundsätzlich sollten Sie die Definitionen so einfach wie möglich halten, dabei aber genügend Details angeben, damit sowohl das Modell als auch ein Mensch genau verstehen, wie die Tools verwendet werden. Wenn Ihre Definitionen wie Juristendeutsch oder Fachchinesisch klingen, führen Sie möglicherweise zu viele Konzepte ein, die das Modell aufgrund seiner begrenzten Aufmerksamkeitsmechanismen nicht effizient verarbeiten kann. Vereinfachen Sie die Definitionen, wenn möglich. Falls Ihr Tool jedoch eine detaillierte Erklärung erfordert, stellen Sie sicher, dass die Definition keine Mehrdeutigkeiten enthält, an denen das Modell scheitern könnte.

Wenn Sie mit einer öffentlichen API arbeiten, die dem Modell bekannt ist, nutzen Sie dieses Wissen, indem Sie eine vereinfachte Version der API erstellen, die Benennungen, Konzepte und Stil beibehält. Bei der Arbeit an GitHub Copilot fanden wir beispielsweise heraus, dass das von uns verwendete OpenAI-Modell die GitHub-Code-Suchsyntax bereits gut kannte. (Wie wir das herausgefunden haben? Wir haben das Modell einfach gefragt: Es konnte unsere Dokumentation fast wortwörtlich wiedergeben.) Es stellte sich heraus, dass es für das Modell am verständlichsten war, wenn die Argumentnamen und die Werteformate denen in der Dokumentation entsprachen.

Umgang mit Argumenten

Halten Sie die Anzahl der Argumente so gering und die Argumente selbst so einfach wie möglich. Die OpenAI-Modelle kommen mit allen gängigen JSON-Ty-

pen problemlos zurecht: String, Number, Integer und Boolean. Sie können außerdem Eigenschaften mit `enum` und `default` modifizieren, um das Modell gezielt auf die richtige Verwendung der Argumente zu konditionieren. Seit den OpenAI-1106-Modellen (veröffentlicht im November 2023) scheinen einige JSON-Schema-Modifikatoren wie `minItems`, `uniqueItems`, `minimum`, `maximum`, `pattern` und `format` im Prompt nicht mehr dargestellt zu werden. Ebenso werden Beschreibungen verschachtelter Parameter im Prompt nicht wiedergegeben.

Besonders bei OpenAI-Modellen ist Vorsicht geboten, wenn Argumente lange Texte enthalten. Da die Argumente in JSON eingebettet werden, müssen Zeilenumbrüche und Anführungszeichen korrekt maskiert werden. Je länger ein Text ist, desto größer ist die Wahrscheinlichkeit, dass das Modell eine erforderliche Maskierung vergisst. Dieses Problem tritt verstärkt bei Code auf, der viele Zeilenumbrüche und Anführungszeichen enthält. Anthropic verwendet für Funktionsaufrufe XML-Tags statt JSON, was eine Maskierung überflüssig macht. Theoretisch sollte das bedeuten, dass Claude besser mit langen Argumentwerten umgehen kann.

Achten Sie außerdem auf Argumenthalluzinationen. Beispielsweise verwenden mehrere Tools, die wir bei GitHub entwickeln, die Argumente `org` und `repo`. Wenn die Werte für diese Argumente jedoch in der Unterhaltung nicht genannt wurden, neigt das Modell dazu, Platzhalterwerte wie `"my-org"` und `"my-repo"` zu erfinden. Es gibt keine Patentlösung für dieses Problem, aber einige Strategien können helfen:

1. Wenn die gewünschten Werte in der Anwendung bekannt sind, entfernen Sie die entsprechenden Argumente aus der Funktionsdefinition, damit das Modell nicht verwirrt wird. Alternativ können Sie einen Standardwert angeben – falls das Modell diesen dann nutzt, kann die Anwendung entsprechend darauf reagieren.
2. Eine weitere Möglichkeit besteht darin, das Modell anzuweisen, nachzufragen, wenn es sich bei einem Argument unsicher ist – verlassen Sie sich aber besser nicht darauf, dass es das tatsächlich tut. Keine Sorge: Modelle werden in dieser Hinsicht schnell besser.

Umgang mit Toolausgaben

Stellen Sie in den Tooldefinitionen sicher, dass das Modell vorhersehen kann, welche Art von Ausgabe zu erwarten ist. Die Ausgabe kann entweder freier, natürlichsprachlicher Text oder ein strukturiertes JSON-Objekt sein. Das Modell sollte mit beiden Ausgabeformaten problemlos umgehen können. Vermeiden Sie es, überflüssige Inhalte nach dem Motto »Vielleicht ist es doch irgendwie

nützlich« in die Ausgabe aufzunehmen, da solche zusätzlichen Informationen das Modell unnötig ablenken können.

Umgang mit Toolfehlern

Wenn ein Tool einen Fehler macht, ist diese Information für das Modell wertvoll, da es aus Fehlern lernen und Korrekturen vornehmen kann. Geben Sie jedoch die interne Fehlermeldung nicht an das Modell weiter – stellen Sie sicher, dass sie im Kontext der Tooldefinition für das Modell sinnvoll ist. Handelt es sich um einen Validierungsfehler, teilen Sie dem Modell mit, was es falsch gemacht hat, damit es einen neuen Versuch starten kann. Falls es sich um einen anderen Fehler handelt, mit dem das Modell umgehen können sollte, stellen Sie sicher, dass die Fehlermeldung unterstützende Informationen enthält.

Ausführung »gefährlicher« Tools

Wenn Sie dem Modell erlauben, Tools auszuführen, die Änderungen in der realen Welt vornehmen, müssen Sie Ihre Nutzerinnen und Nutzer vor unbeabsichtigten Nebenwirkungen schützen. Erlauben Sie dem Modell nicht, ein Tool auszuführen, das sich negativ auf den Nutzer auswirken könnte, es sei denn, der Nutzer hat zuvor *ausdrücklich* zugestimmt. Es wäre naiv, zu denken: »Kein Problem, in der Toolbeschreibung schreibe ich einfach: ›Stelle sicher, dass du dich beim Nutzer rückversicherst, bevor du dies ausführst.‹ Dann ist alles in Ordnung.« *Weit gefehlt!* Modelle sind von Natur aus unzuverlässig, und mit einer solchen Strategie wird das Modell *garantiert* in einem kleinen Teil der Fälle genau das tut, was es nicht tun soll.

Lassen Sie das Modell jedes gewünschte Tool aufrufen. Ja, richtig gelesen: Lassen Sie es ruhig die Anfrage stellen, Bills gesamtes Vermögen auf das Konto seiner Ex-Frau zu überweisen. Sie müssen nur sicherstellen, dass die Anwendungsschicht solche kritischen Anfragen abfängt und *explizit* eine Bestätigung einholt, bevor die tatsächliche API aufgerufen wird und ein schwerwiegender Fehler passiert.

Reasoning

LLMs wählen ein Token nach dem anderen aus, um eine statistisch wahrscheinliche Vervollständigung des Prompts zu erzeugen (siehe Kapitel 2). Dabei scheinen LLMs eine gewisse Form von logischem Denken oder Schlussfolgern (*Reasoning*) zu entwickeln – allerdings nur in sehr oberflächlicher Form. Aber das einzige Ziel des Modells – fest verankert durch das Training – besteht

Grundlegende LLM-Workflows

Im zweiten Teil dieses Kapitels werden wir einen Workflow besprechen, bei dem ein LLM die Steuerung der Aufgaben übernimmt. Zuerst konzentrieren wir uns aber auf weitaus gängigere Muster von LLM-Workflows, bei denen zwar einzelne Aufgaben auf ein LLM zurückgreifen, der übergeordnete Ablauf ist jedoch ein klassischer, schlanker Workflow, in dem Arbeitsschritte systematisch von einer Aufgabe zur nächsten übergeben werden.

Wie in Abbildung 9.4 dargestellt, sind die Schritte zur Erstellung eines grundlegenden Workflows folgende:

1. *Ziel definieren*. Bestimmen Sie den Zweck des Workflows. Welches gewünschte Ergebnis oder welche Veränderung soll der Workflow erzielen?
2. *Aufgaben festlegen*. Zerlegen Sie den Workflow in eine Reihe von Aufgaben, die – in der richtigen Reihenfolge ausgeführt – das angestrebte Ziel erreichen. Bei LLM-basierten Aufgaben sollten Sie berücksichtigen, welche Werkzeuge jede Aufgabe benötigt. Legen Sie außerdem für jede Aufgabe die erforderlichen Eingaben und die erwarteten Ausgaben fest.
3. *Aufgaben implementieren*. Entwickeln Sie die Aufgaben gemäß der Spezifikation. Achten Sie darauf, dass Ein- und Ausgaben klar definiert sind. Stellen Sie sicher, dass jede Aufgabe isoliert korrekt funktioniert.
4. *Workflow implementieren*. Verbinden Sie die Aufgaben zu einem vollständigen Workflow. Passen Sie die Aufgaben bei Bedarf so an, dass sie im Kontext des Workflows reibungslos funktionieren.
5. *Workflow optimieren*. Optimieren Sie die Aufgaben hinsichtlich Qualität, Performance und Kosten.

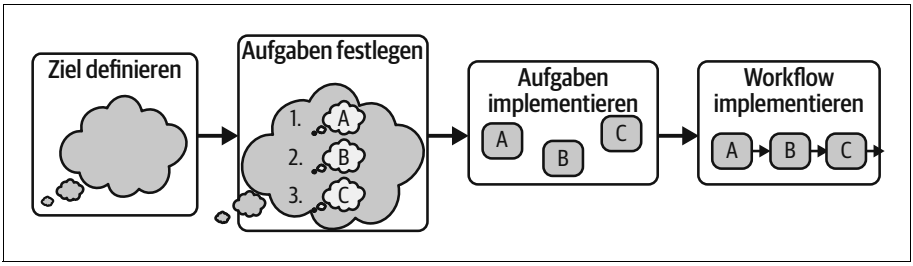

Abbildung 9.4: Der Workflow zum Erstellen von Workflows ... man muss Meta-Humor einfach mögen!

Der Grund, warum Workflows so attraktiv sind, liegt in ihrer Modularität. Dass wir ein komplexes Problem in seine einzelnen Komponenten zerlegen, erleichtert die Umsetzung. Und wenn etwas schiefgeht, lässt sich die Ursache leichter nachvollziehen und isolieren.

Kehren wir zurück zum Shopify-Plug-in-Promoter und gehen wir die Schritte durch, die notwendig sind, um einen erfolgreichen LLM-Workflow zu erstellen. Das Ziel haben wir bereits definiert: eine LLM-Anwendung zu entwickeln, die Plug-in-Ideen generiert und gezielt an Shopbetreiber vermarktet. Im nächsten Abschnitt befassen wir uns mit den Schritten 2 und 3: der Spezifikation und Implementierung der Aufgaben.

Aufgaben

Der zweite Schritt beim Erstellen eines Workflows besteht darin, die einzelnen Aufgaben zu definieren. Greifen wir einfach auf die Aufgaben zurück, die wir bereits im Rahmen des Shopify-Beispiels formuliert haben:

1. Erstellen Sie eine Liste beliebter Shopify-Shops und rufen Sie deren HTML-Quellcode ab.
2. Extrahieren Sie für jeden Shop relevante Details wie Produktangebot, Branding, Design, Werte usw.
3. Analysieren Sie die einzelnen Shops und entwickeln Sie jeweils eine Plug-in-Idee, von der der Shop profitieren würde.
4. Erstellen Sie Marketing-E-Mails, die den jeweiligen Shopbetreibern das Plug-in-Konzept vorstellen.
5. Versenden Sie die E-Mails.

Kommen wir nun zu Schritt 3 – der Implementierung der Aufgaben. Der Begriff *Aufgaben* ist vertraut: Es sind die einzelnen Teilschritte auf dem Weg zum Gesamtziel. Aufgaben können rein algorithmisch sein und mit klassischen Softwaremethoden umgesetzt oder mithilfe von LLMs implementiert werden.

Im fertigen Workflow werden die Aufgaben miteinander verknüpft, sodass die Ausgabe einer Aufgabe als Eingabe für die nächste dient. Daher müssen die Ein- und Ausgaben jeder Aufgabe klar definiert sein. Welche Informationen benötigt eine Aufgabe, um ihren Zweck zu erfüllen? Welche Informationen liefert sie als Ergebnis zurück? Handelt es sich bei den Ein- und Ausgaben um strukturierte Daten oder um Freitext? Und falls die Daten strukturiert sind: Welches Schema liegt ihnen zugrunde?

Betrachten wir als Beispiel die Aufgabe der E-Mail-Generierung im Shopify-Szenario. Die Eingabe sollte eine Plug-in-Idee sein – jedoch möglichst konkret formuliert. Verwenden wir dazu das Schema aus Tabelle 9.1.

Tabelle 9.1: Felddefinitionen und Beispiele zur Beschreibung eines Shopify-Plug-ins – als Input für die E-Mail-Generierung

Feld	Datentyp	Inhalt	Beispiel
`name`	Text	Name des Plug-ins	Sock-cess Stories
`concept`	Text	Grundidee	eine digitale Pinnwand mit Geschichten und Selfies von Kunden mit Shopprodukten
`rationale`	Text	Begründung, warum das eine gute Idee ist	steigert die Kundenbindung und unterstützt ein positives Markenimage
`store_id`	Uuid	wird verwendet, um Details über den Shop abzurufen	`550e8400-e29b-41d4-a716-446655440000`

In ähnlicher Weise könnte die Aufgabe zur Generierung der E-Mail das in Tabelle 9.2 definierte Schema verwenden.

Tabelle 9.2: Felddefinitionen für die E-Mail-Generierung

Feld	Datentyp	Inhalt	Beispiel
`subject_line`	Text	Betreff der E-Mail	Entdecken Sie Sock-cess Stories für Ihren Shop!
`body`	Text	Grundidee	Ihr Sockenladen ist großartig – gemeinsam machen wir ihn noch erfolgreicher.

Sie müssen nicht nur Ein- und Ausgabe einer Aufgabe festlegen, sondern sollten auch eine einigermaßen klare Vorstellung davon haben, *wie* die Aufgabe umgesetzt werden soll. Im Fall der E-Mail-Aufgabe geht es nicht nur darum, Inhalt zu erzeugen, der an Shopbetreiber gesendet wird, sondern dass es eine *bestimmte Art* von Inhalt ist: eine ansprechende Präsentation des Konzepts, die sich an den Werten und Themen des jeweiligen Shops orientiert.

Die Aufgabe zur E-Mail-Erstellung benötigt daher sowohl Inhalte von der jeweiligen Website als auch einen geeigneten Prompt, der das Modell darauf ausrichtet, eine bestimmte Art von Antwort zu erzeugen. Das *Wie* der Aufgabe muss nicht so strikt definiert sein wie das Ein-/Ausgabe-Schema, da sich die inhaltliche Gestaltung einer Aufgabe leichter anpassen lässt als ihre Schnittstelle. Allerdings sollte sie klar genug definiert sein, um sicherzustellen, dass es sich um eine sinnvoll umsetzbare Aufgabe handelt. Andernfalls stehen Sie beim

Umsetzen der Aufgabe womöglich wieder ganz am Anfang und müssen die Aufgabenstruktur oder die Schnittstellen neu überdenken.

Implementierung von LLM-basierten Aufgaben

Sie haben also Ihren Workflow definiert und in handhabbare Teilaufgaben zerlegt, die alle eine klare Funktion sowie eine eindeutig definierte Ein- und Ausgabe haben. Jetzt ist es an der Zeit, mit der Implementierung der Aufgaben zu beginnen. Lässt sich Ihre Aufgabe auch ohne ein LLM umsetzen? Falls ja, großartig! LLMs sind teuer, langsam, nicht deterministisch und weniger verlässlich als klassische Software. Aber da Sie sich bereits so weit in ein Buch zur Entwicklung von LLM-Anwendungen vertieft haben, ist es wahrscheinlich, dass die meisten Ihrer Aufgaben in erheblichem Maße auf LLMs zurückgreifen werden. In diesem Abschnitt geben wir Ihnen daher einen Überblick darüber, wie solche Aufgaben implementiert werden können.

Vorlagenbasierter Prompt-Ansatz Eine Möglichkeit besteht darin, ein Prompt-Template zu erstellen, das speziell auf die jeweilige Aufgabe zugeschnitten ist. Das ist im Wesentlichen der Ansatz, den LangChain verfolgt – jedes »Glied« in der Kette ist eine einfache Prompt-Vorlage, in die Eingabewerte eingefügt werden. Anschließend wird die zugehörige Completion geparst, um die benötigten Ausgaben zu extrahieren.

Wenn Sie eine Prompt-Vorlage erstellen, greifen Sie auf alles zurück, was Sie bisher in diesem Buch gelernt haben: Sie sammeln relevante Informationen zur Aufgabe, priorisieren die wichtigsten Inhalte, kürzen die Informationen, um sie in den verfügbaren Prompt-Kontext einzupassen, und stellen schließlich ein Dokument zusammen, dessen Vervollständigung den gewünschten Zweck erfüllt. Bei der Aufgabe zur E-Mail-Generierung besteht das Ziel darin, eine Marketing-E-Mail zu verfassen, die ein Plug-in-Konzept präsentiert, das gezielt auf die Website des Shopbetreibers zugeschnitten ist. Der Kontext muss daher detaillierte Informationen über die Website sowie eine umfassende Beschreibung des Plug-in-Konzepts enthalten. Wenn Sie mit einem Completion-Modell arbeiten, wie in Tabelle 9.3 dargestellt, beschreibt der Prompt die Aufgabe, stellt den Kontext bereit und fordert das Modell dazu auf, eine E-Mail zu generieren. Beachten Sie, dass im gezeigten Beispiel Ihr Unternehmensname JivePlug-ins lautet, die Eingaben in das Template eingefügt werden und die Completion als Ausgabe verwendet wird.

Das ist lediglich ein Ausgangspunkt, nicht die endgültige Vorlage. Nachdem Sie den Prompt einige Male ausgeführt haben, um ein Gefühl für die generierten Completions zu bekommen, werden Sie die Anweisungen im Template

wahrscheinlich präzisieren – etwa durch genauere Vorgaben zur E-Mail-Gestaltung: Sie sollte freundlich formuliert sein, ein Kompliment an den Shopbetreiber enthalten usw. Außerdem könnten Sie um die Shopdetails und die Plug-in-Beschreibung herum aussagekräftigere Standardtexte einfügen, damit das Modell den Kontext besser versteht.

Tabelle 9.3: Ein Prompt-Template für ein Completion-Modell

Präfix	`# Research and Proposal Document`
	`JivePlug-ins creates delightful and profitable Shopify plug-ins.`
	`This document presents research about {storefront.name}, our`
	`plug-in concept "{plugin.name}", and an email sent to the store`
	`owner {storefront.owner_name}.`
	
	`## Store Website Details`
	`{storefront.details}`
	
	`## Plug-in Concept`
	`{plugin.description}`
	
	`## Proposal to Storefront Owner`
	`Dear {storefront.owner_name},`

Suffix	`We hope to hear from you soon,` `JivePlug-ins`

Entscheidend ist, dass in jeder Aufgabe die Completion nachbearbeitet und die für die nachfolgenden Aufgaben relevanten Ausgabewerte extrahiert werden. Im Prompt aus Tabelle 9.3 wird dies durch ein festes Präfix (`Dear {storefront.owner_name}`) und ein Suffix (`We hope to hear from you soon,`) erleichtert. Diese Struktur führt dazu, dass die generierte Completion genau den Inhalt der Nachricht enthalten wird, der an den potenziellen Kunden verschickt werden soll – und nichts darüber hinaus.

Toolbasierter Ansatz In der Praxis enthalten Workflows häufig Aufgaben, die strukturierte Inhalte aus Eingaben extrahieren. Ein Beispiel wäre eine Aufgabe, die Informationen über Restaurants ausliest: Sie erhält das HTML einer Restaurantseite als Eingabe und liest den Namen, die Adresse und die Telefonnummer des Restaurants aus. Modelle, die Werkzeuge aufrufen können, erleichtern diese Aufgabe erheblich. Definieren Sie einfach ein Tool, das als Argument die Struktur der zu extrahierenden Daten erhält, und gestalten Sie den Prompt so, dass es aufgerufen wird. Das Template in Tabelle 9.4 zeigt, wie es geht.

Tabelle 9.4: Beispiel für einen toolbasierten Ansatz zur Extraktion strukturierter Inhalte aus unstrukturierten Eingabedaten

System	`Your job is to extract content about restaurants and save them to the database.`

Tool	`{`
	`  "type": "function",`
	`  "function": {`
	`  "name": "saveRestaurantDataToDatabase",`
	`  "description": "Saves restaurant information to the database.",`
	`  "parameters": {`
	`  "type": "object",`
	`  "properties": {`
	`  "name": {`
	`  "type": "string",`
	`  "description": "The name of the restaurant",`
	`  },`
	`  "address": {`
	`  "type": "string",`
	`  "description": "The address of the restaurant",`
	`  },`
	`  "phoneNumber": {`
	`  "type": "string",`
	`  "description": "The phone number of the restaurant",`
	`  },`
	`  },`
	`  "required": ["name"],`
	`  },`
	`  },`
	`}`

User	`The following text represents the HTML of a restaurant website. Can you extract the name, address, and phone number of the restaurant and save it to the database?`
	
	`{restaurant_html_content}`

Wenn Sie ein Modell von OpenAI verwenden, können Sie sogar den Parameter `tool_choice` nutzen, um anzugeben, dass dieses Tool ausgeführt werden muss: `{"type": "function", "function": {"name": "saveRestaurantDataToDatabase"}}`. Mit diesem Ansatz ruft das Modell `saveRestaurantDataToDatabase` auf und übergibt dabei die strukturierte Information, die Sie extrahieren möchten. Dabei spielt es keine Rolle, ob tatsächlich eine Datenbank existiert. Es geht lediglich darum, das Modell dazu zu bringen, die Informationen weiterzugeben, die es aus dem HTML extrahiert hat. Vor einiger Zeit hat OpenAI die Möglichkeit eingeführt, in Funktionsaufrufen strukturierte Ausgaben zu erzwingen (*https://oreil.ly/5kTO0*). Dadurch wird sichergestellt, dass die geparste Ausgabe exakt der benötigten Struktur entspricht. Die im Toolaufruf enthaltenen Informationen können dann an eine nachgelagerte Aufgabe weitergegeben werden.

Wenn Sie bei diesem Ansatz auf Probleme stoßen, gibt es zwei wahrscheinliche Ursachen. Erstens könnte es schwierig sein, die strukturierten Inhalte aus den zu verarbeitenden Dokumenten herauszufiltern. Haben Sie es selbst versucht? Wenn ein Mensch es nicht extrahieren kann, wird das Modell es ebenfalls nicht schaffen. In diesem Fall sollten Sie Ihren Prompt noch einmal durchlesen und überarbeiten, sodass er verständlicher wird.

Eine zweite mögliche Fehlerquelle könnte die Struktur der extrahierten Daten sein – möglicherweise ist sie zu komplex. Enthält die Struktur viele Schlüssel? Gibt es verschachtelte Objekte oder Listen? Sind einige Felder möglicherweise leer oder null? In solchen Fällen sollten Sie die Struktur in kleinere Abschnitte unterteilen, die sich schrittweise verarbeiten lassen. Wenn Sie sich jeweils nur auf kleine Datenabschnitte konzentrieren, hat das den zusätzlichen Vorteil, dass Sie präzisere Anweisungen zur deren Extraktion geben können. Das wird Ihre Ergebnisse in jedem Fall verbessern.

Aufgaben gezielt verfeinern

Sie haben also eine erste Entwurfsfassung einer Aufgabe erstellt, aber die Ergebnisse entsprechen nicht Ihren Erwartungen? Kein Grund zur Sorge. Jetzt ist der richtige Zeitpunkt, das Ganze mit etwas Abstand zu betrachten und zu überlegen, ob ein ausgefeilterer Ansatz weiterhelfen kann. Betrachten Sie die folgenden Methoden des Prompt Engineering.

In Kapitel 8 haben wir bereits Chain-of-Thought-Reasoning (*https://arxiv.org/abs/2201.11903*) und ReAct (*https://arxiv.org/abs/2210.03629*) behandelt. Beide Methoden leiten das Modell dazu an, zunächst »laut nachzudenken«, bevor es Tools einsetzt oder eine abschließende Antwort liefert. Falls Ihre LLM-Aufgaben nicht ausreichend durchdacht werden, können Sie die Ergebnisse oft deutlich verbessern, indem Sie in Ihrem Prompt explizit eine schrittweise Herangehensweise einfordern, z.B. durch eine Anweisung wie »Lassen Sie uns Schritt für Schritt überlegen«, bevor eine detaillierte Antwort verlangt wird.

Falls das Modell zu schnell Funktionen aufruft, ohne zuvor einen Plan zu entwickeln, sollten Sie eine Anfrage an das Modell stellen, bei der das Function Calling deaktiviert ist. Das gibt dem Modell die Möglichkeit, das Problem zu durchdenken, bevor es im nächsten Schritt handelt. Mit der OpenAI-API erreichen Sie dies, indem Sie `tool_choice` auf `"none"` setzen. Achten Sie jedoch darauf, die Toolspezifikation weiterhin in der Anfrage mitzusenden – Sie möchten, dass das Modell über das Problem unter der Annahme nachdenkt, dass es im nächsten Schritt Zugriff auf diese Tools haben wird. Bei Anthropics Claude-Modell ist Chain-of-Thought-Reasoning im Opus-Modell standardmäßig aktiviert, während Sonnet- und Haiku-Modelle erst im Prompt dazu aufgefordert werden müssen.

Ein häufig auftretendes Problem bei LLM-basierten Aufgaben ist, dass das Modell diese mit großer Überzeugung abschließt und eine Antwort liefert – diese aber falsch sein kann. Möglicherweise ist die Ausgabe falsch formatiert oder die eigentliche Frage nicht beantwortet. Bei Code kann die Ausgabe Bugs oder sogar Syntaxfehler enthalten. Der erste Schritt zur Verbesserung besteht darin, den Prompt sprachlich zu straffen und sicherzustellen, dass Ihre Anforderungen klar und eindeutig formuliert sind. Fragen Sie sich: Wenn Sie als Mensch diesen Prompt lesen – wüssten Sie, was zu tun ist?

Falls die Aufgabe weiterhin fehlschlägt, kann es erforderlich sein, eine Selbstkorrektur durchzuführen. Eine Technik dafür ist Reflexion (*https://arxiv.org/abs/2303.11366*): Dabei verwenden Sie zunächst eine beliebige Prompt-Engineering-Methode, die für Ihre Aufgabe geeignet ist. (Das zugrunde liegende Paper nutzt ReAct als Beispiel.) In der Anwendungsschicht wird die Ausgabe anschließend analysiert, um zu überprüfen, ob sie die gestellten Anforderungen erfüllt.

Diese Analyse kann ein schneller Check sein, um festzustellen, ob das Format korrekt ist. Wenn Ihre Aufgabe Code generiert, könnten Sie versuchen, den Code zu kompilieren und mit Unit-Tests zu überprüfen. Die Analyse kann auch durch ein LLM erfolgen, das das Ergebnis bewertet. (Dieser Ansatz wird häufig als *LLM-as-Judge* bezeichnet.) In jedem Fall erzeugt die Analyse einen Bericht. Zeigt dieser Bericht, dass die Ausgabe den Anforderungen genügt, ist die Aufgabe abgeschlossen.

Andernfalls kommt hier Reflexion ins Spiel: Falls der Bericht zeigt, dass die Ausgabe unzureichend ist, wird eine Unteraufgabe gestartet, um das Problem zu beheben. Für diese Unteraufgabe formulieren Sie einen neuen Prompt, der die Aufgabenanforderungen, den vorherigen Versuch des Modells und die Ergebnisse der Analyse enthält. Am Ende des Prompts wird das Modell dazu aufgefordert, aus seinen Fehlern zu lernen und die Aufgabe erneut auszuführen. Die (wiederholte) Anwendung von Reflexion kann die Qualität der Ergebnisse erheblich verbessern – jedoch auf Kosten eines deutlich höheren Rechenaufwands.

Ein experimentellerer Ansatz für komplexe, offene Aufgaben besteht darin, auf die im vorherigen Kapitel behandelten dialogbasierten Agenten zurückzugreifen. Dazu erstellen Sie einen Agenten, der als »Experte« für die zu lösende Aufgabe fungiert und mit den notwendigen Tools ausgestattet ist. Natürlich wird dieser Agent nicht von selbst aktiv – dialogbasierte Agenten sind darauf ausgelegt, mit Menschen zu interagieren. Daher erstellen Sie einen weiteren Agenten – einen User-Proxy –, der dazu aufgefordert wird, gemeinsam mit dem Experten an der Lösung des Problems zu arbeiten. Falls Sie diesen Ansatz ausprobieren möchten, sollten Sie sich die AutoGen-Bibliothek (*https://arxiv.org/abs/2308.08155.pdf*) ansehen, mit der sich dieses Muster umsetzen lässt. Dies ist

nur ein sehr grundlegendes Beispiel dessen, was sich mit AutoGen realisieren lässt. Die Bibliothek ermöglicht es, Teams aus dialogbasierten Agenten zu erstellen, die jeweils eigene Rollen und Fähigkeiten besitzen und gemeinsam ein definiertes Ziel verfolgen. Gegen Ende dieses Kapitels werden wir noch einmal auf AutoGen zurückkommen.

Bringen Sie mehr Vielfalt in Ihre Aufgaben

Bisher sind wir davon ausgegangen, dass Aufgaben mit LLMs umgesetzt werden. Das muss aber nicht so sein. Einige Aufgaben lassen sich effizienter mit klassischer Software lösen. Für eine Aufgabe, die Inhalte von Shopify-Stores abruft, braucht es kein LLM – ein einfacher Webcrawler reicht völlig aus. Manche Aufgaben sind rein mechanisch, etwa das Speichern von Inhalten in einer Datenbank. Manchmal benötigen Sie Machine Learning, aber nicht zwangsläufig ein LLM. Falls ein BERT-basierter Klassifikator ausreicht, sollten Sie diesen verwenden: Er klassifiziert Eingaben zuverlässiger und ist zudem schneller und kostengünstiger.

In manchen Fällen kann es sinnvoll sein, menschliche Beteiligung in Aufgaben zu integrieren. Wenn eine Aufgabe eine teure oder irreversible Aktion erfordert, sollten Sie dafür die Zustimmung eines menschlichen Supervisors einholen. Bei Aufgaben, die eine menschliche Bewertung der Ausgabequalität erfordern, sollten Sie menschliche Prüfer einbinden. Wenn in einem Reflexion-basierten Workflow eine kleine Teilmenge der Aufgaben wiederholt fehlschlägt, sollte ein Mensch die fehlerhafte Aufgabe untersuchen und den Prompt anpassen, um die Aufgabe wieder auf Kurs zu bringen.

Selbst wenn Aufgaben auf LLMs basieren, müssen nicht alle *dasselbe* Modell verwenden: Für einfache Aufgaben sollten Sie ein leichtgewichtiges, kostengünstiges, selbst gehostetes LLM einsetzen. Für komplexe Aufgaben nutzen Sie das jeweils aktuelle leistungsstarke Premium-Modell. Und für sehr spezifische Aufgaben empfiehlt sich ein hausintern feingetuntes Modell.

Evaluierung beginnt auf der Aufgabenebene

Schon bevor der vollständige Workflow ausgearbeitet wird, können einzelne Aufgaben isoliert evaluiert werden. Je komplexer das System, desto größer die Wahrscheinlichkeit für Probleme – und desto aufwendiger ist es, diese aufzuspüren. Workflow-basierte Handlungsfähigkeit bietet einen hilfreichen Rahmen für den Aufbau modularer Systeme, da sich Probleme in der Regel auf eine fehlerhaft arbeitende Aufgabe zurückführen lassen. Durchdenken Sie daher immer genau, wie Ihre Aufgaben ausgeführt werden sollen, welche Fehler auftreten und wie diese abgefangen werden könnten. Im nächsten Kapitel werden

wir uns mit der Evaluation von LLM-Anwendungen beschäftigen und Erkenntnisse gewinnen, die sich auch auf die hier besprochenen Aufgaben und Workflows anwenden lassen.

Zusammenführung des Workflows

Sie haben die anstehende Arbeit in eine endliche Menge von Aufgaben unterteilt, von denen jede ihren Teil des Workflows mit hoher Erfolgsquote erfüllt. Nun ist es an der Zeit, diese Aufgaben zu einem funktionierenden Workflow zusammenzusetzen.

Ein *Workflow* ist vernetztes System von Aufgaben, für das sich verschiedene gedankliche Modelle oder Visualisierungen anbieten. Ein Workflow kann als Zustandsmaschine betrachtet werden, in der jede Aufgabe einen Zustand darstellt. Wenn eine Aufgabe eine Eingabe erhält, wird diese in eine von mehreren möglichen Ausgaben transformiert und dann an nachgelagerte Zustände weitergegeben.

Alternativ kann man Aufgaben auch als Knoten ansehen, die in einem Publish-Subscribe-Muster mit anderen Aufgabenknoten verbunden sind und Arbeitsobjekte basierend auf ihren Abonnements senden und empfangen. Man kann sich Aufgaben aber auch als vollständig von einem Workflow-Orchestrator verwaltet vorstellen, der die Aufgaben überwacht und steuert, wie Arbeitsobjekte zwischen ihnen weitergegeben werden. Im Kern besagen alle diese Konzepte das Gleiche – entscheidend ist die Art und Weise, wie die Aufgaben miteinander verknüpft sind.

Aufgaben können in verschiedenen Topologien verbunden werden. Die einfachste Struktur ist eine *Pipeline* – eine Abfolge von Aufgaben, die sequenziell verbunden sind, sodass die Ausgabe einer Aufgabe an höchstens *eine* weitere Aufgabe übergeben wird. Pipelines eignen sich gut zur schrittweisen Verarbeitung von Informationen. Beispielsweise ließe sich das Shopify-Szenario als Pipeline umsetzen, wie in Abbildung 9.5 dargestellt. Der Vorteil von Pipelines liegt in ihrer Einfachheit – das geht allerdings auf Kosten der Flexibilität. So ist in der Abbildung zu sehen, dass die aus der Website extrahierten Details zwar zur Generierung von Plug-in-Konzepten genutzt werden, diese Informationen aber dem E-Mail-Composer nicht zur Verfügung stehen, obwohl sie dort durchaus nützlich sein könnten. Dieses Problem lässt sich umgehen, indem die extrahierten Details durch den Plug-in-Generator weitergeleitet werden. Dadurch entsteht jedoch zwischen den Aufgaben eine stärkere Kopplung, als es sinnvoll wäre. Denn konkret würde das bedeuten, dass der E-Mail-Composer die Shopdetails vom Plug-in-Generator beziehen müsste – was nicht besonders intuitiv ist.

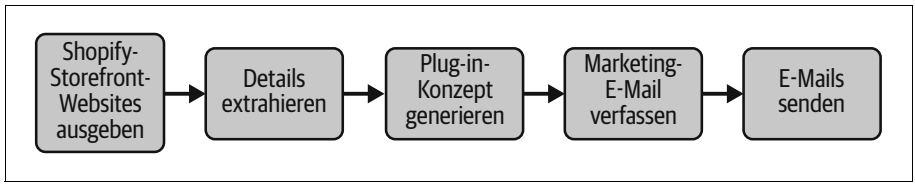

Abbildung 9.5: Eine Pipeline-Implementierung des Shopify-Plug-in-Promoters

Wenn Workflows komplexer werden, kann es vorkommen, dass eine Aufgabe ihre Ausgabe an mehrere nachgelagerte Aufgaben sendet oder Eingaben von mehreren vorgelagerten Aufgaben benötigt. Solange der Arbeitsfluss immer in eine Richtung verläuft (also keine Zyklen entstehen, durch die Informationen zu einer früheren Aufgabe zurückfließen), spricht man von einem *gerichteten azyklischen Graphen* (*Directed Acyclic Graph*, DAG). Das Shopify-Beispiel kann optimiert werden, indem man es als DAG modelliert, bei dem die Shopinformationen gleichzeitig an die Aufgaben zur Konzeptgenerierung und zur E-Mail-Erstellung übermittelt werden (siehe Abbildung 9.6).

DAGs sind essenziell für die Automatisierung von Workflows, da sie viele reale Szenarien abbilden können und dabei dennoch übersichtlich und beherrschbar bleiben. Beliebte Plattformen zur Workflow-Automatisierung wie Airflow (*https://airflow.apache.org*) und Luigi (*https://luigi.readthedocs.io/en/stable*) behandeln Workflows als DAGs, wobei Knoten Aufgaben darstellen und die Verbindungen deren Abhängigkeiten beschreiben. Dadurch lassen sich DAGs leicht nachvollziehen: Eine Aufgabe kann nur ausgeführt werden, wenn alle vorgelagerten Abhängigkeiten erfolgreich abgeschlossen wurden.

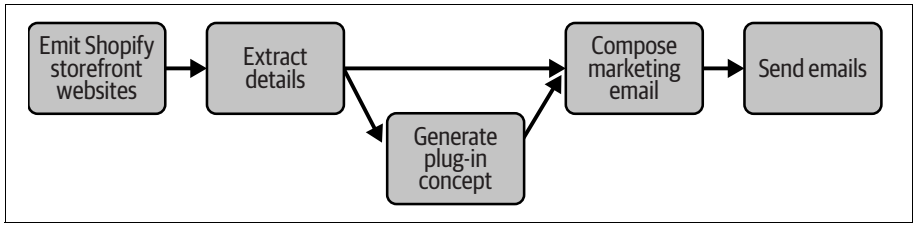

Abbildung 9.6: Eine DAG-Implementierung des Shopify-Plug-in-Promoters

Wie in Abbildung 9.7 dargestellt, ist die allgemeinste Form der Aufgabenanordnung ein *zyklischer Graph* – ein Netzwerk von Aufgaben, in dem die Ausgaben einer Aufgabe zu vorgelagerten Aufgaben zurückfließen und Schleifen entstehen können. Solche zyklischen Strukturen können manchmal sehr nützlich sein. Im Shopify-Workflow ließe sich beispielsweise eine Qualitätskontrolle integrieren: Nur wenn die E-Mails eine ausreichende Qualität aufweisen, werden sie auch an den Shopbetreiber gesendet. Andernfalls wird ein Fehlerbericht

zurück an den Schritt zur Detailextraktion geschickt in der Hoffnung, dass beim nächsten Durchlauf ein besseres Ergebnis erzielt wird.

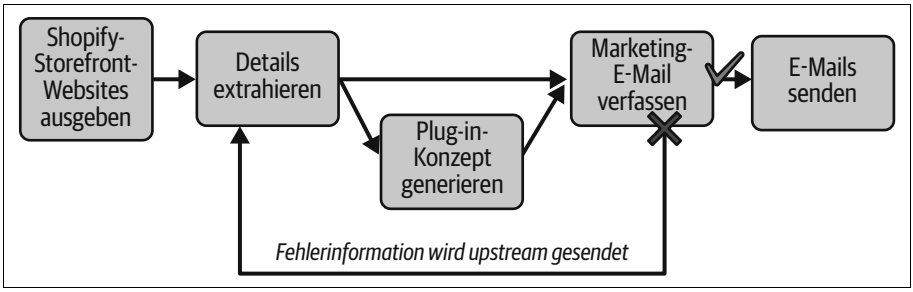

Abbildung 9.7: Eine Implementierung des Shopify-Plug-in-Promoters als zyklischer Graph

In bestimmten Fällen sind zyklische Graphen in LLM-basierten Workflows unvermeidbar – etwa wenn das LLM bei einer bestimmten Aufgabe einen Fehler macht und man das Arbeitselement zurück an eine vorherige Stelle schicken muss, um es eventuell zu korrigieren. Dieses Muster sollte jedoch mit Vorsicht eingesetzt werden, da es die Komplexität stark erhöht. Ein Beispiel dafür findet sich in Abbildung 9.7. Ein Problem besteht darin, dass die Fehlerinformationen beim Zurückleiten zur Extraktionsaufgabe wieder mit den zugehörigen Webseiteninhalten verknüpft werden müssen – während in der DAG-Variante diese Information nie wieder benötigt wird und daher nicht gespeichert werden muss.

Ein weiteres Problem ist, dass jede Aufgabe nun damit rechnen muss, dass ein Arbeitselement mit Fehlerinformationen versehen ist – das muss in den Implementierungen der Aufgaben berücksichtigt werden. Und schließlich stellt sich die Frage: Wie verhindert man, dass Arbeitselemente, bei denen immer wieder Fehler auftreten, endlos durch das System zirkulieren? Dazu ist es nötig, die Anzahl der Versuche zu protokollieren und den Vorgang abzubrechen, sobald die zulässige Anzahl überschritten ist. Wenn Sie überlegen, zyklische Abhängigkeiten in Ihren Workflow zu integrieren, ist es sinnvoll, die Rekursion – wenn möglich – innerhalb einer einzelnen Aufgabe zu kapseln, damit die zusätzliche Komplexität nicht auf die Ebene des Workflows getragen wird, wo dann auch andere Aufgaben mit der zyklischen Abhängigkeit umgehen müssten.

Neben der Aufgabenverknüpfung sollten man auch berücksichtigen, ob der Workflow-Prozess im Batch- oder im Streaming-Modus ablaufen soll. Ein *Batch-Workflow* verarbeitet eine vorab bekannte endliche Menge von Arbeitselementen, während ein *Streaming-Workflow* eine beliebige Anzahl von Arbeitselementen verarbeitet, die während der Ausführung erzeugt oder abgerufen werden. Unser Shopify-Szenario ließe sich auf beide Arten realisieren: als

Batch-Verfahren, bei dem zunächst eine Liste von Shops erstellt und dann abgearbeitet wird, oder im Streaming-Verfahren, bei dem ein Webcrawler kontinuierlich nach Storefronts sucht und diese direkt verarbeitet werden. Beide Ansätze sind hier sinnvoll einsetzbar. Die Batch-Verarbeitung lässt sich in der Regel einfacher einrichten und warten und eignet sich gut für die effiziente Verarbeitung großer Datenmengen, während Streaming besser zu latenzkritischen Aufgaben passt, die in Echtzeit gelöst werden müssen, aber meist komplexer in der Umsetzung ist.

Beispiel-Workflow: Shopify-Plug-in-Marketing

Zu Beginn des Abschnitts »Grundlegende LLM-Workflows« haben wir die Schritte zum Aufbau eines Workflows skizziert. Nun setzen wir diese Schritte praktisch um, indem wir einen vollständigen Workflow für den Shopify-Plug-in-Promoter erstellen. In diesem Szenario versetzen wir uns in die Lage eines kleinen Entwicklerstudios, das sich auf das Shopify-Ökosystem spezialisiert hat. Unser Ziel ist es weiterhin, Shopify-Storefronts zu analysieren, Ideen für Plug-ins zu entwickeln und diese den Shopbetreibern vorzuschlagen – um neue Projekte zu akquirieren.

Ausgehend von unserem einführenden Beispiel haben wir die beteiligten Aufgaben bereits besprochen – nun setzen wir sie um. Natürlich lässt sich eine vollständige Implementierung nicht ohne Weiteres in einem Buch unterbringen, aber wenn Sie es bis hierhin geschafft haben, können Sie sich wahrscheinlich vorstellen, wie diese Aufgaben in der Praxis aussehen würden. Hier ein kurzer Überblick über die Implementierung:

Storefront-HTML bereitstellen
: Das ist eine Mock-Implementierung. Der HTML-Code mehrerer Storefronts wurde manuell erfasst und im Dateisystem gespeichert. Diese Aufgabe gibt ihn einfach aus.

Storefront zusammenfassen
: Hier wird der Text aus dem HTML extrahiert, und ein LLM wird mit einem Prompt aufgefordert, die folgenden zentralen Aspekte der Seite zusammenzufassen:

  1. Was wird verkauft?
  2. Welcher Gesamteindruck entsteht hinsichtlich der Tonalität der Website? Verspielt? Ernst? Entspannend?
  3. Welche Werte stehen im Vordergrund? Nachhaltigkeit? Gesellschaftliches Engagement?
  4. Welche thematischen Schwerpunkte zeigt die Seite? Reisen? Produktivität? Sport?

5. Gibt es etwas, das an der Website besonders lobenswert ist? (Wir bereiten uns hier darauf vor, im E-Mail-Marketing das Ego des Betreibers ein wenig zu streicheln!)
6. Gibt es sonst noch etwas Erwähnenswertes?

Neues Plug-in-Konzept generieren
: Dies ist ein zweistufiger Prozess: Zunächst werden mehrere gute Ideen gesammelt, und die beste wird identifiziert. Anschließend wird ein detaillierter Bericht zur besten Idee erstellt, der ihren Mehrwert für den Kunden beschreibt. Diese Zweiteilung dient dazu, das Chain-of-Thought-Brainstorming vom eigentlichen Plug-in-Konzept zu trennen, denn nur Letzteres wird als Ergebnis beibehalten.

E-Mail generieren
: Auch diese Aufgabe besteht aus mehreren Schritten: Zunächst nutzen wir Chain-of-Thought-Prompting, indem wir das Modell anweisen, eine Strategie zur Vermarktung der Idee zu entwickeln, die zur jeweiligen Storefront passt. Anschließend fragen wir das Modell nach einer Betreffzeile für die E-Mail und schließlich nach dem eigentlichen E-Mail-Text.

E-Mail senden
: Diese Aufgabe ist ebenfalls eine Mock-Implementierung: Die E-Mail wird lediglich auf dem Bildschirm ausgegeben.

Der nächste Schritt besteht darin, die Aufgaben in den vollständigen Workflow zu integrieren, wie in Abbildung 9.8 dargestellt. Im Wesentlichen entspricht dieses Diagramm dem aus Abbildung 9.6 – einem DAG –, jedoch sind in Abbildung 9.8 die spezifischen Ein- und Ausgaben gekennzeichnet.

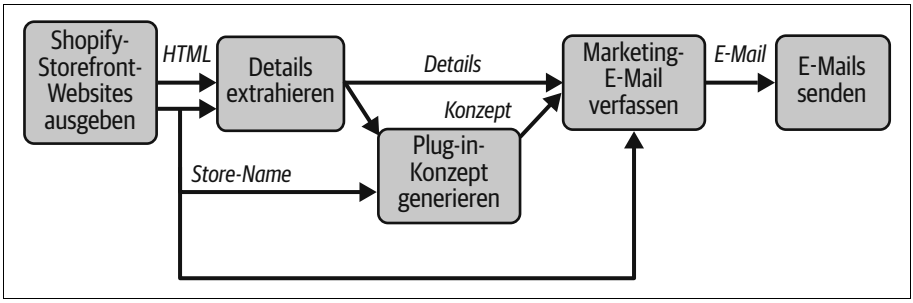

Abbildung 9.8: Finale Umsetzung des Shopify-Plug-in-Promoters

Damit liegt nun ein vollständiger Workflow vor. Zum Test haben wir ihn mit dem HTML-Code eines bekannten Shops für Kochzutaten und Produkte aus der Sichuan-Küche (*https://flybyjing.com*) gefüttert. Und das Ergebnis kann sich für eine körperlose virtuelle Intelligenz, die noch nie die Köstlichkeiten der

Sichuan-Küche erlebt hat, sehen lassen, oder? Wir geben das Ergebnis hier der besseren Verständlichkeit wegen übersetzt wieder.

BETREFF: Mehr Genuss, mehr Inspiration – mit unserem Rezept-Integrations-Plug-in

Liebes Fly-By-Jing-Team,

ich hoffe, es geht Ihnen gut! Ich möchte Ihr beeindruckendes Engagement für Authentizität und handwerkliche Qualität würdigen – Sie bringen die einzigartigen Aromen der Sichuan-Küche zu Feinschmeckern auf der ganzen Welt.

Ich freue mich, Ihnen ein bahnbrechendes Konzept vorzustellen, das die kulinarische Reise Ihrer Kunden revolutionieren wird – unser Rezept-Integrations-Plug-in. Diese innovative Funktion lässt sich nahtlos in Ihren Shopify-Store integrieren und bietet eine kuratierte Sammlung köstlicher Rezepte, die die außergewöhnliche Vielseitigkeit der Fly-By-Jing-Chilisaucen und -Gewürze hervorheben.

Stellen Sie sich eine benutzerfreundliche Oberfläche vor, auf der Kunden eine große Auswahl an Rezepten entdecken können – von verführerischen Vorspeisen bis hin zu exquisiten Desserts, die die einzigartigen Aromen Ihrer Produkte perfekt zur Geltung bringen. Ihr Plug-in bietet nicht nur Inspiration für neue kulinarische Kreationen, sondern ermöglicht Ihren Kunden auch, sich mit einer Community von Food-Enthusiasten auszutauschen und ihre Erfahrungen sowie Rezepte zu teilen.

Neben der Kundenbindung eröffnet dieses Plug-in zahlreiche neue Umsatzpotenziale für Fly By Jing. Durch gezielte Rezeptempfehlungen, exklusive Premium-Inhalte und Kooperationen mit renommierten Köchen und Influencern können Sie Ihre Markenpräsenz weiter ausbauen und Ihre Umsätze nachhaltig steigern.

Gerne stelle ich Ihnen unser Konzept in einem persönlichen Gespräch vor, um dieses innovative Plug-in optimal auf Ihre Markenstrategie abzustimmen und Ihren Kunden ein unvergleichliches kulinarisches Erlebnis zu bieten. Lassen Sie uns gemeinsam eine neue Ära voller Leidenschaft, Kreativität und Geschmack in jeder Küche einläuten.

Beste Grüße

Albert Berryman, Director of Innovation JivePlug-ins Inc.

Der letzte Schritt besteht darin, den Workflow zu optimieren. Das Beispiel, das wir in diesem Kapitel implementiert haben, ist lediglich ein vereinfachtes Modell. Daher sollte zunächst überprüft werden, ob die definierten Aufgaben tatsächlich die *richtigen* sind.

Ein Schwachpunkt dieses Workflows ist die begrenzte Vielfalt der generierten Ideen – es gab viele Vorschläge für virtuelle Anprobe-Plug-ins für Modehändler sowie zahlreiche Impact-Tracker für Stores mit sozialer oder ökologischer Ausrichtung. Eine mögliche Verbesserung wäre es, den Brainstorming-Prozess robuster zu gestalten und gezielt die häufigsten Standardideen zu vermeiden. Dass einige der generierten Konzepte praktisch schwer umsetzbar sind, ist ein weiteres Problem. Hier könnte ein zusätzlicher Prozessschritt helfen, der für jedes Konzept eine grobe Umsetzungsstrategie definiert und überprüft, ob es sich technisch und wirtschaftlich realisieren lässt.

Eine weitere Optimierung wäre die Integration von korrigierendem Feedback in den Workflow. Auf Aufgabenebene könnte dies durch einen Reflexion-Prompt-Flow erfolgen, der die Ergebnisse evaluiert und das Modell gezielt zu Verbesserungen anleitet. Alternativ lässt sich Feedback auf Workflow-Ebene integrieren, indem fehlgeschlagene Arbeitselemente identifiziert und zusammen mit Verbesserungshinweisen an den Anfang des Workflows zurückgeschickt werden.

Sobald die Aufgaben klar definiert sind, sollten Sie Beispieldaten für jede Aufgabe sammeln, um deren Qualität weiter zu optimieren. Bevor eine Aufgabenimplementierung produktiv eingesetzt wird, sollten Sie Offline-Harness-Tests entwickeln, um die Prompts auszuführen und zu überprüfen, ob die Completions dem erwarteten Verhalten entsprechen. So lassen sich Änderungen am Prompt leichter ausrollen, ohne dass die Qualität der Aufgabe darunter leidet. Auch für moderne Optimierungstechniken wie DSPy (*https://arxiv.org/abs/2310.03714*) und TextGrad (*https://arxiv.org/abs/2406.07496*) sind Eingabe/Ausgabe-Beispiele (I/O-Beispiele) nützlich. Diese Frameworks nutzen I/O-Beispiele, um den Prompt so zu optimieren, dass die Qualität – gemessen an einer vorgegebenen Metrik – automatisch verbessert wird.

Sobald eine Aufgabe produktiv eingesetzt wird, ist es wichtig, reale I/O-Daten zu erfassen. Diese können stichprobenartig überprüft werden, um sicherzustellen, dass keine Qualitätsverluste auftreten. Noch wichtiger ist, dass sich diese Daten nutzen lassen, um konkurrierende Implementierungen in A/B-Tests mit Live-Traffic zu evaluieren. Das Thema Evaluierung behandeln wir ausführlich im nächsten Kapitel.

Inhalt

Teil II: Zentrale Techniken

Teil III: Zur Meisterschaft

O'REILLY®

Prompt Engineering für Large Language Models

Large Language Models (LLMs) revolutionieren die Welt. Sie bieten die Möglichkeit, Aufgaben zu automatisieren und komplexe Probleme zu lösen. Eine neue Generation von Softwareanwendungen nutzt diese Modelle als zentrale Bausteine und erschließt in praktisch allen Bereichen völlig neues Potenzial. Um zuverlässig auf diese Funktionen zugreifen zu können, sind jedoch neue Fähigkeiten erforderlich. Mit diesem Buch eignen Sie sich die Kunst und die Techniken des Prompt Engineering an und schöpfen die Möglichkeiten von LLMs voll aus.

Die Branchenexperten John Berryman und Albert Ziegler erklären, wie man effektiv mit KI kommuniziert und Ideen in ein sprachmodellfreundliches Format umwandelt. Durch das Erlernen der konzeptionellen Grundlagen und der praktischen Techniken erwerben Sie das Wissen und die nötige Sicherheit, um die nächste Generation von LLM-basierten Anwendungen zu entwickeln.

Aus dem Inhalt:

- **Verstehen Sie die LLM-Architektur und erfahren Sie, wie Sie am besten mit ihr interagieren können.**
- **Entwerfen Sie eine umfassende Strategie zur Erstellung von Prompts für eine Anwendung.**
- **Sammeln, sortieren und präsentieren Sie Kontextelemente, um eine effiziente Eingabeaufforderung zu erstellen.**
- **Erlernen Sie spezifische Techniken, um Prompts zu erstellen, wie Few-Shot-Prompting, Chain-of-Thought-Prompting und RAG.**

»Albert und John stehen hinter einem der erfolgreichsten kommerziellen Projekte der Geschichte – Copilot –, einem generativen KI-Produkt auf GitHub. Sie sind großartige Vorbilder, von denen man viel lernen kann. Ihre Texte machen das Thema Prompt Engineering für jeden zugänglich.«

— Hamel Husain
Unabhängiger KI-Forscher und Berater

John Berryman ist Gründer von Arcturus Labs, einem Unternehmen, das sich auf die Entwicklung von LLM-basierten Anwendungen spezialisiert hat. Er war einer der ersten Entwickler bei GitHub Copilot und arbeitete dort an Chat- und Code-Vervollständigungsfunktionen. John ist außerdem Suchmaschinenexperte und Autor des Fachbuchs »Relevant Search« (Manning).

Albert Ziegler ist Head of AI beim KI-Cybersicherheitsunternehmen XBOW. Als einer der Gründungsentwickler von GitHub Copilot, dem ersten erfolgreichen LLM-Produkt im industriellen Maßstab, entwarf er dessen Systeme für die Modellinteraktion und das Prompt Engineering.

www.dpunkt.de

Euro 39,90 (D)
ISBN 978-3-96009-270-4

9 783960 092704

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier