



Anwendung Generativer KI

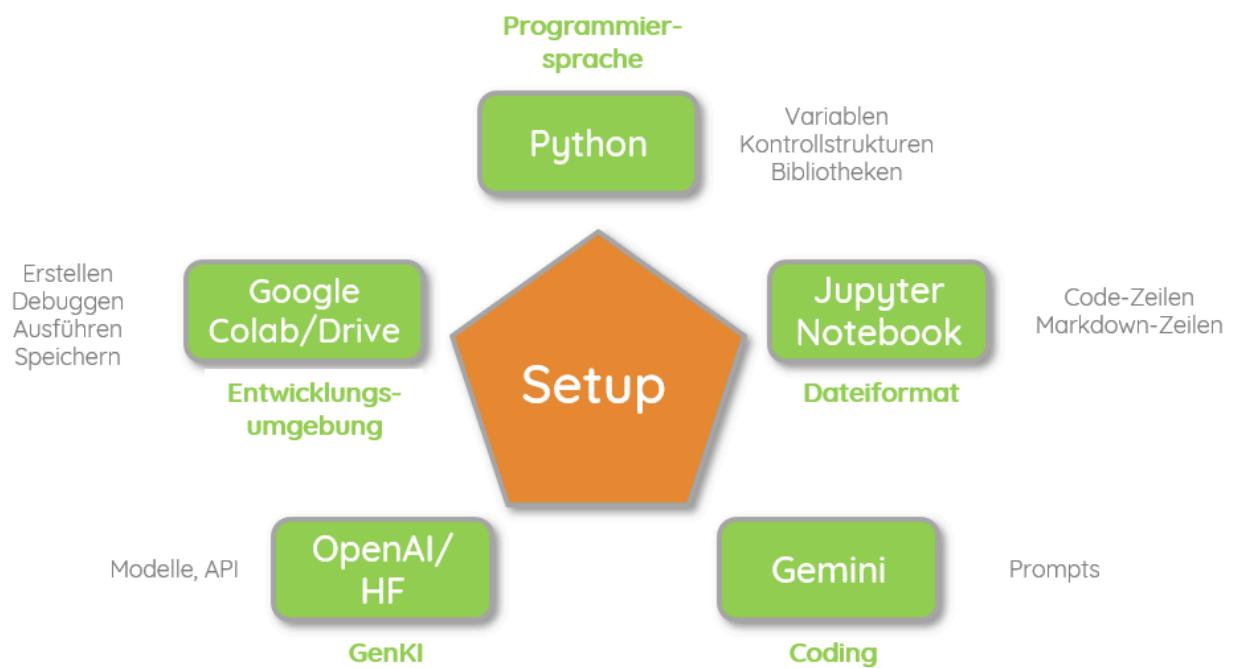
September 2025

Lizenz

- Die Kursmaterialien wurden von Ralf Bendig erstellt, sofern nicht anders angegeben.
- Lizenz CC BY 4.0.

Titelseite: Bild mit MS Copilot erstellt

IT-Setup



Pinboard

https://bit.ly/3T2evF6

Google Colab(oratory)



- Google Colaboratory, kurz Colab, ist eine kostenlose Entwicklungsumgebung, die vollständig in der Cloud ausgeführt wird.
- In Colab können Jupyter-Notebooks erstellt, bearbeiten und ausgeführt werden.
- Colab unterstützt viele beliebte Machine-Learning-Bibliotheken, die einfach in ein Notebook geladen werden können.
- Colab erlaubt es unterschiedliche Laufzeitumgebungen zu definieren in denen man neben einer CPU auch GPUs und TPUs verwenden kann.
- Colab hat mit Gemini eine integriert GenKI für Coding.

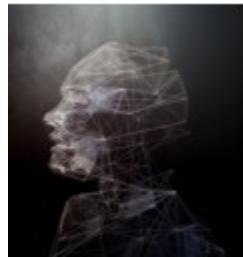
CPU = Central Processing Unit, GPU = Graphics Processing Unit, TPU = Tensor Processing Unit

Jupyter-Notebook



- Die Jupyter App ist ein Entwicklungsumgebung, die das Bearbeiten und Ausführen von Programmiersprachen, u.a. Python, über einen Webbrowser ermöglicht.
- Der Name Jupyter bezieht sich auf die drei wesentlichen Programmiersprachen Julia, Python und R und ist auch eine Hommage an Galileos Notizbucheinträge zur Entdeckung der Jupitermonde.
- Mit der Jupyter App kann man Notizbücher erstellen. Diese Notizbücher (Dateiendung .ipynb) enthalten:
 - **Programmcode**, der ausgeführt werden kann,
 - **Markdown Zeilen**, das sind Textzeilen mit Formatierungsangaben.
- Die Jupyter-Notebooks werden v.a. für interaktive, wissenschaftliche Analysen und Berechnungen, z.B. Data Analytics und Machine Learning, verwendet.

M00 - Kursplan GenAI



Anwendung Generativer KI

Stand: 05.2025

1 Basismodule (Module 1-12)

Die Basismodule bilden das Fundament des Kurses und vermitteln grundlegende Konzepte und Werkzeuge der generativen KI:

- **Einführende Grundlagen:** Allgemeine Einführung in generative KI, Modellansteuerung und fundamentale Frameworks (Module 1-4)
- **Technische Grundlagen:** Vertiefung in Transformer-Architektur, Memory-Konzepte und Output-Parser (Module 5-7)
- **Praktische Anwendungen:** Einführung in RAG, multimodale Bildverarbeitung, Agenten, Gradio und lokale Modelle (Module 8-12)

Diese Module stellen sicher, dass Sie über ein solides Grundverständnis der generativen KI-Technologien verfügen.

2 Erweiterungsmodule (Module 13-20)

Die Erweiterungsmodule bauen auf den Grundlagen auf und bieten fortgeschrittene Konzepte und Spezialisierungen:

- **Erweiterte multimodale Anwendungen:** SQL RAG, Audio- und Videoverarbeitung (Module 13-15)
- **Modelloptimierung:** Fine-Tuning, Modellauswahl und Evaluation (Module 16-17)

- **Fortgeschrittene Methoden:** Advanced Prompt Engineering (Modul 18)
- **Regulatorische Aspekte:** EU AI Act und Ethik (Modul 19)
- **KI-Challenge:** Praktische Anwendung und Integration der Kursmodule (Modul 20)

Diese Module vertiefen spezifische Anwendungsbereiche und bieten fortgeschrittene Techniken für professionelle KI-Anwendungen.

Die Progression von Basis zu Erweiterung folgt einem logischen Lernpfad: Zuerst erlernen Sie die Grundprinzipien und -werkzeuge, bevor Sie sich mit spezialisierteren und komplexeren Themen befassen.

Die Basismodule sind obligatorisch, die Erweiterungsmodule sind fakultativ.

3 Modulübersicht nach Basis/Erweiterung

Modul-Nr.	Modultyp	NB	PDF	Inhalt	Themen	Relevanz
1	Basis	✓	✓	Einführung GenAI	<ul style="list-style-type: none"> • Kursüberblick • Überblick Generative AI • Einführung OpenAI • Einführung Hugging Face • Einführung LangChain 	Fundamentales Verständnis der Gen-AI-Technologien als Grundlage für den Kurs.
2	Basis	✓	✓	Grundlagen Modellansteuerung	<ul style="list-style-type: none"> • Überblick Prompting, RAG, Fine-Tuning • Einsatzszenarien • Entscheidungskriterien • Trade-offs 	Essentielles Verständnis der verschiedenen Ansätze zur Modellansteuerung.

Modul-Nr.	Modultyp	NB	PDF	Inhalt	Themen	Relevanz
3	Basis	✓	✓	Codieren mit GenAI	<ul style="list-style-type: none"> Prompting für Codegenerierung Revisionsprompts Debugging mit LLMs Prozessintegration Grenzen der LLM-Codegenerierung 	Praktische Fähigkeiten für LLM-gestützte Programm-Entwicklung.
4	Basis	✓	✓	LangChain 101	<ul style="list-style-type: none"> Was ist LangChain & Architektur Kernkonzepte (Chains, Models, Prompts) Best Practices & Design Patterns Weiterführende Ressourcen & Community 	Fundamentales Verständnis von LangChain als zentrales Framework für die Entwicklung von LLM-Anwendungen.
5	Basis	✓	✓	Large Language Models und Transformer	<ul style="list-style-type: none"> Foundation Model Transformer-Architektur Textgenerierung Textzusammenfassung Textklassifizierung LLM schreibt ein Buch 	Vertieftes Verständnis der Transformer-Architektur und LLM-Funktionsweise.
6	Basis	✓		Chat und Memory	<ul style="list-style-type: none"> LangChain-Konversationen Conversation Buffer Window Memory Conversation Token Buffer Memory Conversation Summary Memory Persisting Memory 	Memory-Typen für Flexibilität und Skalierbarkeit, um Konversationen effizient zu verwalten.

Modul-Nr.	Modultyp	NB	PDF	Inhalt	Themen	Relevanz
7	Basis	✓		Output Parser	<ul style="list-style-type: none"> • Structured Output Parser • CSV/JSON • Pydantic Parser • Custom Output Parser 	Parser-Typen für robuste und flexible Verarbeitung von Modellantworten.
8	Basis	✓	✓	Retrieval Augmented Generation	<ul style="list-style-type: none"> • Einführung in RAG • ChromaDB • Embeddings • Q&A über Dokumente • Embedding-Datenbanken 	Schlüsseltechnologie für Informationsverarbeitung mit unstrukturierten Daten.
9	Basis	✓	✓	Multimodal Bild	<ul style="list-style-type: none"> • Bildgenerierung • In-/Outpainting • Bildklassifizierung • Objekterkennung • Bildbeschreibung 	Erweiterung um visuelle Komponenten.
10	Basis	✓		Agents	<ul style="list-style-type: none"> • Grundlagen von KI-Agenten • Agentenarchitekturen • Planung und Zielverfolgung • Multi-Agenten-systeme 	Entwicklung autonomer KI-Systeme.
11	Basis	✓		Gradio	<ul style="list-style-type: none"> • Grundkonzepte von Gradio • Installation und Setup • Zentrale Komponenten • Praktische Beispiele • Fortgeschrittene Konzepte • Integration mit KI-Modellen • Best Practices • Deployment und Sharing 	UI-Entwicklung für KI-Anwendungen.

Modul-Nr.	Modultyp	NB	PDF	Inhalt	Themen	Relevanz
12	Basis	✓		Lokale und Open Source Modelle	<ul style="list-style-type: none"> • Einführung • Ollama • Lokale Modelle und LangChain • Open Source vs. Closed Source • Beispiele Open Source Modelle • Lizenzierung und rechtliche Aspekte • Auswahlkriterien Open Source • Zukunftstrends bei Open Source 	Einsatz von KI-Modellen ohne Cloud-Abhängigkeit.
13	Erweiterung	✓		SQL RAG	<ul style="list-style-type: none"> • Einführung in SAG • Vergleich zu RAG • Integration von LLMs mit Datenbanken • SQL-Generierung mit LLMs • Datenvalidierung und Sicherheitsaspekte • Praktische Anwendungsfälle • SAG mit LangChain 	Schlüsseltechnologie für die Arbeit mit strukturierten Daten und Datenbanken.

Modul-Nr.	Modultyp	NB	PDF	Inhalt	Themen	Relevanz
14	Erweiterung	✓	✓	Multimodal Audio	<ul style="list-style-type: none"> • Speech-to-Text (STT) • Text-to-Speech (TTS) • Sprachanalyse • Audio-Summary • Audio-Pipeline • Podcast 	Erschließung von Sprachanwendungen.
15	Erweiterung	✓		Multimodal Video	<ul style="list-style-type: none"> • Video-zu-Text (VTT) • Text-zu-Video (TTV) • Bild-zu-Video (ITV) • Videoanalyse • Video-Objekterkennung 	Integration von Videoverarbeitung und -analyse für multimodale KI-Anwendungen.
16	Erweiterung	✓	✓	Fine-Tuning	<ul style="list-style-type: none"> • Fine-Tuning mit Dashboard • Fine-Tuning mit Code • Auswerten des FT-Modells 	Optimierung und effiziente Anpassung für spezifische Anwendungsfälle.
17	Erweiterung		✓	Modellauswahl und Evaluation	<ul style="list-style-type: none"> • KI-Modelle • Bewertung von Modellen • Modellauswahlprozess • Auswahlkriterien • Benchmarks und Modellvergleiche 	Fundierte Modellauswahl und Bewertung.
18	Erweiterung		✓	Advanced Prompt Engineering	<ul style="list-style-type: none"> • Einführung Prompt-Engineering • Few-Shot und Chain-of-Thought • Persona- und Rollenmuster • Frage- und Prüfmuster • Inhalt- und Struktur-Muster • Chat- und Reasoning-Modelle 	Fortgeschrittene Prompt-Strategien.

Modul-Nr.	Modultyp	NB	PDF	Inhalt	Themen	Relevanz
19	Erweiterung		✓	EU AI Act /Ethik	<ul style="list-style-type: none"> • Risikobasierte Einstufung • Hochrisiko-Anwendungen • Strenge Auflagen für Hochrisiko-KI • Transparenz- und Informationspflichten • Ethische Grundsätze 	Der EU AI Act ist seit 2025 das zentrale Regelwerk für den vertrauenswürdigen, sicheren und ethischen Einsatz von KI in Europa.
20	Erweiterung		✓	KI-Challenge	<ul style="list-style-type: none"> • Projektoptionen • Integration mehrerer Technologien • Entwicklung einer End-to-End-Anwendung • Graido-Benutzeroberfläche • Evaluation und Dokumentation 	Anwendung und Integration der erlernten Konzepte in einer Gesamtlösung.

4 Modulübersicht nach Themencluster

1. Grundlagen und Frameworks

- Einführung GenAI (Modul 1)
- Grundlagen Modellansteuerung (Modul 2)
- LangChain 101 (Modul 4)
- Large Language Models und Transformer (Modul 5)

2. Praktische Anwendungsentwicklung

- Codieren mit GenAI (Modul 3)
- Chat und Memory (Modul 6)

- Output Parser (Modul 7)
- Gradio (Modul 11)
- KI-Challenge (Modul 20)

3. Datenintegration und Retrieval

- Retrieval Augmented Generation (Modul 8)
- Lokale und Open Source Modelle (Modul 12)
- SQL RAG (Modul 13)

4. Multimodale Anwendungen

- Multimodal Bild (Modul 9)
- Multimodal Audio (Modul 14)
- Multimodal Video (Modul 15)

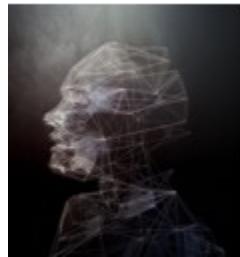
5. Fortgeschrittene Techniken

- Agents (Modul 10)
- Fine-Tuning (Modul 16)
- Advanced Prompt Engineering (Modul 18)

6. Modellbewertung und Regulierung

- Modellauswahl und Evaluation (Modul 17)
- EU AI Act/Ethik (Modul 19)

M04 - OpenAI vs LangChain



Anwendung Generativer KI

Stand 05.2025

1 Einleitung

Die Entwicklung von Applikationen, die auf großen Sprachmodellen (Large Language Models, LLMs) basieren, hat in den letzten Jahren rasant an Fahrt aufgenommen. Für Entwickler stellt sich dabei oft die zentrale Frage nach den richtigen Werkzeugen und Frameworks. Zwei prominente Optionen im Jahr 2025 sind die direkte Nutzung der Programmierschnittstelle (API) von OpenAI und der Einsatz des LangChain-Frameworks, typischerweise mit einem OpenAI-Modell als Backend. Angesichts der Schnelllebigkeit in diesem Technologiefeld ist ein aktueller Vergleich unerlässlich, um fundierte Entscheidungen treffen zu können.

Dieser Bericht stellt die aktuellen Funktionalitäten der direkten OpenAI API und des LangChain-Frameworks (bei Verwendung mit OpenAI-Modellen) gegenüber. Der Fokus liegt auf den Kernaspekten Prompting-Mechanismen, Output-Parsing und dem Aufruf von Sprachmodellen. Ergänzend werden die kritischen Themen Tool/Function Calling sowie Fehlerbehandlung und Debugging beleuchtet, da sie für die praktische Entwicklung von LLM-Applikationen von entscheidender Bedeutung sind. Eine wichtige Neuerung seitens OpenAI, die **Responses API 1**, wird ebenfalls in die Betrachtung einbezogen, da sie potenziell signifikante Auswirkungen auf den Vergleich und die Art und Weise hat, wie Entwickler mit den Modellen interagieren.

2 Überblick

Die folgende Tabelle bietet eine erste, hochrangige Zusammenfassung der Kernunterschiede zwischen der direkten Nutzung der OpenAI API und der Verwendung von LangChain mit einem OpenAI-Backend. Sie dient als Referenzpunkt für die nachfolgenden detaillierten Abschnitte.

Aspekt	OpenAI API (Direkt)	LangChain (mit OpenAI)
Prompting-Mechanismen	messages -Array (Chat Completions API) 2; input & instructions (Responses API).3 Manuelle Strukturierung.	PromptTemplate , ChatPromptTemplate .5 Flexible Template-Erstellung und -Verwaltung. LCEL für Verkettung.7
Output-Parsing	Manuelles Parsen von Text; tool_calls für strukturiertes JSON.8	Dedizierte OutputParser (z.B. StrOutputParser , JsonOutputParser , PydanticOutputParser).5 Automatische Strukturierung und Validierung.
Aufruf von Sprachmodellen	client.chat.completions.create() 2, client.responses.create().1 Direkter API-Zugriff. Streaming via stream=True .	ChatOpenAI().invoke() , ChatOpenAI().stream() .5 Abstrahierte, konsistente Schnittstelle.
Tool/Function Calling	tools -Parameter mit JSON-Schema-Definition.8 tool_choice zur Steuerung. Responses API mit Built-in Tools.1	bind_tools mit Pydantic-Modellen oder Funktionen.5 Orchestrierung innerhalb von Agents.
Abstraktionsebene	Gering. Direkte Kontrolle über API-Parameter.	Hoch. Vereinfachung durch Abstraktionen und standardisierte Komponenten.
Flexibilität (Modellagnostik)	Bindung an OpenAI-Modelle.	Hoch. Prinzipiell einfacher Wechsel des LLM-Providers möglich.13
Fehlerbehandlung & Debugging	Standard API-Fehlercodes und Exceptions.15 Manuelle Implementierung von Retries.	Erweiterte Parser-Retries (RetryOutputParser).16 Observability-Plattform LangSmith für Tracing und Debugging.5

Zustandsmanagement (Conversation State)	Manuell mit Chat Completions API (gesamter Verlauf muss gesendet werden).1 Serverseitig mit Responses API (<code>store: true</code>).1	LangChain Memory-Module für verschiedene Speicherstrategien.5
Komplexität der Implementierung	Geeignet für einfache, direkte Aufgaben. Potenziell geringere Einstiegshürde für Basisfunktionen.19	Geeignet für komplexe, mehrstufige Workflows (Chains, Agents).13 Kann bei einfachen Aufgaben Overhead erzeugen.

Diese Tabelle verdeutlicht, dass die direkte OpenAI API maximale Kontrolle und Direktheit bietet, während LangChain durch Abstraktion, Modularität und spezialisierte Werkzeuge die Entwicklung komplexerer Anwendungen vereinfachen kann.

3 Prompting-Mechanismen

Die Art und Weise, wie Anweisungen (Prompts) an ein Sprachmodell formuliert und übergeben werden, ist fundamental für dessen Verhalten und die Qualität der generierten Antworten.

3.1 OpenAI API (Direkt)

Bei der direkten Nutzung der OpenAI API erfolgt das Prompting für Chat-Modelle primär über den `messages`-Parameter. Dieser ist ein Array von Dictionaries, wobei jedes Dictionary eine einzelne Nachricht in der Konversation repräsentiert und die Schlüssel `role` und `content` enthält.2

Die möglichen Rollen sind:

- `system` : Definiert das übergeordnete Verhalten, die Persönlichkeit oder spezifische Anweisungen für das Modell. Diese Nachricht wird typischerweise am Anfang des `messages` -Arrays platziert.2
- `user` : Repräsentiert die Eingaben des Endnutzers oder der Applikation.
- `assistant` : Stellt die vorherigen Antworten des Modells dar.

Für die **Chat Completions API** ist es notwendig, den gesamten bisherigen Konversationsverlauf bei jedem API-Aufruf mitzusenden, damit das Modell den Kontext beibehält.1 Dies kann bei sehr langen Konversationen zu einer erheblichen Menge an übertragenen Daten führen.

Eine neuere Entwicklung ist die **OpenAI Responses API** (Stand Q1/2025), die alternative Mechanismen bietet.¹ Hier kann der `instructions`-Parameter genutzt werden, um eine Systemnachricht zu übergeben. Dies ist besonders relevant in Kombination mit `previous_response_id`, da die Instruktionen aus einer vorherigen Antwort nicht automatisch auf die nächste übertragen werden, was das flexible Austauschen von Systemanweisungen vereinfacht.³ Zudem ermöglicht die Responses API mit dem Parameter `store: true` ein serverseitiges Management des Konversationszustands, wodurch das wiederholte Senden des gesamten Verlaufs entfällt.¹

Codebeispiel (Python, 2025-kompatibel): OpenAI API Chat Completion

```
from openai import OpenAI

# Es wird davon ausgegangen, dass der OPENAI_API_KEY als Umgebungsvariable gesetzt ist.
client = OpenAI()

try:
    chat_completion = client.chat.completions.create(
        model="gpt-4o", # Stand 2025 ein gängiges, leistungsfähiges Modell
        messages=[
            {"role": "system", "content": "Du bist ein hilfreicher Assistent, der prägnante Antworten gibt."},
            {"role": "user", "content": "Was ist die Hauptstadt von Frankreich?"}
        ]
    )
    if chat_completion.choices:
        assistant_message = chat_completion.choices.message.content
        print(f"OpenAI API Assistent: {assistant_message}")
    else:
        print("Keine Antwort von der API erhalten.")

except Exception as e:
    print(f"Ein Fehler ist aufgetreten: {e}")

# Beispielhafter Aufruf mit der neueren Responses API (Konzept, API kann variieren)
# try:
#     response_stream = client.responses.create(
```

```
#     model="gpt-4.1", # Beispielmodell für Responses API
#     input=[
#         {"role": "user", "content": "Erzähle einen Witz über Programmieren."}
#     ],
#     instructions="Antworte humorvoll und kurz.",
#     store=True # Aktiviert serverseitiges Zustandsmanagement
# )
# # Verarbeitung des Streams oder der finalen Antwort hier
# # for event in response_stream:
# #     if event.type == 'response.output_text.delta':
# #         print(event.data.delta, end="")
# #     print("\n(Beispiel für Responses API)")
# except Exception as e:
#     print(f"Ein Fehler mit der Responses API ist aufgetreten: {e}")
```

Code basierend auf 3

3.2 LangChain (mit OpenAI)

LangChain abstrahiert die direkte API-Interaktion und bietet für das Prompting mächtige Werkzeuge in Form von `PromptTemplate` und `ChatPromptTemplate`.⁵

- `PromptTemplate` wird für einfache String-basierte Prompts verwendet.
- `ChatPromptTemplate` ist für Chat-Modelle konzipiert und erlaubt die Definition einer Sequenz von Nachrichten, ähnlich dem `messages`-Array der OpenAI API, jedoch mit der Möglichkeit, Platzhalter für dynamische Inhalte zu verwenden.⁶

LangChain unterstützt verschiedene Nachrichtentypen innerhalb eines `ChatPromptTemplate`, darunter `SystemMessage`, `HumanMessage`, `AIMessage` und das flexiblere `ChatMessagePromptTemplate`, das beliebige Rollen erlaubt.⁶ Ein besonders nützliches Element ist `MessagesPlaceholder`. Dieser Platzhalter ermöglicht es, eine dynamische Liste von Nachrichten – beispielsweise eine Chat-Historie aus einem Memory-Modul – an einer bestimmten Stelle in das Template einzufügen.⁶

Die erstellten Prompt-Templates werden typischerweise mithilfe der LangChain Expression Language (LCEL) durch den | (Pipe)-Operator mit einem LLM-Modell und optional einem Output-Parser verkettet.⁵

Codebeispiel (Python, 2025-kompatibel): LangChain ChatPromptTemplate

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.messages import SystemMessage, HumanMessage

# Es wird davon ausgegangen, dass der OPENAI_API_KEY als Umgebungsvariable gesetzt ist.
llm = ChatOpenAI(model="gpt-4o", temperature=0) # Stand 2025

# Definition eines ChatPromptTemplate
prompt_template = ChatPromptTemplate.from_messages()

# Alternative Definition mit Tupeln (oft kürzer)
# prompt_template_alt = ChatPromptTemplate.from_messages()

# Verkettung mit LCEL
chain = prompt_template | llm | StrOutputParser()

try:
    # Aufruf der Kette mit dynamischen Werten für die Platzhalter
    response = chain.invoke({"land": "Italien"})
    print(f"LangChain Assistent: {response}")
except Exception as e:
    print(f"Ein Fehler mit LangChain ist aufgetreten: {e}")
```

Code basierend auf 6

3.3 Vergleichende Analyse

Die direkte OpenAI API bietet granulare Kontrolle über die an das Modell gesendeten Nachrichten. Dies erfordert jedoch ein manuelles Management der Nachrichtenliste und des Konversationskontextes, es sei denn, die neuere Responses API mit serverseitigem State-Management wird genutzt.¹

LangChain hingegen führt eine Abstraktionsebene durch Prompt-Templates ein. Diese erhöhen die Wiederverwendbarkeit und Lesbarkeit von Code, insbesondere bei komplexen oder sich häufig ändernden Prompt-Strukturen. Die LCEL ermöglicht eine elegante Verkettung von Komponenten.

Die Wahl des Ansatzes hängt stark von den Projektanforderungen ab. Für einfache Anwendungen mit statischen Prompts mag die direkte API-Nutzung ausreichend sein. Sobald jedoch dynamische Prompt-Generierung, komplexe Kontextualisierung oder die Notwendigkeit zur einfachen Integration von Chat-Historien (über `MessagesPlaceholder`) ins Spiel kommen, bieten die LangChain-Mechanismen deutliche Vorteile in Bezug auf Organisation und Wartbarkeit.

Ein wichtiger Aspekt ist die Entwicklung bei OpenAI selbst. Mit der Einführung der Responses API und Features wie `store: true` für das Zustandsmanagement¹ nähert sich OpenAI Funktionalitäten an, die traditionell Stärken von Frameworks wie LangChain waren. Für Anwendungen, die primär auf OpenAI-Modelle setzen und grundlegendes Konversationsmanagement benötigen, könnte die direkte Nutzung der Responses API eine schlankere Alternative darstellen. LangChain behält seine Stärken bei sehr komplexen Prompt-Manipulationen, der Notwendigkeit von Modellagnostik oder wenn umfangreiche Logik vor dem eigentlichen LLM-Aufruf ausgeführt werden muss. Die Flexibilität von LangChain durch Templates und Platzhalter kann jedoch für Einsteiger eine höhere initiale Lernkurve bedeuten als das direkte Arbeiten mit dem `messages`-Array der OpenAI API.²⁰

4 Output-Parsing

Nachdem das LLM eine Antwort generiert hat, ist der nächste entscheidende Schritt das Parsen dieses Outputs, insbesondere wenn strukturierte Daten erwartet werden.

4.1 OpenAI API (Direkt)

Standardmäßig liefert die OpenAI API die Antworten der LLMs als reinen Text. Wenn die Anwendung strukturierte Daten, beispielsweise im JSON-Format, benötigt, muss dieser Text manuell geparsst und validiert werden.

Der empfohlene und robustere Weg, um strukturierte Daten von der API zu erhalten, ist die Nutzung von **Tool Calls** (früher Function Calling).⁸ Dabei wird dem Modell im API-Aufruf eine Liste von verfügbaren "Tools" (Funktionen) mitsamt einer Beschreibung und einem JSON-Schema für deren erwartete Parameter übergeben. Das Modell kann dann entscheiden, ein solches Tool "aufzurufen", indem es ein JSON-Objekt zurückgibt, das den Namen des Tools und die zugehörigen Argumente enthält. Dieses JSON-Objekt ist dann bereits strukturiert und kann direkt weiterverarbeitet werden. Die `finish_reason` in der API-Antwort (z.B. `tool_calls`) signalisiert, dass das Modell ein Tool verwenden möchte.²

Die neue **Responses API** ist explizit darauf ausgelegt, Workflows mit Tool-Nutzung zu vereinfachen.¹ OpenAI bietet auch einen "strict mode" für Tool Calls, der die Einhaltung des definierten Schemas erzwingen soll, allerdings mit einigen Einschränkungen verbunden ist.⁸

Codebeispiel (Python, 2025-kompatibel): OpenAI API Output-Parsing mit Tool Call

```
import json
from openai import OpenAI

client = OpenAI()

# Definition des Tools, das das Modell "aufrufen" kann
tools =
    []
]

messages = [
    {"role": "user", "content": "Max Mustermann ist 30 Jahre alt."}
]

try:
    response = client.chat.completions.create(
```

```
model="gpt-4o",
messages=messages,
tools=tools,
tool_choice="auto" # Lässt das Modell entscheiden, ob ein Tool genutzt wird
)

response_message = response.choices.message
tool_calls = response_message.tool_calls

if tool_calls:
    for tool_call in tool_calls:
        if tool_call.name == "extract_user_info":
            function_args = json.loads(tool_call.arguments)
            name = function_args.get("name")
            age = function_args.get("age")
            print(f"OpenAI API (Tool Call) - Name: {name}, Alter: {age}")
else:
    # Fallback, falls kein Tool Call erfolgte, sondern eine direkte Antwort
    print(f"OpenAI API (Direkte Antwort): {response_message.content}")

except Exception as e:
    print(f"Ein Fehler ist aufgetreten: {e}")
```

Code basierend auf 8

4.2 LangChain (mit OpenAI)

LangChain stellt eine Vielzahl von `OutputParser`-Klassen zur Verfügung, um die Antworten von LLMs zu verarbeiten und in gewünschte Formate zu überführen.⁵ Zu den wichtigsten gehören:

- `StrOutputParser`: Gibt die LLM-Antwort als einfachen String zurück. Dies ist die grundlegendste Form des Parsings.¹²
- `JsonOutputParser`: Versucht, die LLM-Antwort als JSON-Objekt zu parsen. Optional kann ein Pydantic-Modell übergeben werden, um das Schema zu definieren und die geparseten Daten zu validieren.¹⁰

- **PydanticOutputParser** : Parst die LLM-Antwort direkt in eine Instanz eines vordefinierten Pydantic-Modells. Dies bietet starke Typsicherheit und eine klare Datenstruktur.¹⁰

Viele Parser in LangChain bieten eine Methode `get_format_instructions()`. Diese generiert eine textuelle Anweisung, die dem Prompt hinzugefügt werden kann, um das LLM anzuleiten, seine Ausgabe im korrekten, vom Parser erwarteten Format zu generieren.¹⁰ Output-Parser werden typischerweise als letztes Glied in einer LCEL-Kette verwendet: `prompt | model | parser`.¹⁰ LangChain unterstützt auch das Streaming von strukturierten Outputs, was bedeutet, dass Teile der strukturierten Antwort bereits verarbeitet werden können, während das Modell noch generiert.¹⁰

Codebeispiel (Python, 2025-kompatibel): LangChain mit PydanticOutputParser

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import PromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field # Wichtig: langchain_core.pydantic_v1 für Kompatibilität
from langchain_core.output_parsers import PydanticOutputParser

# Es wird davon ausgegangen, dass der OPENAI_API_KEY als Umgebungsvariable gesetzt ist.
llm = ChatOpenAI(model="gpt-4o", temperature=0)

# Definition des Pydantic-Modells für die strukturierte Ausgabe
class UserInfo(BaseModel):
    name: str = Field(description="Der vollständige Name der Person")
    age: int = Field(description="Das Alter der Person in Jahren")
    city: str = Field(description="Die Stadt, in der die Person wohnt")

# Initialisierung des Parsers mit dem Pydantic-Modell
pydantic_parser = PydanticOutputParser(pydantic_object=UserInfo)

# Erstellung des Prompt-Templates mit Formatierungsanweisungen vom Parser
prompt = PromptTemplate(
    template="Extrahiere die Informationen aus der folgenden Nutzereingabe.\n{n{format_instructions}}\n{n{query}}\n",
    input_variables=["query"],
```

```
partial_variables={"format_instructions": pydantic_parser.get_format_instructions()}

# Verkettung mit LCEL
chain = prompt | llm | pydantic_parser

try:
    user_query = "Anna Schmidt ist 28 Jahre alt und lebt in Berlin."
    parsed_output = chain.invoke({"query": user_query})
    print(f"LangChain (PydanticOutputParser) - Name: {parsed_output.name}, Alter: {parsed_output.age}, Stadt: {parsed_output.city}")
except Exception as e:
    print(f"Ein Fehler mit LangChain ist aufgetreten: {e}")
```

Code basierend auf 31

4.3 Vergleichende Analyse

Die direkte Nutzung der OpenAI API erfordert für strukturierte Daten entweder manuelles Parsen von Text-Antworten oder die Implementierung des Tool Calling Mechanismus. Letzterer ist der robustere Ansatz, da das Modell angeleitet wird, direkt JSON-konforme Argumente zu liefern.

LangChain bietet mit seinen `OutputParser`-Klassen eine höhere Abstraktionsebene und spezialisierte Werkzeuge, die das Parsen, Strukturieren und Validieren von LLM-Antworten erheblich vereinfachen. Insbesondere die Kombination mit Pydantic-Modellen ermöglicht eine typsichere und gut definierte Verarbeitung strukturierter Daten. Die automatische Generierung von Formatieranweisungen hilft zudem, die LLM-Ausgabe in die gewünschte Form zu lenken.

Für Anwendungen, die stark auf komplexe, strukturierte Daten angewiesen sind, bietet LangChain oft einen komfortableren und fehlertoleranteren Entwicklungsprozess. Die Qualität der `format_instructions` bei LangChain-Parsern oder der `description` bei OpenAI-Tools ist dabei entscheidend für die Zuverlässigkeit des LLMs, korrekt formatierte Daten zu liefern.⁸ Ungenaue Instruktionen können zu Parsing-Fehlern führen. Die Wahl des Mechanismus beeinflusst somit direkt die Robustheit der Anwendung. Während

LangChains Parser eine eingebaute Validierungslogik bieten, liegt bei direkter API-Nutzung die Verantwortung für die Validierung (z.B. von Tool Call Argumenten, die nicht immer valides JSON sein müssen⁹) und die Fehlerbehandlung stärker beim Entwickler.

Der Trend geht klar in Richtung zuverlässiger Extraktion strukturierter Daten. OpenAI forciert dies über `tool_calls` und die Vereinfachung der Tool-Nutzung in der neuen Responses API.¹ LangChain bietet hierfür eine breitere Palette an spezialisierten Parsern.

5 Aufruf von Sprachmodellen (LLM)

Der technische Aufruf des Sprachmodells, inklusive der Handhabung von Parametern wie Streaming, unterscheidet sich ebenfalls zwischen der direkten API-Nutzung und LangChain.

5.1 OpenAI API (Direkt)

Für Chat-Modelle ist die primäre Methode zum Aufruf des LLMs `client.chat.completions.create()`.² Mit der Einführung der **Responses API** (Stand Q1/2025) steht zusätzlich `client.responses.create()` zur Verfügung, welche insbesondere für komplexere Workflows mit Tool-Nutzung, Code-Ausführung und Zustandsmanagement konzipiert wurde.¹

Das Streaming von Antworten, also das empfangen der generierten Tokens in Echtzeit, wird durch Setzen des Parameters `stream=True` im API-Request aktiviert.⁴ Die Antwort wird dann als eine Serie von Events oder Chunks geliefert, die iterativ verarbeitet werden können.

Es ist wichtig zu beachten, dass ältere API-Endpunkte wie die `Completions API` (genutzt für Modelle wie `gpt-3.5-turbo-instruct`) zunehmend an Bedeutung verlieren und durch die `Chat Completions API` sowie die neue `Responses API` ersetzt werden. OpenAI fokussiert sich auf einen "conversation-in and message-out" Ansatz², und einige ältere Modelle werden sukzessive abgekündigt (`deprecated`).¹

Codebeispiel (Python, 2025-kompatibel): OpenAI API Streaming

```
from openai import OpenAI

client = OpenAI()

try:
```

```
# Streaming mit der Chat Completions API
stream = client.chat.completions.create(
    model="gpt-4o",
    messages=,
    stream=True,
)
print("OpenAI API (Streaming): ", end="")
for chunk in stream:
    if chunk.choices and chunk.choices.delta and chunk.choices.delta.content:
        print(chunk.choices.delta.content, end="", flush=True)
print("\n")

# Konzeptuelles Streaming mit der Responses API (Syntax kann leicht variieren)
# print("OpenAI Responses API (Streaming): ", end="")
# response_api_stream = client.responses.create(
#     model="gpt-4.1", # Beispielmodell für Responses API
#     input=[{"role": "user", "content": "Zähle bis 3."}],
#     stream=True
# )
# for event in response_api_stream:
#     if event.type == 'response.output_text.delta': # Beispielhafter Event-Typ
#         print(event.data.delta, end="", flush=True) # Zugriff auf Delta kann variieren
# print("\n(Beispiel für Responses API Streaming)")

except Exception as e:
    print(f"Ein Fehler ist aufgetreten: {e}")
```

Code basierend auf 4

5.2 LangChain (mit OpenAI)

In LangChain dient die Klasse `ChatOpenAI` aus dem Paket `langchain_openai` als Wrapper für die Chat-Modelle von OpenAI.11 Der Aufruf des Modells erfolgt über standardisierte Methoden:

- `invoke()` : Für eine einzelne, vollständige Antwort vom Modell.
- `stream()` : Für eine gestreamte Antwort, bei der die Tokens nacheinander eintreffen.⁵

Typischerweise wird das `ChatOpenAI`-Objekt mittels LCEL in eine Kette eingebunden, z.B. `prompt | llm | parser`.¹² LangChain abstrahiert dabei die direkten API-Aufrufe. Wenn die `stream()`-Methode verwendet wird, setzt LangChain intern die notwendigen Parameter (wie `stream=True` bei der OpenAI API) und liefert einen Iterator über die Antwort-Chunks. LangChain bietet auch die Möglichkeit, die Token-Nutzung über `usage_metadata` in der Antwort zu verfolgen.¹¹

Codebeispiel (Python, 2025-kompatibel): LangChain Streaming mit ChatOpenAI

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.messages import HumanMessage

llm = ChatOpenAI(model="gpt-4o", temperature=0)

prompt = ChatPromptTemplate.from_messages()

# Verkettung mit LCEL für Streaming
chain = prompt | llm # Für StrOutputParser würde man `| StrOutputParser()` anhängen

try:
    print("LangChain (Streaming): ", end="")
    # Die stream() Methode gibt einen Iterator über AIMessageChunk Objekte zurück
    for chunk in chain.stream({}):
        # Leeres Dict, da Prompt keine Variablen hat
        if chunk.content:
            print(chunk.content, end="", flush=True)
    print("\n")

    # Abrufen von usage_metadata nach einem invoke-Aufruf (nicht direkt beim Streamen jedes Chunks)
    # full_response = chain.invoke({})
    # if hasattr(full_response, 'usage_metadata') and full_response.usage_metadata:
    #     print(f"Token-Nutzung: {full_response.usage_metadata}")
```

```
except Exception as e:  
    print(f"Ein Fehler mit LangChain ist aufgetreten: {e}")
```

Code basierend auf 11

5.3 Vergleichende Analyse

Die direkte OpenAI API bietet unmittelbaren Zugriff auf die API-Endpunkte und volle Kontrolle über alle Request-Parameter. Die Einführung der Responses API 1 stellt eine signifikante Weiterentwicklung dar, die komplexere Interaktionsmuster nativ unterstützt.

LangChain vereinfacht den Aufruf von LLMs durch eine konsistente Schnittstelle (`invoke()`, `stream()`) und die nahtlose Integration in LCEL-Ketten. Dies reduziert die Komplexität des Codes für den Entwickler, führt aber auch eine zusätzliche Abstraktionsebene ein. Streaming ist bei beiden Ansätzen gut unterstützt und ermöglicht interaktive Anwendungen.

Die Weiterentwicklung der nativen OpenAI API-Primitiven, wie die Responses API, könnte die Notwendigkeit von Frameworks wie LangChain für bestimmte Anwendungsfälle reduzieren, insbesondere wenn die native API bereits "mächtig genug" wird und Aufgaben wie Zustandsmanagement oder vereinfachte Tool-Integration übernimmt.¹ Die Abstraktion durch LangChain kann die Entwicklung beschleunigen²⁰, birgt aber auch das Risiko, dass die Fehlersuche bei unerwartetem Verhalten oder bei Inkompatibilitäten mit neuesten API-Features erschwert wird, falls das Verhalten der Abstraktion nicht gänzlich verstanden ist.³⁹ Für Projekte, die ausschließlich OpenAI-Modelle nutzen, könnte die direkte API-Nutzung, insbesondere mit der Responses API, für viele Szenarien ausreichend und potenziell performanter sein.²⁰ LangChain behält seine Stärken bei Modellagnostik und der Implementierung sehr komplexer, benutzerdefinierter Logikketten.

6 Tool/Function Calling

Tool Calling (früher Function Calling) ist ein Mechanismus, der es LLMs ermöglicht, strukturierte Daten zu generieren oder externe Systeme und APIs aufzurufen, um ihre Fähigkeiten zu erweitern.

6.1 OpenAI API (Direkt)

Bei der direkten Nutzung der OpenAI API wird der `tools`-Parameter (der den älteren `functions`-Parameter ersetzt) im API-Aufruf verwendet.⁸ Diesem Parameter wird eine Liste von Tool-Definitionen übergeben. Jede Definition enthält:

- `type` : Muss "function" sein.
- `name` : Der Name der Funktion/des Tools.
- `description` : Eine für das LLM verständliche Beschreibung, was das Tool tut und wann es verwendet werden sollte. Klare und detaillierte Beschreibungen sind hier essenziell.⁸
- `parameters` : Ein JSON-Schema, das die vom Tool erwarteten Eingabeparameter definiert.

Das LLM entscheidet basierend auf dem Prompt und den Tool-Beschreibungen, ob und welches Tool mit welchen Argumenten aufgerufen werden soll. Wenn das Modell ein Tool nutzen möchte, enthält seine Antwort ein `tool_calls`-Objekt (oder mehrere bei parallelen Aufrufen). Dieses Objekt spezifiziert den Namen des Tools und die vom Modell generierten Argumente als JSON-String.⁸

Mit dem `tool_choice`-Parameter kann das Verhalten des Modells gesteuert werden:

- "auto" (Standard): Das Modell entscheidet frei.
- "required" : Das Modell muss mindestens ein Tool aufrufen.
- `{"type": "function", "name": "my_function"}` : Das Modell wird gezwungen, genau dieses spezifische Tool aufzurufen.⁸

Einige neuere Modelle unterstützen parallele Tool-Aufrufe, d.h. das Modell kann in einer einzigen Antwort mehrere Tool-Aufrufe anfordern.⁹ Der "strict mode" kann für Tool-Aufrufe aktiviert werden, um die Einhaltung des Schemas zu erzwingen, hat aber gewisse Einschränkungen, z.B. müssen alle Felder als `required` markiert und `additionalProperties` auf `false` gesetzt sein.⁸

Die **OpenAI Responses API** (Stand Q1/2025) integriert Tool-Nutzung weiter und bietet sogar eingebaute Tools wie `web_search_preview`, `file_search` (für RAG-ähnliche Funktionalität) und `computer_use_preview` (für Browser-Interaktionen).¹

Codebeispiel (Python, 2025-kompatibel): OpenAI API Tool Calling

```
import json
from openai import OpenAI
```

```
client = OpenAI()

# Definition des Tools
tools_definition = [
    {
        "name": "get_current_weather",
        "description": "Get the current weather for a given location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {"type": "string", "description": "The city and state for which to get the weather forecast."}
            }
        }
    }
]

messages = [{"role": "user", "content": "Wie ist das Wetter in London in Celsius?"}]

try:
    # Verwendung von client.chat.completions.create
    response = client.chat.completions.create(
        model="gpt-4o",
        messages=messages,
        tools=tools_definition,
        tool_choice="auto",  # oder z.B. {"type": "function", "name": "get_current_weather"}
    )
    response_message = response.choices.message

    if response_message.tool_calls:
        for tool_call in response_message.tool_calls:
            if tool_call.name == "get_current_weather":
                function_args = json.loads(tool_call.arguments)
                location = function_args.get("location")
                unit = function_args.get("unit", "celsius") # Default falls nicht spezifiziert
                print(f"OpenAI API - Tool Call: get_current_weather für {location} in {unit}")
                # Hier würde der tatsächliche Aufruf der Wetterfunktion erfolgen
                # z.B. weather_data = get_weather_function(location, unit)
                # Und dann eine neue Nachricht mit den Tool-Ergebnissen an die API senden
    else:
        print(f"OpenAI API - Direkte Antwort: {response_message.content}")
```

```
except Exception as e:  
    print(f"Ein Fehler ist aufgetreten: {e}")
```

Code basierend auf 8

6.2 LangChain (mit OpenAI)

LangChain vereinfacht die Arbeit mit Tools durch die Methode `bind_tools` des `ChatOpenAI`-Objekts.⁵ Diese Methode akzeptiert:

- Pydantic-Modelle: Das Schema der Tool-Parameter wird direkt aus dem Pydantic-Modell abgeleitet.
- Python-Funktionen: LangChain versucht, das Schema aus den Typ-Annotationen der Funktion zu inferieren.
- LangChain `Tool`-Objekte: Eine LangChain-spezifische Abstraktion für Tools.

LangChain konvertiert diese Definitionen intern in das von der OpenAI API erwartete JSON-Schemaformat.¹² Wenn das LLM ein Tool aufrufen möchte, enthält die zurückgegebene `AIMessage` ein `tool_calls`-Attribut. Dieses Attribut beinhaltet die vom Modell angeforderten Aufrufe in einem standardisierten, provider-agnostischen Format.¹²

Die eigentliche Stärke von LangChain liegt hier oft in der nachgelagerten Orchestrierung: die Ausführung der aufgerufenen Tools und die Rückmeldung der Ergebnisse an das LLM. Dies ist typischerweise Teil der Logik von LangChain Agents. LangChain bietet auch Möglichkeiten, das Erzwingen eines Tools oder das Deaktivieren paralleler Tool-Aufrufe zu steuern, falls vom Modell unterstützt.⁵

Codebeispiel (Python, 2025-kompatibel): LangChain Tool Calling mit Pydantic

```
from langchain_openai import ChatOpenAI  
from langchain_core.prompts import ChatPromptTemplate  
from langchain_core.pydantic_v1 import BaseModel, Field  
from langchain_core.messages import HumanMessage  
import json  
  
# Es wird davon ausgegangen, dass der OPENAI_API_KEY als Umgebungsvariable gesetzt ist.
```

```
llm = ChatOpenAI(model="gpt-4o", temperature=0)

# Definition des Tool-Schemas mit Pydantic
class GetWeatherTool(BaseModel):
    """Ruft das aktuelle Wetter für einen bestimmten Ort ab.""" # Docstring wird als Beschreibung genutzt
    location: str = Field(..., description="Der Ort, z.B. 'Berlin, Deutschland'")
    unit: str = Field(default="celsius", description="Die Temperatureinheit, 'celsius' oder 'fahrenheit'")

# Binden des Tools an das LLM
llm_with_tools = llm.bind_tools()

# Erstellen einer einfachen Kette
prompt = ChatPromptTemplate.from_messages([
    HumanMessage(content="{user_query}")
])
chain = prompt | llm_with_tools

try:
    ai_msg = chain.invoke({"user_query": "Wie ist das Wetter in Paris?"})

    if ai_msg.tool_calls:
        for tool_call in ai_msg.tool_calls:
            # tool_call ist hier ein LangChain ToolCall-Objekt
            # tool_call.name, tool_call.args (dict), tool_call.id
            if tool_call['name'] == "GetWeatherTool": # Name des Pydantic-Modells
                print(f"LangChain - Tool Call: {tool_call['name']} mit Argumenten: {tool_call['args']}"))
                # Hier würde die Logik zur Ausführung des Tools stehen
                # und das Ergebnis zurück an das LLM gesendet werden (typischerweise in einem Agenten-Loop)
    else:
        print(f"LangChain - Direkte Antwort: {ai_msg.content}")

except Exception as e:
    print(f"Ein Fehler mit LangChain ist aufgetreten: {e}")
```

Code basierend auf 11

6.3 Vergleichende Analyse

Beide Ansätze ermöglichen die leistungsstarke Tool Calling Funktionalität. Die direkte OpenAI API gibt dem Entwickler die volle Kontrolle über die JSON-Schema-Definition der Tools.

LangChain vereinfacht die Definition und das Binden von Tools erheblich, insbesondere durch die direkte Nutzung von Pydantic-Klassen oder Python-Funktionen. Die Abstraktion der Schema-Konvertierung kann Entwicklungszeit sparen. Die eigentliche Stärke von LangChain manifestiert sich jedoch oft in der Orchestrierung der Tool-Ausführungen und der komplexen Interaktionslogik, die in Agenten-Systemen benötigt wird.

Tool Calling hat sich als fundamentaler Mechanismus etabliert, um LLMs mit externen Datenquellen und Fähigkeiten auszustatten. OpenAI treibt diese Entwicklung aktiv voran, wie die Integration von Built-in Tools in die Responses API zeigt.¹ Die Klarheit und Präzision der Tool-Beschreibungen und Parameterdefinitionen ist für die zuverlässige Funktionsweise entscheidend.⁸ Vage oder missverständliche Beschreibungen führen dazu, dass das LLM falsche Tools wählt oder inkorrekte Argumente generiert.

Während die OpenAI API die grundlegende Infrastruktur für Tool Calls bereitstellt, kann LangChain den Aufwand für die Integration und Verwaltung multipler Tools, insbesondere in komplexen Agenten-Architekturen, deutlich reduzieren. Die Wahl hängt davon ab, ob eine "Batteries-included"-Lösung für die Tool-Orchestrierung gesucht wird (eher LangChain) oder ob die Logik der Tool-Auswahl und -Ausführung vollständig selbst implementiert werden soll (eher direkte OpenAI API). Die neuen "built-in tools" der OpenAI Responses API 1 könnten jedoch einfachere Anwendungsfälle ohne den Overhead von LangChain abdecken.

7 Fehlerbehandlung & Debugging

Robuste Fehlerbehandlung und effektive Debugging-Strategien sind unerlässlich für die Entwicklung zuverlässiger LLM-Applikationen.

7.1 OpenAI API (Direkt)

Die OpenAI API verwendet Standard-HTTP-Statuscodes, um allgemeine Fehler zu signalisieren (z.B. 4xx für Client-Fehler, 5xx für Server-Fehler). Detailliertere Fehlerinformationen werden im JSON-Body der Fehlerantwort bereitgestellt und enthalten typischerweise Felder wie `type` (z.B. `invalid_request_error`, `api_error`), `code` (ein spezifischer Fehlercode, z.B. `rate_limit_exceeded`, `model_not_found`) und `message` (eine menschenlesbare Beschreibung des Fehlers).¹⁵

Die offizielle OpenAI Python-Bibliothek fängt diese Fehler ab und wirft spezifische Python-Exceptions, die von `openai.APIError` erben, wie z.B. `openai.RateLimitError`, `openai.NotFoundError` oder `openai.BadRequestError` (welches `InvalidRequestError` abgelöst hat).¹⁵ Entwickler müssen diese Exceptions in ihrem Code explizit behandeln.

Beim Einsatz von Tool Calling ist es zudem wichtig, die vom Modell generierten JSON-Argumente auf Gültigkeit zu prüfen und potenzielle Fehler beim Parsen oder bei der Tool-Ausführung abzufangen.⁹ Das Debugging erfolgt primär durch sorgfältiges Logging der API-Requests und -Responses sowie die Analyse der zurückgegebenen Fehlermeldungen.

Codebeispiel (Python, 2025-kompatibel): OpenAI API Fehlerbehandlung

```
from openai import OpenAI, APIError, RateLimitError, BadRequestError

client = OpenAI()

try:
    chat_completion = client.chat.completions.create(
        model="gpt-4o-hopefully-valid-model", # Beispiel für potenziellen Fehler
        messages=[{"role": "user", "content": "Hallo Welt!"}]
)
#... Verarbeitung der Antwort...
```

```
print("OpenAI API: Erfolgreich.")

except RateLimitError as e:
    print(f"OpenAI API RateLimitError: Das Ratenlimit wurde überschritten. Details: {e}")
except BadRequestError as e:
    print(f"OpenAI API BadRequestError: Ungültige Anfrage. Details: {e}")
    # Beinhaltet Fehler wie ungültige Parameter, falsches Modell etc. [15]
except APIError as e:
    print(f"Allgemeiner OpenAI APIError: {e}")
except Exception as e:
    print(f"Ein unerwarteter Fehler ist aufgetreten: {e}")
```

7.2 LangChain (mit OpenAI)

LangChain fängt in der Regel Fehler von der zugrundeliegenden API (in diesem Fall der OpenAI API) ab und kann diese entweder direkt weiterleiten oder in eigene, LangChain-spezifische Exceptions umwandeln.

Ein besonderer Vorteil von LangChain liegt in den spezialisierten Output-Parsern, die Mechanismen zur Fehlerkorrektur bieten. Der `RetryOutputParser` und der `OutputFixingParser` können beispielsweise versuchen, Parsing-Fehler automatisch zu beheben, indem sie das LLM mit modifizierten Anweisungen erneut aufrufen.¹⁰ Der `RetryOutputParser` ist so konzipiert, dass er den ursprünglichen Prompt und die fehlerhafte Completion erneut an ein (möglicherweise anderes) LLM sendet, mit der Bitte, den Fehler zu korrigieren.¹¹

Für das Debugging und die Observability von LangChain-Anwendungen ist **LangSmith** ein zentrales Werkzeug.⁵ LangSmith ist eine Plattform, die detailliertes Tracing, Monitoring und Evaluierung von LLM-Applikationen ermöglicht – nicht nur für LangChain, sondern auch für andere Frameworks oder direkte API-Nutzung. Es bietet tiefe Einblicke in jeden Schritt einer Kette oder eines Agentenlaufs, visualisiert Token-Nutzung, Latenzen und aufgetretene Fehler.

Zusätzlich ermöglichen LangChain Callbacks das Einhaken in verschiedene Phasen der Ausführungskette für benutzerdefiniertes Logging oder spezifische Fehlerbehandlungs Routinen.⁵ LangGraph, eine Erweiterung von LangChain für den Bau zustandsbehafteter, Multi-Akteur-Anwendungen, bietet ebenfalls verbesserte Visualisierungs- und Debugging-Möglichkeiten für komplexe Abläufe.³⁷

Codebeispiel (Python, 2025-kompatibel): LangChain mit RetryOutputParser

```
from langchain_openai import ChatOpenAI, OpenAI # OpenAI für den Retry-LLM
from langchain_core.prompts import PromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field
from langchain_core.output_parsers import PydanticOutputParser, OutputParserException
from langchain.output_parsers.retry import RetryOutputParser # Import aus langchain.output_parsers

# Es wird davon ausgegangen, dass der OPENAI_API_KEY als Umgebungsvariable gesetzt ist.

# Pydantic Modell für die gewünschte Ausgabe
class Action(BaseModel):
    action: str = Field(description="Die auszuführende Aktion")
    action_input: str = Field(description="Die Eingabe für die Aktion")

# Haupt-LLM und Parser
main_llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0) # Ein günstigeres Modell, das eher Fehler macht
parser = PydanticOutputParser(pydantic_object=Action)

prompt = PromptTemplate(
    template="Der Nutzer möchte folgendes tun: {query}.\n{format_instructions}\nAntworte nur mit dem JSON-Objekt.",
    input_variables=["query"],
    partial_variables={"format_instructions": parser.get_format_instructions()}
)

# LLM für den RetryOutputParser (kann dasselbe oder ein anderes sein)
retry_llm = OpenAI(temperature=0) # Oft ein einfacheres, günstigeres Modell für Korrekturen

# Erstellen des RetryOutputParser
# Wichtig: Der RetryOutputParser erwartet, dass der `parser` einen String parst.
# Wenn der PydanticOutputParser direkt verwendet wird, kann es zu Typfehlern kommen,
# da PydanticOutputParser ein Pydantic-Objekt erwartet/liefert.
# Eine gängige Lösung ist, eine Kette zu bauen, die sicherstellt, dass der Retry-Mechanismus
```

```
# mit String-Repräsentationen arbeitet oder der Parser entsprechend angepasst wird.  
# Für dieses Beispiel wird eine vereinfachte Annahme getroffen, dass der Output des LLM  
# zunächst als String behandelt wird, bevor der PydanticParser ihn verarbeitet.  
  
# Korrekter Aufbau einer Kette mit RetryOutputParser ist komplexer,  
# hier ein konzeptioneller Hinweis auf die Nutzung.  
# Die direkte Integration von PydanticOutputParser in RetryOutputParser erfordert oft  
# eine sorgfältige Handhabung der Ein- und Ausgabetypen der beteiligten Parser.  
# Siehe LangChain Dokumentation für fortgeschrittene Muster.  
  
# Konzeptioneller Einsatz:  
# retry_parser = RetryOutputParser.from_llm(parser=parser, llm=retry_llm, prompt=prompt)  
# full_chain = prompt | main_llm | retry_parser # Dies kann zu Typfehlern führen, je nach Implementierung  
  
# Ein robusterer Ansatz mit RetryOutputParser involviert oft eine explizite Fehlerbehandlung  
# und erneuten Aufruf. Hier ein vereinfachtes Beispiel, das die Idee illustriert:  
  
bad_response_from_llm = '{"action": "search"}' # Fehlendes action_input  
  
try:  
    parsed_action = parser.parse(bad_response_from_llm)  
    print(f"LangChain - Direkt gelesen: {parsed_action}")  
except OutputParserException as e:  
    print(f"LangChain - Ursprünglicher Parsing-Fehler: {e}")  
    # Hier könnte der RetryOutputParser oder OutputFixingParser eingesetzt werden.  
    # Beispiel mit OutputFixingParser (ähnlich zu Retry, aber oft einfacher für Pydantic)  
    from langchain.output_parsers import OutputFixingParser  
    fix_parser = OutputFixingParser.from_llm(parser=parser, llm=ChatOpenAI(model="gpt-4o"))  
    try:  
        fixed_action = fix_parser.parse(bad_response_from_llm) # OutputFixingParser versucht, den Fehler zu beheben  
        print(f"LangChain - Korrigiert durch OutputFixingParser: {fixed_action}")  
    except OutputParserException as e_fix:  
        print(f"LangChain - Fehler auch nach Korrekturversuch: {e_fix}")
```

```
# Hinweis zur LangSmith-Integration (Tracing aktivieren):
# import os
# os.environ = "true"
# os.environ["LANGCHAIN_API_KEY"] = "DEIN_LANGSMITH_API_KEY" # Ersetzen durch eigenen Key
# os.environ = "Mein Projektnname" # Optional
# Nun würden alle LangChain Aufrufe automatisch an LangSmith gesendet.
```

Code basierend auf 16

7.3 Vergleichende Analyse

Die direkte OpenAI API liefert grundlegende Fehlerinformationen, deren Interpretation und Handhabung (z.B. Implementierung von Retry-Logik) dem Entwickler obliegt.

LangChain bietet hier deutlich erweiterte Möglichkeiten. Spezialisierte Parser wie `RetryOutputParser` können die Robustheit gegenüber fehlerhaften LLM-Ausgaben erhöhen, indem sie automatische Korrekturversuche unternehmen. Der entscheidende Vorteil für komplexe Anwendungen ist jedoch die Observability-Plattform LangSmith. Sie ermöglicht ein tiefgreifendes Verständnis des Anwendungsverhaltens, was für das Debugging nicht-deterministischer Systeme, die LLM-Ketten und Agenten beinhalten, unerlässlich ist.

Mit zunehmender Komplexität von LLM-Anwendungen wird eine effektive Fehlerbehandlung und ein leistungsfähiges Debugging immer wichtiger. Reine API-Fehlercodes reichen oft nicht aus, um das Verhalten von Systemen zu verstehen, die aus vielen miteinander verbundenen, potenziell fehleranfälligen Komponenten bestehen (API-Verfügbarkeit, Ratenlimits, fehlerhafte Prompts, falsche Tool-Auswahl, ungültige LLM-Outputs, Fehler in externen Tools). Die "Blackbox"-Natur von LLMs erschwert das Debugging zusätzlich. Werkzeuge wie LangSmith, die ein detailliertes Tracing jedes Schritts ermöglichen⁴⁵, sind daher von großem Wert, um Ursachen für unerwartetes Verhalten oder Fehler in komplexen LangChain-Anwendungen zu identifizieren. Ohne solche Werkzeuge ist man oft auf zeitaufwendiges "Print-Debugging" oder Vermutungen angewiesen. Die Investition in das Verständnis und die Nutzung von Observability-Tools kann die Entwicklungszeit signifikant verkürzen und die Zuverlässigkeit von LangChain-Anwendungen maßgeblich verbessern. Bei direkter OpenAI API-Nutzung müssen Entwickler äquivalente Logging- und Debugging-Strategien selbst aufbauen.

8 Einsatzszenarien und Best Practices

Die Entscheidung zwischen der direkten Nutzung der OpenAI API und dem Einsatz von LangChain hängt stark von den spezifischen Anforderungen des Projekts, der Komplexität der Anwendung und den Präferenzen des Entwicklungsteams ab.

8.1 Wann OpenAI API?

- **Einfache, klar definierte Aufgaben:** Wenn die Anwendung eine überschaubare Aufgabe erfüllt, wie z.B. eine einfache Frage-Antwort-Funktion oder Textgenerierung basierend auf einem statischen Prompt, und minimale Latenz sowie volle Kontrolle über den API-Request entscheidend sind.¹⁹ Die geringere Anzahl an Abstraktionsebenen kann hier zu einer besseren Performance führen.²⁰
- **Primäre Nutzung von OpenAI-Modellen:** Wenn ausschließlich OpenAI-Modelle zum Einsatz kommen und keine Notwendigkeit für Abstraktionen besteht, die einen Wechsel des LLM-Providers erleichtern würden.¹³
- **Schnelles Prototyping mit minimalem Setup:** Für sehr schnelle Prototypen oder Experimente, bei denen der Overhead eines zusätzlichen Frameworks vermieden werden soll und die Komplexität gering ist.¹³
- **Vermeidung von Framework-Abhängigkeiten:** Wenn Entwickler die "Magie" oder die zusätzlichen Abstraktionsebenen von LangChain als hinderlich empfinden oder eine direkte, transparente Kontrolle über jeden Aspekt der API-Interaktion bevorzugen.³⁹
- **Nutzung neuester OpenAI API-Features:** Wenn die neuesten Funktionen der OpenAI API (z.B. die Responses API mit serverseitigem State Management und Built-in Tools 1) die benötigte Funktionalität bereits nativ und zufriedenstellend abdecken.

8.2 Wann LangChain?

- **Komplexe Anwendungen:** Für Anwendungen, die komplexe Logik erfordern, wie die Verkettung mehrerer LLM-Aufrufe (Chains), die Implementierung von Agenten-Logik mit Entscheidungsfindung und Tool-Orchestrierung, oder anspruchsvolles Memory-Management für langanhaltende Konversationen.¹³
- **Retrieval Augmented Generation (RAG):** Wenn RAG ein Kernbestandteil der Anwendung ist, um LLMs mit externem Wissen anzureichern. LangChain bietet hierfür umfassende Komponenten (Document Loaders, Text Splitters, Vector Stores, Retrievers).¹⁴
- **Modellagnostik:** Wenn die Anwendung so konzipiert werden soll, dass potenziell verschiedene LLM-Provider (neben OpenAI auch Anthropic, Cohere, lokale Modelle etc.) genutzt oder einfach ausgetauscht werden können.¹³

- **Fortgeschrittenes Output-Parsing und Fehlerbehandlung:** Wenn robuste Mechanismen für das Parsen strukturierter Daten (z.B. mit Pydantic-Validierung) und automatische Fehlerkorrekturversuche (z.B. `RetryOutputParser`) benötigt werden.¹⁰
- **Observability und Debugging mit LangSmith:** Wenn die Nachvollziehbarkeit, das Monitoring und das Debugging komplexer LLM-Ketten und Agenten für die Entwicklung und Wartung essenziell sind. LangSmith ist hier ein mächtiges Werkzeug.¹⁷
- **Beschleunigung der Entwicklung:** Durch die Nutzung vorgefertigter Komponenten, Standardisierungen und Abstraktionen kann LangChain die Entwicklungszeit für bestimmte Anwendungsfälle verkürzen.²⁰

8.3 Best Practices für beide Ansätze:

- **OpenAI API (Direkt):**
 - **API-Key Management:** API-Keys niemals im Code hardcoden, sondern sicher über Umgebungsvariablen oder Secret-Management-Dienste verwalten.⁵⁶
 - **Nutzungsüberwachung:** Kosten und API-Nutzung regelmäßig überwachen, um Budgets einzuhalten.⁵⁶
 - **Prompt Engineering:** Klare, präzise und detaillierte Prompts formulieren. Dies gilt insbesondere für die Beschreibungen von Tools und deren Parametern, um die korrekte Auswahl und Anwendung durch das LLM sicherzustellen.⁸
 - **Fehlerbehandlung:** Eine robuste Fehlerbehandlung für API-Fehler (Netzwerkprobleme, Ratenlimits, ungültige Anfragen) und fehlerhafte Modellantworten implementieren.⁹
- **LangChain (mit OpenAI):**
 - **Verständnis der Abstraktionen:** Ein gutes Verständnis dafür entwickeln, wie LangChain-Komponenten intern funktionieren und auf die zugrundeliegende OpenAI API abgebildet werden.
 - **LangSmith nutzen:** Für Debugging, Monitoring und Evaluierung komplexer Ketten und Agenten LangSmith einsetzen.¹⁷
 - **Modellspezifisches Prompting:** Auch wenn LangChain eine gewisse Abstraktion bietet, können Prompts dennoch modellspezifische Optimierungen erfordern, um die besten Ergebnisse zu erzielen.⁵⁷
 - **Iterative Entwicklung:** Prompts, Ketten und Agenten iterativ entwickeln und kontinuierlich testen und verfeinern.⁵⁷
- **Für beide Ansätze:**
 - **Kostenmanagement:** Mit einfacheren, kostengünstigeren Modellen für das Prototyping beginnen und erst für die Produktion oder anspruchsvollere Aufgaben auf teurere Modelle umsteigen.³⁹
 - **Testing und Validierung:** Die generierten Antworten und das Verhalten der Anwendung kontinuierlich testen und validieren.

- **Sicherheit:** Sicherheitsaspekte berücksichtigen, insbesondere bei der Verwendung von Tools, die mit externen Systemen interagieren oder Code ausführen (Risiko von Prompt Injection).1

Die Grenze zwischen der "direkten API-Nutzung" und der "Framework-Nutzung" kann verschwimmen, da OpenAI selbst zunehmend High-Level-Funktionen in seine API integriert (z.B. die Responses API 1 oder die Assistants API 14). Dies könnte dazu führen, dass sich Frameworks wie LangChain stärker auf sehr komplexe Orchestrierungsaufgaben, Multi-Modell-Szenarien und spezialisierte Agenten-Architekturen konzentrieren. OpenAI hat ein natürliches Interesse daran, die Nutzung seiner API so einfach und leistungsfähig wie möglich zu gestalten, um Entwickler direkt an die eigene Plattform zu binden. Features, die häufig über Frameworks realisiert werden, könnten daher nativ implementiert werden, wenn eine breite Nachfrage besteht.

Die Wahl zwischen direkter API und LangChain ist oft eine Abwägung zwischen Entwicklungsgeschwindigkeit und Abstraktion auf der einen Seite und Performance, feingranularer Kontrolle und Transparenz auf der anderen Seite.20 LangChain kann die Entwicklungszeit für komplexe Systeme verkürzen 20, fügt aber eine zusätzliche Schicht hinzu, die potenziell Latenz verursachen oder das Debugging ohne Werkzeuge wie LangSmith erschweren kann. Es gibt keine "One-size-fits-all"-Lösung. Die "Best Practice" besteht darin, die Anforderungen des Projekts genau zu analysieren. Für ein Startup, das schnell einen MVP mit OpenAI entwickeln möchte, könnte die direkte API (ggf. unter Nutzung der Responses API) ideal sein. Für ein Unternehmen, das eine komplexe, modellagnostische Agentenplattform mit RAG und umfangreichem Tooling aufbaut, ist LangChain wahrscheinlich die passendere Wahl. Die Möglichkeit, LangChain für die übergeordnete Orchestrierung zu nutzen und dabei spezifische Aufgaben an OpenAI-Agenten (oder die Responses API) zu delegieren, stellt einen interessanten hybriden Ansatz dar.13

9 Zusammenfassung

Der Vergleich zwischen der direkten Nutzung der OpenAI API und dem Einsatz von LangChain (mit OpenAI-Modellen) zeigt, dass beide Ansätze ihre Berechtigung und spezifische Stärken haben, die je nach Anwendungsfall und Entwicklerpräferenz zum Tragen kommen.

Kernunterschiede und Trade-offs:

- **Direkte OpenAI API:** Bietet maximale Kontrolle, Transparenz und potenziell geringere Latenz durch weniger Abstraktionsebenen. Der Entwickler interagiert unmittelbar mit den API-Primitiven von OpenAI. Dies erfordert jedoch oft mehr manuellen Code für Aufgaben wie Prompt-Management, Output-Parsing komplexer Strukturen (abseits von Tool Calls), Zustandsmanagement (außer bei Nutzung der

neueren Responses API) und Fehlerbehandlung. Die Lernkurve für Basisfunktionalitäten kann flacher sein, aber die Implementierung fortgeschritten Muster liegt vollständig in der Verantwortung des Entwicklers.

- **LangChain (mit OpenAI):** Stellt eine Abstraktionsebene über der direkten API bereit und bietet eine Fülle von modularen Komponenten, standardisierten Schnittstellen (LCEL) und Werkzeugen (PromptTemplates, OutputParser, Memory-Module, Agents). Dies kann die Entwicklung komplexer Anwendungen beschleunigen, die Wiederverwendbarkeit von Code fördern und die Integration verschiedener LLMs oder Datenquellen erleichtern. Die Observability-Plattform LangSmith ist ein signifikanter Mehrwert für das Debugging und Monitoring. Der Preis für diese Abstraktion kann eine etwas höhere Latenz, eine steilere Lernkurve für das Framework selbst und eine Abhängigkeit von der Aktualität und Stabilität des Frameworks sein.

Die Entscheidung ist somit ein Abwagen zwischen Direktheit und Kontrolle (OpenAI API) versus Abstraktion und Orchestrierungsleistung (LangChain). Es geht um Performance versus Entwicklungsgeschwindigkeit und Einfachheit für simple Aufgaben versus Funktionsumfang für komplexe Systeme.

Handlungsempfehlungen basierend auf Projektszenarien (Stand 2025):

1. **Szenario 1: Einfache LLM-Integration, Fokus auf OpenAI, schnelle Prototypen, geringe Komplexität.**
 - **Empfehlung:** Direkte Nutzung der **OpenAI API**.
 - **Begründung:** Für klar definierte, überschaubare Aufgaben ist der direkte Weg oft der effizienteste und performanteste. Die neue **OpenAI Responses API** 1 sollte hier besonders evaluiert werden, da sie Funktionen wie serverseitiges Zustandsmanagement und integrierte Tools bietet, die den Bedarf an externen Frameworks für viele gängige Anwendungsfälle reduzieren können.
2. **Szenario 2: Komplexe Workflows, Agenten-Systeme, Retrieval Augmented Generation (RAG), Bedarf an Modellflexibilität oder fortgeschrittener Orchestrierung.**
 - **Empfehlung:** Einsatz von **LangChain** mit einem OpenAI-Modell als Backend.
 - **Begründung:** LangChain spielt seine Stärken bei der Verkettung von Komponenten (Chains), der Implementierung von Agenten mit Tool-Nutzung und Entscheidungslogik, der Integration von RAG-Pipelines und dem Management von Konversationshistorie aus.¹⁸ Die LangChain Expression Language (LCEL)²⁹, spezialisierte Output-Parser¹⁰ und die Debugging-Möglichkeiten mit LangSmith⁴⁵ sind hier wertvolle Werkzeuge. Die prinzipielle Modellagnostik von LangChain bietet zudem Zukunftssicherheit.¹³
3. **Szenario 3: Lernprojekte, Experimente und schrittweise Komplexitätssteigerung.**
 - **Empfehlung:** Mit der **direkten OpenAI API beginnen**, um ein grundlegendes Verständnis für die Funktionsweise von LLMs, Prompting und API-Interaktionen zu entwickeln. Anschließend **LangChain evaluieren**, um Konzepte wie Chains, Agents und

fortgeschrittenes Output-Parsing in einem strukturierten Rahmen zu erkunden und die Abstraktionsvorteile für komplexere Aufgaben kennenzulernen.

Abschließender Gedanke zur Zukunft:

Die Landschaft der LLM-APIs und -Frameworks ist extrem dynamisch. OpenAI entwickelt seine APIs kontinuierlich weiter und integriert Funktionalitäten, die zuvor Domänen von Frameworks waren (siehe Responses API 1 und Assistants API 14). Gleichzeitig erweitern Frameworks wie LangChain und dessen Ökosystem (z.B. LangGraph 37) ständig ihren Funktionsumfang und verbessern ihre Werkzeuge. Für Entwickler ist es daher unerlässlich, die aktuellen Entwicklungen aufmerksam zu verfolgen und die Wahl ihrer Werkzeuge regelmäßig zu überprüfen, um die für ihre spezifischen Bedürfnisse am besten geeignete und zukunftssicherste Lösung zu finden. Eine hybride Nutzung, bei der LangChain für die Orchestrierung und die direkte OpenAI API (oder spezifische OpenAI-Agenten-Features) für Kernaufgaben genutzt wird, könnte in vielen Fällen einen optimalen Mittelweg darstellen.¹³

10 Referenzen

1. OpenAI API: Responses vs. Chat Completions - Simon Willison's Weblog, Zugriff am Mai 6, 2025, <https://simonwillison.net/2025/Mar/11/responses-vs-chat-completions/>
2. Work with chat completion models - Azure OpenAI Service - Learn Microsoft, Zugriff am Mai 6, 2025, <https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/chatgpt>
3. API Reference - OpenAI API, Zugriff am Mai 6, 2025, <https://platform.openai.com/docs/api-reference/chat>
4. Streaming API responses - OpenAI API, Zugriff am Mai 6, 2025, <https://platform.openai.com/docs/guides/streaming-responses>
5. How-to guides - LangChain, Zugriff am Mai 6, 2025, https://python.langchain.com/docs/how_to/
6. Quick reference | 🦜 LangChain, Zugriff am Mai 6, 2025, https://python.langchain.com/v0.1/docs/modules/model_io/prompts/quick_start/
7. Langchain ChatOpenAI Examples - Restack, Zugriff am Mai 6, 2025, <https://www.restack.io/docs/langchain-knowledge-chatopenai-examples-cat-ai>
8. Function calling - OpenAI API, Zugriff am Mai 6, 2025, <https://platform.openai.com/docs/guides/function-calling>
9. How to use function calling with Azure OpenAI Service - Learn Microsoft, Zugriff am Mai 6, 2025, <https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/function-calling>

10. A Comprehensive Guide to Output Parsers - Analytics Vidhya, Zugriff am Mai 6, 2025,
<https://www.analyticsvidhya.com/blog/2024/11/output-parsers/>
11. ChatOpenAI — LangChain documentation, Zugriff am Mai 6, 2025,
https://python.langchain.com/api_reference/openai/chat_models/langchain_openai.chat_models.base.ChatOpenAI.html
12. ChatOpenAI | 🦜 LangChain, Zugriff am Mai 6, 2025, <https://python.langchain.com/v0.1/docs/integrations/chat/openai/>
13. LangChain, OpenAI Agents, and the Rise of the Agentic Stack - FullStack Labs, Zugriff am Mai 6, 2025,
<https://www.fullstack.com/labs/resources/blog/langchain-openai-agents-and-the-agentic-stack>
14. Assistants API vs using LangChain (or other) library - OpenAI Developer Forum, Zugriff am Mai 6, 2025,
<https://community.openai.com/t/assistants-api-vs-using-langchain-or-other-library/956187>
15. Error code for OpenAI Chat Completion - API, Zugriff am Mai 6, 2025, <https://community.openai.com/t/error-code-for-openai-chat-completion/1102402>
16. RetryOutputParser — LangChain documentation, Zugriff am Mai 6, 2025,
https://api.python.langchain.com/en/latest/langchain/output_parsers/langchain.output_parsers.retry.RetryOutputParser.html
17. An Introduction to Debugging And Testing LLMs in LangSmith - DataCamp, Zugriff am Mai 6, 2025,
<https://www.datacamp.com/tutorial/introduction-to-langsmith>
18. Conceptual guide | 🦜 LangChain, Zugriff am Mai 6, 2025, <https://python.langchain.com/docs/concepts/>
19. LangChain vs OpenAI API – Which One Should You Choose for AI Chatbots?, Zugriff am Mai 6, 2025,
https://searchcreators.org/search_blog/post/langchain-vs-openai-api-which-one-should-you-choos/
20. LangChain vs OpenAI API: When Simplicity Meets Scalability | Aditya Bhattacharya, Zugriff am Mai 6, 2025, <https://blogs.adityabh.is-a.dev/posts/langchain-vs-openai-simplicity-vs-scalability/>
21. langchain · PyPI, Zugriff am Mai 6, 2025, <https://pypi.org/project/langchain/>
22. Prompt Templates | 🦜 LangChain, Zugriff am Mai 6, 2025, https://python.langchain.com/docs/concepts/prompt_templates/
23. langchain_core.prompts.chat.ChatPromptTemplate — LangChain 0.2.17, Zugriff am Mai 6, 2025,
https://api.python.langchain.com/en/latest/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html
24. ChatPromptTemplate — LangChain documentation, Zugriff am Mai 6, 2025,
https://python.langchain.com/v0.2/api_reference/core/prompts/langchain_core.prompts.chat.ChatPromptTemplate.html
25. langchain.prompts.chat.ChatPromptTemplate, Zugriff am Mai 6, 2025, <https://sj-langchain.readthedocs.io/en/latest/prompts/langchain.prompts.chat.ChatPromptTemplate.html>

26. LangChain Expression Language Explained - Pinecone, Zugriff am Mai 6, 2025,
<https://www.pinecone.io/learn/series/langchain/langchain-expression-language/>
27. 11-langchain-expression-language.ipynb - GitHub, Zugriff am Mai 6, 2025, <https://github.com/pinecone-io/examples/blob/master/learn/generation/langchain/handbook/11-langchain-expression-language.ipynb>
28. LangChain Expression Language (LCEL), Zugriff am Mai 6, 2025, <https://js.langchain.com/docs/concepts/lcel/>
29. LangChain Expression Language (LCEL), Zugriff am Mai 6, 2025, <https://python.langchain.com/docs/concepts/lcel/>
30. LangChain Expression Language (LCEL), Zugriff am Mai 6, 2025, https://python.langchain.com/v0.1/docs/expression_language/
31. How to parse JSON output | 🦜 LangChain, Zugriff am Mai 6, 2025, https://python.langchain.com/docs/how_to/output_parser_json/
32. PydanticOutputParser — LangChain documentation, Zugriff am Mai 6, 2025,
https://python.langchain.com/api_reference/core/output_parsers/langchain_core.output_parsers.pydantic.PydanticOutputParser.html
33. langchain_core.output_parsers.pydantic.PydanticOutputParser — LangChain 0.2.17, Zugriff am Mai 6, 2025,
https://api.python.langchain.com/en/latest/output_parsers/langchain_core.output_parsers.pydantic.PydanticOutputParser.html
34. Pydantic parser - LangChain, Zugriff am Mai 6, 2025,
https://python.langchain.com/v0.1/docs/modules/model_io/output_parsers/types/pydantic/
35. openai-python/examples/streaming.py at main - GitHub, Zugriff am Mai 6, 2025, <https://github.com/openai/openai-python/blob/main/examples/streaming.py>
36. Understanding which models are available to call with the APIs - OpenAI Developer Forum, Zugriff am Mai 6, 2025,
<https://community.openai.com/t/understanding-which-models-are-available-to-call-with-the-apis/1141192>
37. Introduction | 🦜 LangChain, Zugriff am Mai 6, 2025, <https://python.langchain.com/docs/introduction/>
38. ChatOpenAI — LangChain documentation, Zugriff am Mai 6, 2025,
https://api.python.langchain.com/en/latest/openai/chat_models/langchain_openai.chat_models.base.ChatOpenAI.html
39. Direct OpenAI API vs. LangChain: A Performance and Workflow Comparison - Reddit, Zugriff am Mai 6, 2025,
https://www.reddit.com/r/LangChain/comments/1hd6w5x/direct_openai_api_vs_langchain_a_performance_and/
40. Direct OpenAI API vs. LangChain: A Performance and Workflow Comparison - Reddit, Zugriff am Mai 6, 2025,
https://www.reddit.com/r/OpenAI/comments/1hd6x21/direct_openai_api_vs_langchain_a_performance_and/
41. Prompting Best Practices for Tool Use (Function Calling) - OpenAI Developer Forum, Zugriff am Mai 6, 2025,
<https://community.openai.com/t/prompting-best-practices-for-tool-use-function-calling/1123036>

42. How to retry when a parsing error occurs - LangChain, Zugriff am Mai 6, 2025,
https://python.langchain.com/docs/how_to/output_parser_retry/
43. OutputFixingParser — LangChain 0.0.149 - Read the Docs, Zugriff am Mai 6, 2025,
https://langchain.readthedocs.io/en/stable/modules/prompts/output_parsers/examples/output_fixing_parser.html
44. Langchain vs Langsmith: Framework Comparison + Alternatives | Generative AI Collaboration Platform - Orq.ai, Zugriff am Mai 6, 2025,
<https://orq.ai/blog/langchain-vs-langsmith>
45. LangSmith - LangChain, Zugriff am Mai 6, 2025, <https://www.langchain.com/langsmith>
46. Breaking Down Langchain vs Langsmith for Smarter AI App Building - Lamicat.ai Labs, Zugriff am Mai 6, 2025,
<https://blog.lamicat.ai/guides/langchain-vs-langsmith/>
47. LangChain Debug Guide — Restack, Zugriff am Mai 6, 2025, <https://www.restack.io/docs/langchain-knowledge-langchain-debug-guide>
48. LangChain, Zugriff am Mai 6, 2025, <https://www.langchain.com/>
49. Revolutionizing AI with LangChain, LangGraph, LangSmith - MyScale, Zugriff am Mai 6, 2025, <https://myscale.com/blog/langchain-langgraph-langsmith-game-changers-now/>
50. LangSmith Pricing - LangChain, Zugriff am Mai 6, 2025, <https://www.langchain.com/pricing-langsmith>
51. Announcing LangSmith, a unified platform for debugging, testing, evaluating, and monitoring your LLM applications - LangChain Blog, Zugriff am Mai 6, 2025, <https://blog.langchain.dev/announcing-langsmith/>
52. Spoke to 22 LangGraph devs and here's what we found : r/LangChain - Reddit, Zugriff am Mai 6, 2025,
https://www.reddit.com/r/LangChain/comments/1eh0ly3/spoke_to_22_langgraph_devs_and_heres_what_we_found/
53. `RetryOutputParser` error when used with `PydanticOutputParser` · Issue #19145 - GitHub, Zugriff am Mai 6, 2025,
<https://github.com/langchain-ai/langchain/issues/19145>
54. Why use Langchain Libraries for instead of OpenAI Libraries? - YouTube, Zugriff am Mai 6, 2025, https://www.youtube.com/watch?v=g-VHv_IpbTM
55. Switching from Direct calls to OpenAI to Langchain - DevTools daily, Zugriff am Mai 6, 2025,
<https://www.devtoolsdaily.com/blog/switching-from-direct-openai-to-langchain/>
56. Production best practices - OpenAI API, Zugriff am Mai 6, 2025, <https://platform.openai.com/docs/guides/production-best-practices>
57. Langchain Vs Openai Api Comparison - Restack, Zugriff am Mai 6, 2025, <https://www.restack.io/docs/langchain-knowledge-langchain-vs-openai-cat-ai>

M05 - Transformer



Anwendung Generativer KI

Stand: 03.2025

1 Was ist ein Transformer?

Ein Transformer ist eine spezielle Art von neuralem Netzwerk, das seit 2017 die KI-Welt revolutioniert hat. Diese Architektur steckt hinter bekannten Textgeneratoren wie ChatGPT, Llama und Gemini. Transformer werden auch für Bilder, Audio und andere Anwendungen genutzt.

2 Textgenerierende Transformer

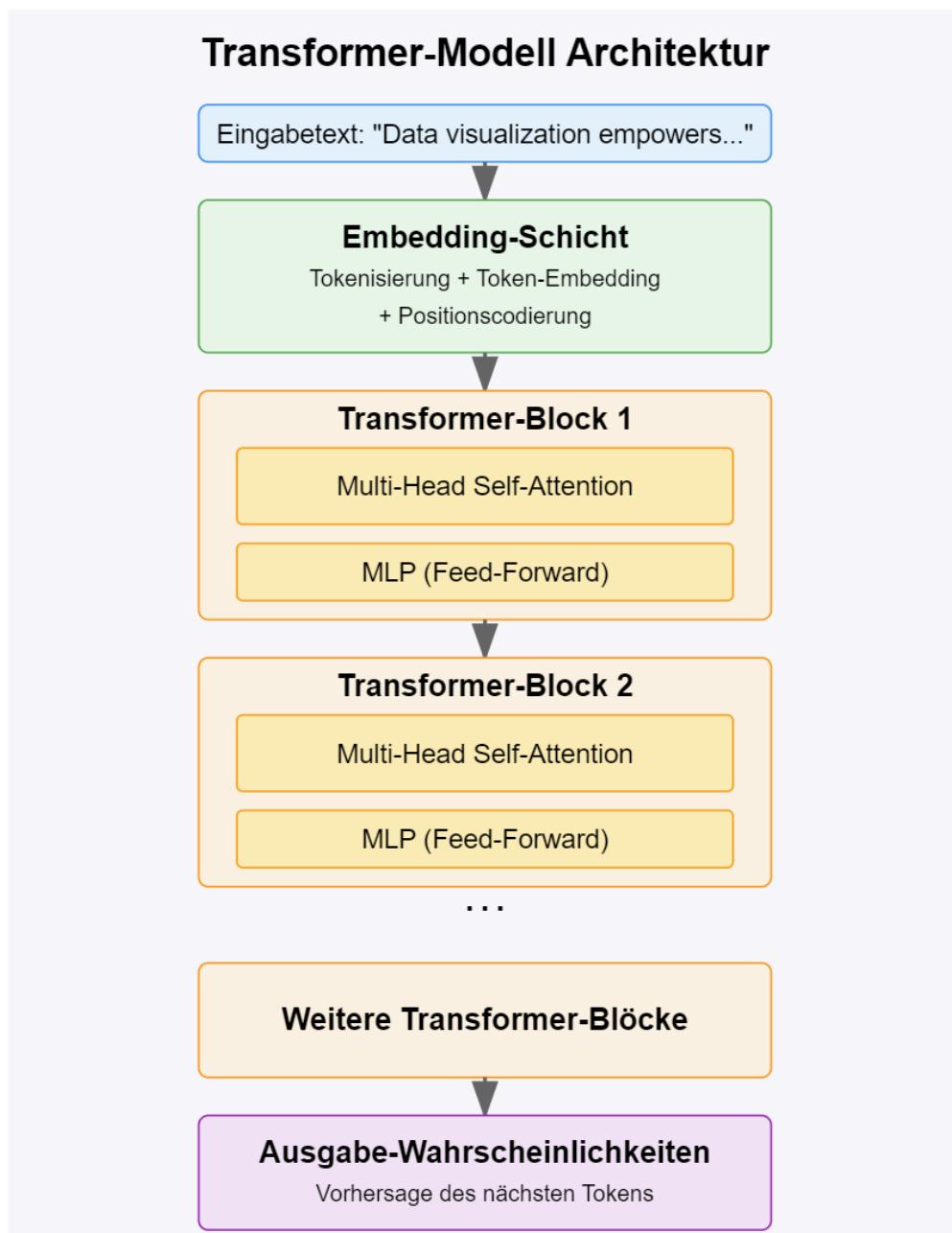
Sie arbeiten nach dem Prinzip "Was kommt als nächstes?": Wenn du einen Text eingibst, berechnet das Modell, welches Wort mit der höchsten Wahrscheinlichkeit folgen sollte. Der wichtigste Teil ist der **Self-Attention-Mechanismus**, der es dem Modell erlaubt, Beziehungen zwischen allen Wörtern in einem Text zu verstehen.

Ein bekanntes Beispiel ist GPT-2 (small) mit 124 Millionen Parametern - kein Riese nach heutigen Standards, aber ein gutes Modell zum Verständnis der Grundlagen.

3 Aufbau eines Transformer-Modells

Ein Transformer besteht aus drei Hauptteilen:

1. **Embedding-Schicht**: Wandelt Text in Zahlen um
2. **Transformer-Blöcke**: Verarbeiten diese Zahlen
3. **Ausgabe-Schicht**: Berechnet Wahrscheinlichkeiten für das nächste Wort



3.1 Embedding - Text in Zahlen

Bevor ein Transformer arbeiten kann, muss Text in Zahlen umgewandelt werden:

1. **Tokenisierung:** Der Text wird in Einzelteile (Token) zerlegt. Diese können ganze Wörter oder Wortteile sein.
2. **Token-Embedding:** Jedem Token wird ein Zahlenvektor zugewiesen (bei GPT-2 small sind das 768 Zahlen pro Token).
3. **Positionscodierung:** Dem Modell wird mitgeteilt, an welcher Position jedes Token steht.

Beispiel: "Data visualization empowers users to..." wird in Token zerlegt und jedes Token bekommt einen eigenen Zahlenvektor.

3.2 Transformer-Blöcke: Das Herzstück

Hier findet die eigentliche Verarbeitung statt. Jeder Block enthält:

- **Multi-Head Self-Attention:** Ermöglicht dem Modell, auf relevante Teile des Textes zu "achten"
- **MLP (Mehrschichtiges Perzeptron):** Ein kleines neuronales Netzwerk zur Weiterverarbeitung

Mehrere **Transformer-Blöcke** werden hintereinandergeschaltet, damit das Modell die Informationen schrittweise immer besser verstehen kann.

Jeder Block verfeinert die Darstellung ein Stück weiter:

- Frühere Blöcke erkennen eher einfache Muster (z. B. Beziehungen zwischen einzelnen Wörtern),
- spätere Blöcke erkennen komplexere Zusammenhänge (z. B. die Bedeutung ganzer Sätze oder Abschnitte).

Durch diese gestufte Verarbeitung kann das Modell tiefere, genauere Einsichten in den Text gewinnen.

3.2.1 (Multi-Head) Self-Attention

Dies ist der Kern eines Transformers. Für jeden Token werden drei Arten von Vektoren berechnet:

- **Query (Q):** "Was suche ich?" - ähnlich einer Suchanfrage
- **Key (K):** "Wo könnte es sein?" - ähnlich Seitentiteln
- **Value (V):** "Was ist der Inhalt?" - die eigentliche Information

Durch Berechnungen mit diesen Vektoren bestimmt das Modell, wie stark jedes Wort auf andere Wörter "achten" soll. GPT-Modelle verwenden dabei "maskierte Attention", was bedeutet, dass ein Wort nur auf vorangegangene Wörter achten darf, nicht auf zukünftige.

Die Attention wird auf mehrere "Köpfe" (Heads) aufgeteilt, damit das Modell unterschiedliche Beziehungen zwischen Wörtern lernen kann.

3.2.2 MLP (Mehrschichtiges Perzeptron):

Im Transformer verarbeitet das **MLP** die Informationen, die die Attention gesammelt hat, noch einmal weiter.

Es hilft dem Modell dabei, wichtige Zusammenhänge und Muster besser zu erkennen und die Darstellung der Daten zu verfeinern.

Das Ergebnis ist eine stärkere, "intelligentere" Repräsentation des Textes, bevor er an den nächsten Block weitergegeben wird.

3.3 Ausgabe-Wahrscheinlichkeiten

Am Ende berechnet das Modell Wahrscheinlichkeiten für jedes mögliche nächste Wort. Mit verschiedenen Parametern kann man die Textgenerierung steuern:

- **Temperatur:** Steuert, wie "kreativ" oder wie "sicher" die Antworten sind
 - Niedrige Temperatur: vorhersehbare, sichere Antworten
 - Hohe Temperatur: kreativere, überraschendere Antworten
- **Top-k-Sampling:** Beschränkt die Auswahl auf die k wahrscheinlichsten Wörter
- **Top-p-Sampling:** Wählt aus einer dynamischen Anzahl wahrscheinlicher Wörter aus

4 Python-Beispiel für einen einfachen Transformer

Hier ist ein einfaches Beispiel, wie man mit der Hugging Face Transformers-Bibliothek einen vortrainierten Transformer verwenden kann:

```
import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer

def generate_text(prompt, max_length=20):
    """Generiert Text mit GPT-2 basierend auf einem Eingabeprompt"""

    # Modell und Tokenizer laden
    tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
    model = GPT2LMHeadModel.from_pretrained('gpt2')

    # Tokenizer konfigurieren und Text vorbereiten
    tokenizer.pad_token = tokenizer.eos_token
    inputs = tokenizer(prompt, return_tensors="pt", padding=True)

    # Text generieren
    outputs = model.generate(
        inputs["input_ids"],
        attention_mask=inputs["attention_mask"],
        max_length=max_length,
        temperature=0.7,           # Kreativitätsparameter
        top_k=10,                  # Top-k Sampling
        do_sample=True,             # Sampling aktivieren
        pad_token_id=tokenizer.eos_token_id
    )

    # Ergebnis dekodieren und zurückgeben
    return tokenizer.decode(outputs[0], skip_special_tokens=True)

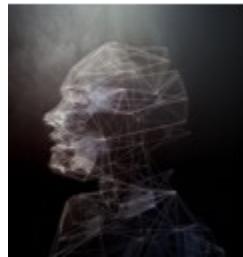
# Beispiel ausführen
if __name__ == "__main__":
```

```
prompt = "Data visualization empowers users to"  
print(generate_text(prompt))
```

💡 Tip

[Transformer Explainer: LLM Transformer Model Visually Explained](#)

M08a - Tokenizing & Chunking



Anwendung Generativer KI

Stand: 02.2025

Die effiziente Textverarbeitung beruht auf drei zentralen Elementen: der Wahl des richtigen Tokenizers, der optimalen Chunk-Größe und einer passenden Chunking-Strategie. Diese Faktoren bilden die Basis für eine erfolgreiche Dokumentenanalyse in NLP-Anwendungen. Im Folgenden erfahren Sie, wie Sie diese Parameter systematisch an die Eigenschaften Ihres Dokuments und die Anforderungen der Anwendung (z. B. Fragebeantwortung, Zusammenfassung, Code-Verarbeitung) anpassen und optimieren können.

1 Tokenizer-, Chunking- & Strategieauswahl

1.1 Dokumenttypen

Dokumenttyp	Empfohlener Tokenizer	Chunk-Größe (Tokens)	Überlappung (%)	Empfohlene Chunking-Strategie	Begründung
Lange Texte	SentencePiece oder BPE	512–1024	20–30%	Semantisches & embeddingbasiertes Chunking	Diese Tokenizer zerlegen den Text in kleinere, semantisch sinnvolle Einheiten. Größere Chunks helfen, den Kontext beizubehalten und logische Einheiten in dichten Texten zu bewahren.
Mittel-lange Texte	WordPiece	256–512	10–20%	Semantisches Chunking	WordPiece verarbeitet gemischte Sprache gut. Semantisches Chunking fasst narrative und strukturierte Abschnitte optimal zusammen, ohne den Text zu stark zu fragmentieren.
Kurze Texte	Whitespace-/Symbol-basierte Tokenizer	50–200	0–5%	Rekursives Zeichen-Chucking (bei unklaren Grenzen)	Kurze, oft stark strukturierte Texte profitieren von kleinen Chunks. Rekursives Zeichen-Chucking kann helfen, bei fehlenden klaren Grenzen die Struktur zu wahren.

Dokumenttyp	Empfohlener Tokenizer	Chunk-Größe (Tokens)	Überlappung (%)	Empfohlene Chunking-Strategie	Begründung
Code & Technische Dokumente	Whitespace- oder benutzerdefinierte symbolbasierte Tokenizer (mit funktionsspezifischen Regeln)	Basierend auf logischen Blöcken (z. B. pro Funktion oder Absatz, ca. 256 Tokens)	Variabel (idealerweise minimale Überlappung oder blockgrenzenangepasst)	Agentisches Chunking (unter Einbeziehung logischer und syntaktischer Strukturen)	Die strukturelle Integrität ist entscheidend, um die Semantik des Codes zu erhalten. Agentisches Chunking berücksichtigt funktionale Zusammenhänge und stellt die Intaktheit der Blöcke sicher.

1.2 Anwendungsszenarien

Szenario	Ziel	Empfohlenes Chunking	Empfohlene Strategie	Begründung
Antworten auf Fragen	Exakte Extraktion relevanter Passagen	Moderat bis große Chunks (512 Tokens bei langen Texten) mit hoher Überlappung (30–50%)	Kombination aus semantischem und embeddingbasiertem Chunking	Hohe Überlappung stellt sicher, dass der Kontext zwischen den Chunks nicht verloren geht. Semantische Grenzen und embeddingbasierte Analysen erfassen relevante Abschnitte präzise.
Zusammenfassungen	Verdichtung des Inhalts bei Beibehaltung der Kernaussagen	Mittlere Chunks (256 Tokens) mit moderater Überlappung (10–20%)	Semantisches Chunking	Semantisches Chunking bewahrt komplett Sinnabschnitte, sodass die Kernaussagen klar extrahiert werden können, ohne den Kontext zu verlieren.
Informationsretrieval (RAG)	Effiziente Auffindbarkeit relevanter Abschnitte	Chunks von 256–512 Tokens mit moderater Überlappung (10–20%)	Embeddingbasiertes Chunking	Embeddingbasiertes Chunking gruppier semantisch verwandte Inhalte. So werden relevante Informationen leichter auffindbar und retrieval-technisch optimal aufbereitet.

Szenario	Ziel	Empfohlenes Chunking	Empfohlene Strategie	Begründung
Named Entity Recognition (NER)	Identifikation wichtiger Entitäten (Namen, Daten usw.)	Chunks, die an Satzgrenzen ausgerichtet sind (ca. 256 Tokens) und minimale Überlappung (5–15%)	Semantisches Chunking (ggf. kombiniert mit embeddingbasierten Ansätzen)	Durch an Satzgrenzen ausgerichtete Chunks wird vermieden, dass Entitäten aufgespalten werden. Eine embeddingbasierte Analyse kann zusätzlich helfen, zusammengehörige Entitäten zu erfassen.
Textklassifikation	Zuweisung von Labels zu Dokumenten oder Abschnitten	Größere, grobere Chunks (gesamtes Dokument oder 512 Tokens) mit wenig bis keiner Überlappung	Semantisches Chunking (optional mit reduzierter Granularität)	Gröbere Unterteilungen verhindern Rauschen, während semantische Einheiten erhalten bleiben, die für die Klassifikation relevant sind.
Code-Kommentierung/Erklärung	Verständnis und Erklärung von Codeabschnitten	Chunks, die durch logische Blöcke definiert sind (z. B. pro Funktion, Modul) mit Überlappung nur, wenn notwendig (blockgrenzenbezogen)	Agentisches Chunking	Agentisches Chunking berücksichtigt syntaktische und semantische Aspekte des Codes. So bleiben logische Zusammenhänge, wie Funktionsdefinitionen, erhalten und können optimal erklärt werden.



Bevor Sie eine konkrete Implementierung starten, sollten Sie Ihre Dokumente genau analysieren, um die für Ihren Anwendungsfall optimale Kombination aus Tokenizer, Chunk-Größe, Überlappung und Chunking-Strategie auszuwählen. Eine Pilotphase mit verschiedenen Einstellungen kann helfen, den besten Ansatz zu ermitteln.

2 Beispiel

```
● ● ●

# Original Text
text = "Maschinelles Lernen ist ein spannendes Thema."

# Schritt 1: Text zu Token
text_tokens = ["Masch", "inelles", "_Lernen", "_ist", "_ein", "_spannendes", "_Thema", "."]

# Schritt 2: Token zu IDs
token_ids = [2847, 1123, 892, 345, 287, 4561, 1876, 13]

# Schritt 3: Chunking (Chunk-Größe = 4)
chunks = [
    # Chunk 1: ["Masch", "inelles", "_Lernen", "_ist"]
    [2847, 1123, 892, 345],

    # Chunk 2: ["_ist", "_ein", "_spannendes", "_Thema"]
    [345, 287, 4561, 1876]
]
```

- Tokenizing:
 - Zerlegt Text in kleinste Einheiten (Token)
 - Diese Token werden in Zahlen (IDs) umgewandelt
 - Ein Token kann ein Wort, Teil eines Wortes oder ein Satzzeichen sein
- Chunking:
 - Gruppiert die Token in verarbeitbare Blöcke
 - Beispiel: Bei max. 4096 Token pro Anfrage werden längere Texte in Chunks aufgeteilt
 - Jeder Chunk behält dabei genug Überlappung (hier 1) zum vorherigen Chunk für Kontexterhalt
- Zusammenspiel:

- Text wird erst tokenisiert (in kleinste Einheiten zerlegt)
- Die Token werden dann in Chunks gruppiert (für Verarbeitung)
- Chunks werden nacheinander verarbeitet
- LLM behält Kontext zwischen Chunks durch Überlappungen

3 Parameter- und Strategieauswahl

- **Tokenizer-Auswahl:**
 - **SentencePiece/BPE** sind ideal für lange, unstrukturierte Texte, da sie feine Subworteinheiten erzeugen und dabei semantische Bedeutung beibehalten.
 - **WordPiece** ist optimal für hybride Texte, in denen technische sowie allgemeine Sprache vorkommen.
 - **Whitespace-/Symbol-basierte Tokenizer** (oder speziell angepasste Tokenizer für Code) gewährleisten, dass die Struktur, beispielsweise in kurzen Texten oder Quellcode, erhalten bleibt.
- **Chunk-Größe und Überlappung:**
 - Die **Chunk-Größe** wird so gewählt, dass jeweils eine komplette logische Einheit erfasst wird. Längere Texte benötigen größere Chunks, während bei kurzen Texten kleinere, präzisere Segmente ausreichend sind.
 - **Überlappung** hilft dabei, Kontextinformationen am Rand der Chunks nicht zu verlieren. Für komplexe Aufgaben (wie präzise Fragebeantwortung) ist eine höhere Überlappung vorteilhaft, wohingegen bei Aufgaben wie Klassifikation geringere Überlappungen ausreichend sind.
- **Zusätzliche Chunking-Strategien:**
 - **Semantisches Chunking** zielt darauf ab, thematisch und inhaltlich zusammenhängende Abschnitte zu bilden.
 - **Rekursives Zeichen-Chucking** eignet sich, wenn keine klaren sprachlichen Grenzen vorliegen oder bei sehr strukturierten, kurzen Dokumenten.
 - **Embeddingbasiertes Chunking** nutzt Ähnlichkeiten im Einbettungsraum, um semantisch verwandte Inhalte zu gruppieren, was insbesondere bei Retrieval-Aufgaben nützlich ist.
 - **Agentisches Chunking** verwendet agentenbasierte Verfahren, um logische und syntaktische Zusammenhänge zu identifizieren – ein Ansatz, der besonders bei Code und technischen Dokumenten Vorteile bietet.
- **Praktische Rahmenbedingungen:**

- **Speicherverbrauch und Verarbeitungsgeschwindigkeit** lassen sich durch Anpassung der Chunk-Größe steuern. Kleinere Chunks reduzieren den Speicherbedarf und beschleunigen die Verarbeitung, was vor allem bei großen Datenmengen von Bedeutung ist.
- **Kosten** können durch die Optimierung der Überlappung minimiert werden. Eine zu hohe Überlappung erhöht Redundanzen und Rechenaufwand, sodass hier ein ausgewogenes Verhältnis gefunden werden muss.

Tip:



Tip

Erstellen Sie eine Checkliste für die Parameterwahl, die alle wesentlichen Aspekte – von Tokenizer-Auswahl über Chunk-Größe bis hin zur konkreten Chunking-Strategie – abdeckt. Dies unterstützt Sie dabei, systematisch vorzugehen und sicherzustellen, dass alle praktischen Rahmenbedingungen berücksichtigt werden.

4 Optimierung für verschiedene Modellgrößen

- **Große Modelle:**
 - Können größere Chunk-Größen (bis zu 1024 Tokens) verarbeiten, wodurch mehr Kontext erhalten bleibt.
 - Profitieren in Szenarien wie der Fragebeantwortung von einer höheren Überlappung (30–50%) und einer Kombination aus semantischem und embeddingbasiertem Chunking.
- **Kleinere Modelle:**
 - Sollten kleinere Chunk-Größen (z. B. 256–512 Tokens) verwenden, um den Speicherbedarf und die Verarbeitungsgeschwindigkeit zu optimieren.
 - Eine geringere Überlappung (10–20%) ist empfehlenswert, um unnötige Redundanz zu vermeiden, während dennoch ausreichend Kontext für die jeweilige Aufgabe erhalten bleibt.
- **Iterative Feinabstimmung:**
 - Es empfiehlt sich, anhand von Metriken wie F1-Score (bei Fragebeantwortung), ROUGE (bei Zusammenfassungen) und Recall@K (bei Retrieval-Aufgaben) verschiedene Einstellungen zu evaluieren und schrittweise zu optimieren.
 - Szenariospezifische Experimente können helfen, die Wahl des Tokenizers, die Chunk-Größe, die Überlappung und die jeweils verwendete Chunking-Strategie iterativ anzupassen.



Nutzen Sie automatisierte Tests und Monitoring-Tools, um die Leistung Ihrer Modelle kontinuierlich zu überwachen und dynamisch Anpassungen an den Chunking-Parametern vorzunehmen. Eine regelmäßige Evaluation ermöglicht es, auf Veränderungen im Input oder in den Anforderungen schnell zu reagieren.

5 Anhang 1: Checkliste Parameterwahl

- 1. Analyse der Dokumenteigenschaften
 - Dokumenttyp identifizieren:** Lange Texte, mittel-lange Texte, kurze Texte, Code/technische Dokumente.
 - Typische Eigenschaften erfassen:** Länge, Struktur, Informationsdichte, spezielle Formatierungen (z. B. Tabellen, Codeblöcke).
- 2. Auswahl des Tokenizers
 - Sprachliche und technische Anforderungen prüfen:**
 - Lange, unstrukturierte Texte: SentencePiece oder BPE
 - Gemischte Inhalte (technisch und allgemein): WordPiece
 - Stark strukturierte Daten oder Code: Whitespace-/Symbol-basierte Tokenizer oder angepasste Tokenizer
 - Spezifische Anforderungen an Subwort-Einheiten bewerten.**
- 3. Festlegung der Chunk-Größe
 - Ziel der Chunking-Einheit definieren:** Soll ein vollständiger Satz, Absatz oder logische Einheit abgebildet werden?
 - Dokumenttyp berücksichtigen:**
 - Lange Texte: 512–1024 Tokens
 - Mittel-lange Texte: 256–512 Tokens
 - Kurze Texte: 50–200 Tokens
 - Code/technische Dokumente: Abhängig von logischen Blöcken (z. B. pro Funktion)
 - Praktische Rahmenbedingungen einbeziehen:** Speicherverbrauch und Verarbeitungsgeschwindigkeit.
- 4. Definition der Überlappung
 - Ziel des Überlappungsgrades festlegen:** Sicherstellung des Kontext-Erhalts, ohne unnötige Redundanz.
 - Empfohlene Überlappungswerte anpassen:**
 - Hohe Überlappung (30–50%) für kontext-sensitive Aufgaben wie Q&A.
 - Geringere Überlappung (0–5% bis 10–20%) für Klassifikation oder strukturierte Daten.
- 5. Auswahl der konkreten Chunking-Strategie
 - Strategie zur Erhaltung semantischer Einheiten prüfen:**

- Semantisches Chunking, um zusammenhängende inhaltliche Blöcke zu bilden.
- Alternativen in Betracht ziehen, falls keine klaren Grenzen vorliegen:**
 - Rekursives Zeichen-Chucking.
- Spezialfälle für Retrieval und NER:**
 - Embeddingbasiertes Chunking, um semantisch verwandte Abschnitte zu gruppieren.
- Besondere Anforderungen bei Code:**
 - Agentisches Chunking, um logische und syntaktische Zusammenhänge zu berücksichtigen.
- **6. Evaluation und iterative Feinabstimmung**
 - Metriken definieren:** F1-Score, ROUGE, Recall@K usw.
 - Pilotphase durchführen:** Verschiedene Einstellungen testen und Ergebnisse vergleichen.
 - Parameter anpassen:** Basierend auf den Evaluierungsergebnissen systematisch justieren.
- **7. Monitoring und praktische Rahmenbedingungen**
 - Automatisierte Tests und Monitoring einrichten:** Zur kontinuierlichen Überwachung der Modellleistung.
 - Kosten und Ressourcenverbrauch im Blick behalten:** Speicher, Rechenleistung und Kosten optimieren.
 - Regelmäßige Überprüfung:** Auf Veränderungen im Input oder in den Anforderungen reagieren und Parameter entsprechend anpassen.

Diese Checkliste unterstützt Sie dabei, einen strukturierten Ansatz zu verfolgen, sodass alle relevanten Parameter und praktischen Rahmenbedingungen systematisch berücksichtigt werden.

6 Anhang 2: Vom Wort zur Zahl

Warum ist es sinnvoll bei der Verarbeitung von natürlicher Sprache Worte in Zahlen umzuwandeln.

Zahlen sind direkt maschinenlesbar

Computer arbeiten intern mit binären Zahlen (0 und 1). Jede Zahl lässt sich direkt als Bitfolge darstellen. Zum Beispiel:

- Die Zahl **5** ist im Binärsystem **101**.
- Diese Darstellung ist eindeutig, kompakt und sofort verwendbar.

Worte sind komplexe Zeichenfolgen

Worte müssen zunächst **kodiert** werden, bevor der Computer sie verarbeiten kann. Dafür nutzt man z. B.:

- **ASCII** oder **Unicode** zur Umwandlung von Buchstaben in Zahlen.
- „Hallo“ → [72, 97, 108, 108, 111] (im ASCII-Code)

Worte haben Mehrdeutigkeit

Sprache ist für Menschen gemacht – sie ist:

- **mehrdeutig** („Bank“ = Sitzmöbel oder Geldinstitut)
- **kontextabhängig**
- **grammatisch komplex** (Zeitformen, Fälle, Satzbau usw.)

Das macht die Verarbeitung von Sprache viel schwieriger als die von Zahlen, bei denen jede Zahl klar definiert ist.

Zahlen folgen klaren Regeln

Mit Zahlen kann der Computer sofort rechnen, vergleichen oder sortieren. Sie folgen mathematischen Gesetzen, die leicht implementiert sind.

M08b - Embeddings



Anwendung Generativer KI

Stand: 03.2025

Damit Künstliche Intelligenz (KI) sinnvoll mit Sprache, Bildern oder anderen Inhalten arbeiten kann, muss sie deren Bedeutung erfassen. Allerdings verarbeitet ein Computer keine Wörter oder Bilder direkt, sondern nur Zahlen. **Embeddings** sind eine Methode, um solche Inhalte als Zahlen zu kodieren, sodass die KI Zusammenhänge und Bedeutungen erkennen kann.

1 Was sind Embeddings?

Ein **Embedding** ist eine mathematische Darstellung eines Wortes, Satzes oder Bildes in Form eines Vektors, also einer Zahlenliste. Diese Zahlen erfassen Ähnlichkeiten und Zusammenhänge zwischen verschiedenen Konzepten.

Beispiel für Sprache:

- Das Wort „King“ könnte als Zahlenvektor **[0.96, 0.92, 0.08, 0.67]** dargestellt werden.
- Das Wort „Queen“ könnte **[0.98, 0.07, 0.93, 0.71]** haben.
- Das Wort „Girl“ könnte **[0.56, 0.09, 0.91, 0.11]** haben.

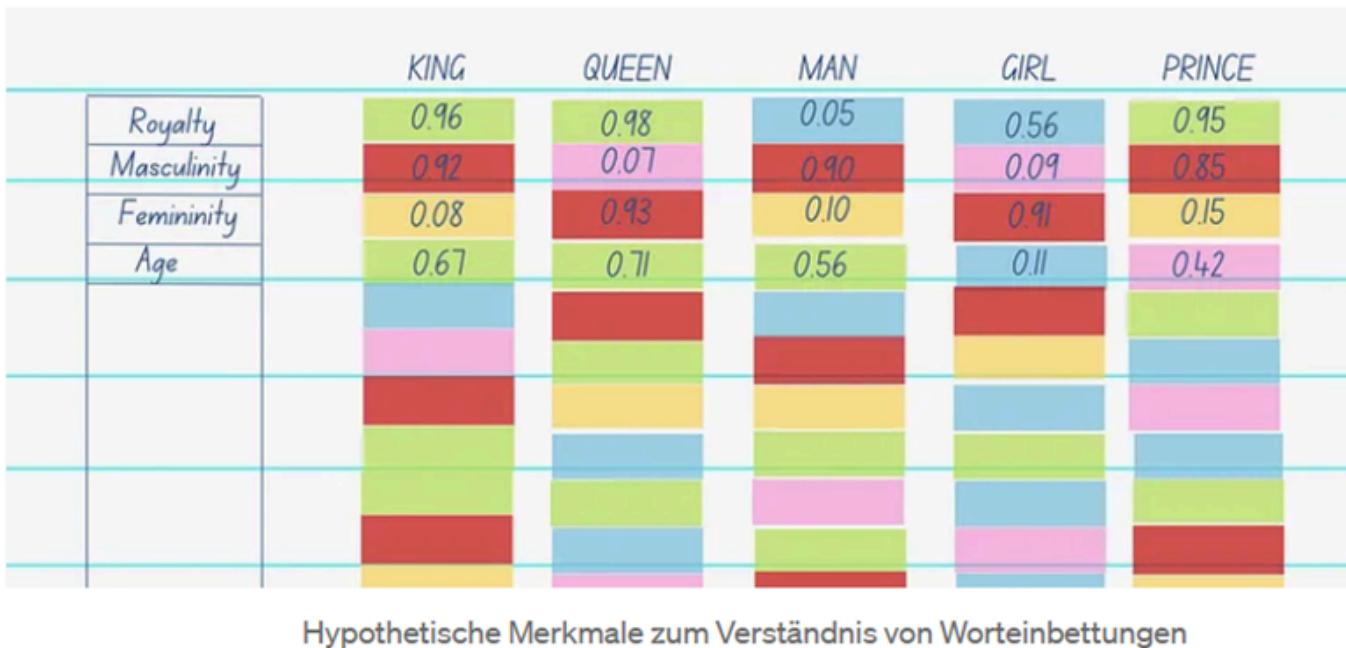
→ Die Zahlen von „King“ und „Queen“ sind **ähnlicher** als die von „Man“ und „Girl“. Dies zeigt, dass die KI die inhaltliche Nähe dieser Begriffe versteht.

Beispiel für Bilder:

- Ein Bild von einem Hund wird in Zahlen umgewandelt.
- Ein ähnliches Bild erhält einen ähnlichen Zahlenvektor.
- Dadurch kann die KI visuelle Ähnlichkeiten erkennen.

Embeddings werden nicht nur für Sprache und Bilder genutzt, sondern auch in Empfehlungssystemen für Musik, Filme oder sogar in der medizinischen Forschung zur Mustererkennung.

Hypothetisches Beispiel für die Embeddings:



Quelle: [Ein Leitfaden für Anfänger zu Word2Vec. Grundlagen von Word2Vec + Implementierung... | von Manan Suri | Medium](#)

2 Wie entstehen Embeddings?

Embeddings werden mit **künstlichen neuronalen Netzen** oder **statistischen Methoden** erzeugt. Dabei durchläuft der Prozess mehrere Schritte:

1. Daten sammeln

- Sprachmodelle nutzen große Mengen an Texten aus Büchern, Webseiten oder Artikeln.
- Bilderkennungsmodelle analysieren Millionen von Fotos mit passenden Beschreibungen.
- Musik- oder Videoplattformen sammeln Daten zu Nutzerverhalten und Inhaltsmerkmalen.

2. Daten in Zahlen umwandeln

- Wörter werden als **Vektoren** dargestellt, die Bedeutungsähnlichkeiten widerspiegeln.
- Bilder werden in **Pixelwerte** und Merkmale wie Kanten oder Farben umgerechnet.
- Musik wird anhand von Frequenzmustern und Rhythmen analysiert.

3. Neuronale Netze trainieren

- Modelle wie **Word2Vec**, **GloVe** oder **FastText** für Sprache sowie **ResNet** oder **VGG** für Bilder lernen, welche Begriffe oder Objekte ähnlich sind.
- Empfehlungssysteme analysieren, welche Songs oder Filme Nutzer häufig zusammen konsumieren.

4. Ähnlichkeiten erkennen

- Begriffe mit ähnlicher Bedeutung liegen im Zahlenraum nahe beieinander.
- Beispiel: Das Embedding für „König“ liegt näher an „Königin“ als an „Banane“.
- Bilder von Hunden liegen näher an Wölfen als an Autos.

5. Feinabstimmung (Fine-Tuning)

- Embeddings können für spezifische Anwendungen optimiert werden.
 - Beispiel: Eine KI für medizinische Analysen trainiert spezielle Embeddings für Fachbegriffe.
 - Streaming-Dienste passen ihre Embeddings an individuelle Nutzerpräferenzen an.
-

3 Positional Encoding

Die Positions kodierung fügt jedem Token-Vektor (aus der Einbettungsmatrix) Informationen über seine Position in der Sequenz hinzu. Dies geschieht durch die Kombination von Positionsinformationen und den ursprünglichen Token-Einbettungen. Ohne zusätzliche Information gäbe es keinen Unterschied zwischen:

- *Die Katze jagt den Hund* und
- *Den Hund jagt die Katze*

Die Positions kodierung ist wie ein kleiner Hinweiszettel, der sagt, welches Wort an welcher Stelle steht.

Positionskodierung im maschinellen Verständnis

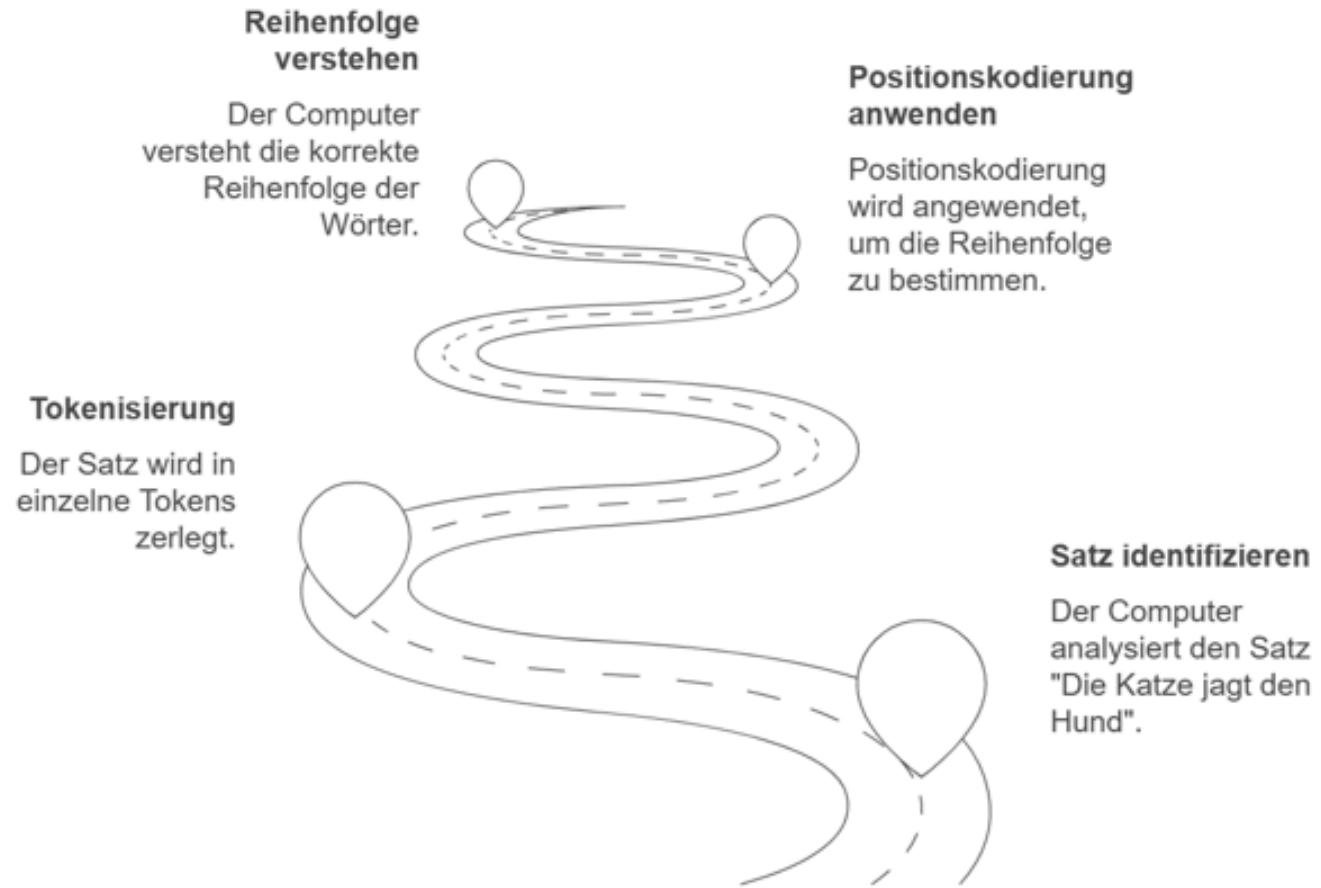


Bild mit napkin.ai erstellt

4 Embedding-Modelle

Es gibt verschiedene Einbettungsmodelle wie Word2Vec, GloVe und FastText für Wortrepräsentationen, BERT für kontextuelle Einbettungen sowie Node2Vec und LSTM-basierte Modelle für Netzwerke und Sequenzen, die jeweils auf spezifische Anwendungsfälle und Datenstrukturen optimiert sind.

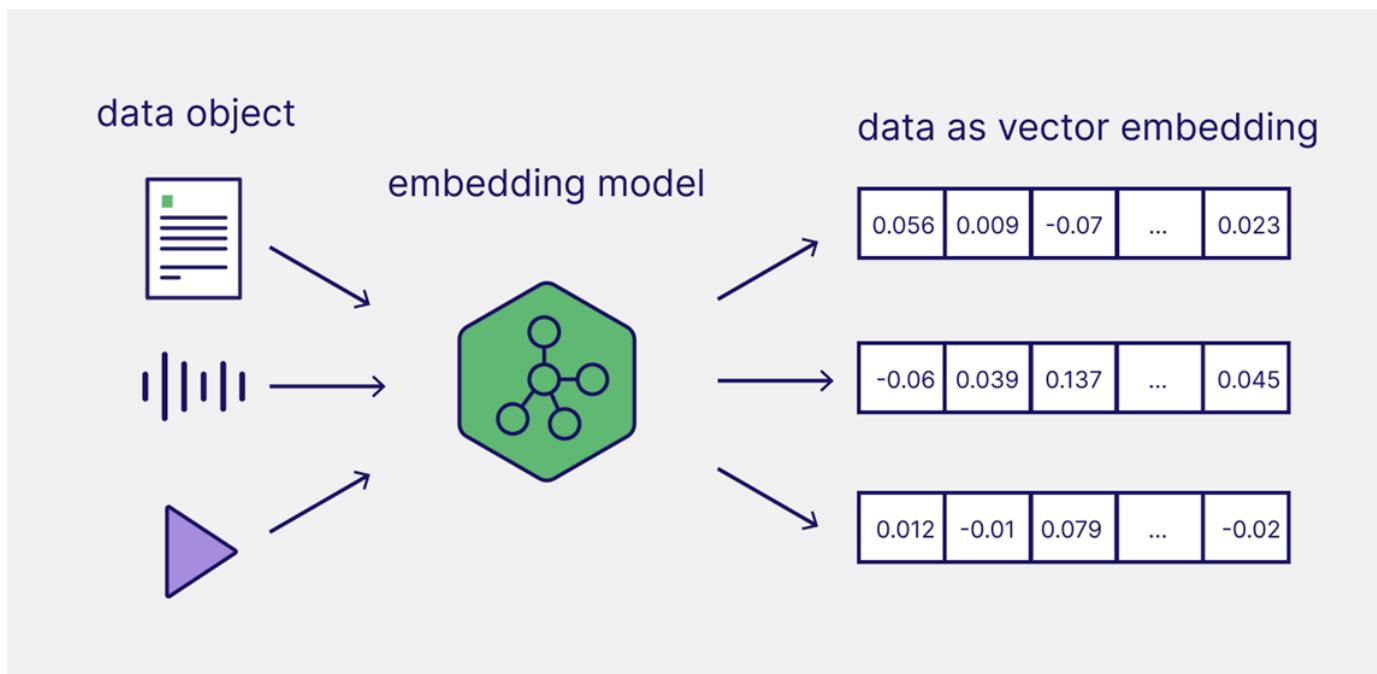


Bild: [Step-by-Step Guide to Choosing the Best Embedding Model for Your Application](#) | [Weaviate - Vector Database](#)

Übersicht Einbettungsmodelle:

Einbettungsvektor	Typische Größen	Einsatzbereich
Word2Vec	100-300 Dimensionen	Wort- und Satzhähnlichkeiten, NLP
GloVe	50, 100, 200, 300 Dimensionen	Semantische Wortbeziehungen, NLP
FastText	100-300 Dimensionen	OOV-Wortbehandlung, NLP
BERT (Basisversion)	768 Dimensionen	Kontextuelle Textverarbeitung, NLP
BERT (Large-Version)	1024 Dimensionen	Fortgeschrittene NLP-Anwendungen
text-embedding-ada-002 (OpenAI)	1536 Dimensionen	Hochqualitative semantische Suche, RAG
sentence-transformers	384-768 Dimensionen (modellabhängig)	Semantische Ähnlichkeit, Clustering, RAG
Benutzer- und Produkteinbettungen	50-200 Dimensionen	Empfehlungssysteme, Personalisierung
Einbettungen aus CNNs (VGG16)	4096 Dimensionen (für FC-Schichten)	Bildverarbeitung, Objekterkennung
Einbettungen aus CNNs (ResNet)	Variiert (tiefer mit unterschiedlichen Größen)	Bildanalyse, Feature-Extraktion
Node2Vec	64-256 Dimensionen	Graphenanalysen, soziale Netzwerke

Einbettungsvektor	Typische Größen	Einsatzbereich
LSTM-basierte Sequenzeinbettungen	50-500 Dimensionen	Zeitreihen, Sprachmodellierung, NLP

5 Training von Embedding-Modellen

Das Training von Embedding-Modellen wie Word2Vec basiert auf der Idee, dass Wörter, die in ähnlichen Kontexten vorkommen, ähnliche Bedeutungen haben. Hier wird detaillierter beschrieben, wie dieses Prinzip im Training umgesetzt wird:

Algorithmus-Auswahl

Word2Vec bietet zwei grundlegende Modelle zur Generierung von Wort-Embeddings:

1. **CBOW (Continuous Bag of Words)**: Hierbei wird das Zielwort basierend auf einem umgebenden Wortkontext vorhergesagt. Das Modell bekommt mehrere Wörter als Eingabe (den Kontext) und versucht, das Wort in der Mitte (das Zielwort) zu vorherzusagen.
2. **Skip-Gram**: Hier wird der umgekehrte Ansatz verfolgt. Ausgehend von einem Zielwort versucht das Modell, die umgebenden Kontextwörter vorherzusagen.

Training

Das Training von Word2Vec kann wie folgt zusammengefasst werden:

- **Initialisierung**: Zuerst werden Vektoren für jedes Wort zufällig initialisiert.
- **Durchlauf durch den Korpus**: Das Modell geht durch den gesamten Textkorpus, nimmt jedes Wort zusammen mit seinen Nachbarwörtern (innerhalb eines bestimmten Fensters) und führt Trainingsiterationen durch.
- **Verlustfunktion**: Die Hauptaufgabe beim Training ist die Optimierung der Verlustfunktion. Für CBOW und Skip-Gram wird oft eine Funktion verwendet, die die logarithmische Wahrscheinlichkeit maximiert, korrekte Wörter basierend auf ihren Kontexten vorherzusagen.
 - Bei **CBOW** wird der Verlust berechnet, indem die Differenz zwischen dem vorhergesagten Zielwort und dem tatsächlichen Zielwort über die Softmax-Funktion gemessen wird.
 - Beim **Skip-Gram** wird der Verlust für jedes vorhergesagte Kontextwort berechnet.
- **Backpropagation**: Mit Hilfe des Gradientenabstiegs oder ähnlicher Optimierungsalgorithmen werden die Gewichte (Wortvektoren) so angepasst, dass die Verlustfunktion minimiert wird. Dies bedeutet, dass die Wortvektoren nach und nach angepasst werden, um den wahren Kontext besser widerzuspiegeln.

Ergebnis

Das Ergebnis des Trainings ist ein Set von Vektoren, eines für jedes Wort im Vokabular. Wörter, die in ähnlichen Kontexten vorkommen, enden nahe beieinander im Vektorraum, was ihre semantische Ähnlichkeit widerspiegelt. Diese Vektoren können dann in verschiedenen nachgelagerten maschinellen Lernaufgaben verwendet werden, z.B. in der Sentiment-Analyse, bei der Klassifikation von Dokumenten oder anderen NLP-Aufgaben, die eine numerische Repräsentation von Text erfordern.

Evaluierung

Um die Qualität der Embeddings zu überprüfen, werden oft qualitative Tests wie die Suche nach den nächsten Nachbarn (ähnliche Wörter finden) oder quantitative Benchmarks (z.B. auf Datensätzen für analoge Aufgaben) durchgeführt. Diese Evaluierungen helfen dabei festzustellen, ob das Modell die Wortbedeutungen effektiv erfasst hat.

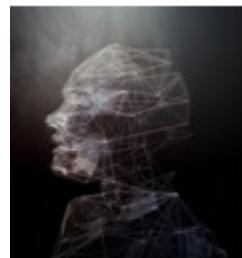
6 Warum sind Embeddings so wichtig?

- **Sprachverarbeitung:** Chatbots, Übersetzungen und Textanalysen basieren auf Embeddings.
- **Bilderkennung:** KI kann ähnliche Bilder oder Objekte erkennen.
- **Suche & Empfehlungssysteme:** Personalisierte Vorschläge auf Plattformen wie Netflix, Spotify oder YouTube nutzen Embeddings.
- **Musik- und Videovorschläge:** Streaming-Dienste berechnen Nutzerpräferenzen basierend auf Embeddings.
- **Medizinische Diagnosen:** KI analysiert Krankheitsbilder und medizinische Muster durch Embeddings.
- **Generative KI:** Sprachmodelle wie ChatGPT nutzen Embeddings, um kontextbezogene Antworten zu generieren.

Fazit

Embeddings sind ein zentrales Konzept in der modernen KI. Sie ermöglichen Maschinen, Bedeutungen zu erfassen, Muster zu erkennen und personalisierte Inhalte zu liefern. Ohne Embeddings wären viele heutige KI-Technologien nicht denkbar – von Chatbots über Bilderkennung bis hin zu Streaming-Diensten. Sie sind das **unsichtbare Gerüst**, das intelligente Systeme erst möglich macht.

M08c - Vektordatenbanken



Anwendung Generativer KI

Stand: 02.2025

Hier ist eine vergleichende Übersicht zu **Vektordatenbanken**, die häufig im Kontext von KI-Anwendungen, insbesondere für die Speicherung und Abfrage von Embeddings (Vektoren), verwendet werden:

1 Übersicht

Aspekt	Pinecone	Weaviate	Milvus	FAISS (Facebook AI Similarity Search)	Chroma
Definition	Cloud-native Vektordatenbank für Echtzeit-Suche und Empfehlungssysteme.	Open-Source-Vektordatenbank mit integrierter NLP- und KI-Funktionalität.	Open-Source-Vektordatenbank für skalierbare Vektorähnlichkeitssuche.	Bibliothek für effiziente Ähnlichkeitssuche in großen Vektordatensätzen.	Open-Source-Vektordatenbank für KI-Anwendungen, einfach zu integrieren.

Aspekt	Pinecone	Weaviate	Milvus	FAISS (Facebook AI Similarity Search)	Chroma
Zweck	Echtzeit-Suche und Empfehlungssysteme.	Kombiniert Vektorsuche mit strukturierten Daten und NLP-Funktionen.	Skalierbare Vektorähnlichkeitssuche für große Datensätze.	Effiziente Ähnlichkeitssuche in großen Vektordatensätzen.	Einfache Integration von Vektorsuche in KI-Anwendungen.
Modellunterstützung	Unterstützt verschiedene Embedding-Modelle.	Unterstützt verschiedene Embedding-Modelle und NLP-Modelle.	Unterstützt verschiedene Embedding-Modelle.	Unterstützt verschiedene Embedding-Modelle.	Unterstützt verschiedene Embedding-Modelle.
Flexibilität	Hoch, da cloud-nativ und für Echtzeit-Anwendungen optimiert.	Hoch, durch Kombination von Vektorsuche und strukturierten Daten.	Hoch, durch Skalierbarkeit und Unterstützung großer Datensätze.	Hoch, als Bibliothek in bestehende Anwendungen integrierbar.	Hoch, durch einfache Integration und Nutzung.
Anwendungsfälle	Empfehlungssysteme, personalisierte Suche, Echtzeit-Analyse.	KI-gestützte Suche, Wissensgraphen, NLP-Anwendungen.	Skalierbare Ähnlichkeitssuche, Bild- und Textsuche.	Effiziente Suche in großen Vektordatensätzen, Forschung und Entwicklung.	KI-Anwendungen, semantische Suche, Chatbots.
Community & Support	Kommerzieller Support, wachsende Community.	Aktive Open-Source-Community, kommerzielle Unterstützung verfügbar.	Große Open-Source-Community, kommerzielle Unterstützung verfügbar.	Große Community, da von Facebook entwickelt, aber keine kommerzielle Unterstützung.	Wachsende Open-Source-Community.

Aspekt	Pinecone	Weaviate	Milvus	FAISS (Facebook AI Similarity Search)	Chroma
Integration	Einfache Integration mit Cloud-Diensten und KI-Frameworks.	Integration mit NLP-Tools, KI-Frameworks und Datenbanken.	Integration mit KI-Frameworks und Big-Data-Tools.	Integration in bestehende Anwendungen als Bibliothek.	Einfache Integration mit KI-Frameworks wie LangChain.
Open Source	Nein	Ja	Ja	Ja	Ja
Lernkurve	Mittel, aufgrund der Cloud-Integration und Echtzeit-Funktionen.	Mittel, durch Kombination von Vektorsuche und strukturierten Daten.	Mittel bis hoch, aufgrund der Skalierbarkeit und Komplexität.	Mittel, als Bibliothek mit Fokus auf Effizienz.	Niedrig bis mittel, dank einfacher Integration und Nutzung.
Beispiele	Personalisierte Produktempfehlungen, Echtzeit-Suche.	KI-gestützte Wissensgraphen, semantische Suche.	Bild- und Textsuche in großen Datensätzen.	Forschung und Entwicklung, effiziente Suche in großen Datensätzen.	Semantische Suche in Chatbots, KI-gestützte Anwendungen.

2 Zusammenfassung:

- **Pinecone**: Ideal für Echtzeit-Anwendungen und kommerzielle Nutzung, aber nicht Open Source.
- **Weaviate**: Kombiniert Vektorsuche mit strukturierten Daten und NLP, gut für KI-gestützte Anwendungen.
- **Milvus**: Skalierbare Lösung für große Datensätze, geeignet für komplexe Anwendungen.
- **FAISS**: Effiziente Bibliothek für die Suche in großen Vektordatensätzen, ideal für Forschung und Entwicklung.
- **Chroma**: Einfache Integration und Nutzung, ideal für KI-Anwendungen und Einsteiger.

Die Wahl der Vektordatenbank hängt von Ihren spezifischen Anforderungen ab:

- **Pinecone** für Echtzeit-Anwendungen und kommerzielle Nutzung.
- **Weaviate** für KI-gestützte Anwendungen mit strukturierten Daten.
- **Milvus** für skalierbare Lösungen mit großen Datensätzen.
- **FAISS** für effiziente Suche in der Forschung.
- **Chroma** für einfache Integration und KI-Anwendungen.

Fazit

Chroma überzeugt durch seine einfache Integration, intuitive Python-API und geringe Einstiegshürden, was es zur idealen Vektordatenbank für Einsteiger und schnelle Prototypen macht. Als Open-Source-Lösung bietet es zudem Kostenvorteile und Flexibilität, während es gleichzeitig leistungsfähig genug für die meisten KI-Anwendungsfälle bleibt.

M09 - Multimodal Bild

Stand: 03.2025

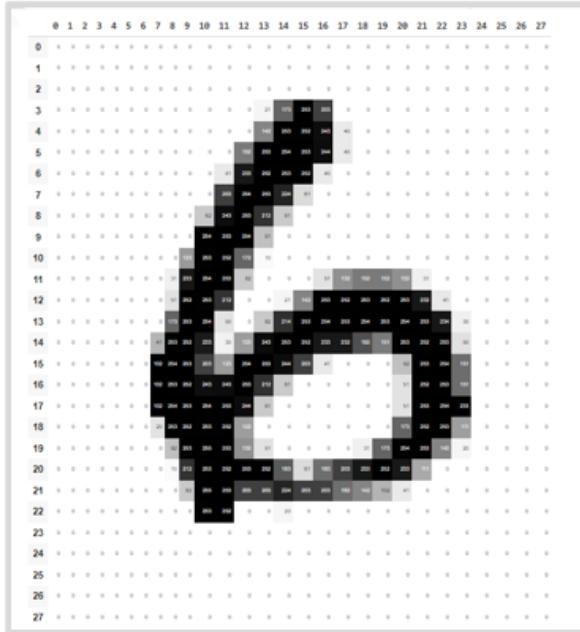
1 Grundlagen Bilderkennung

Ein Computer "sieht" ein Bild nicht wie ein Mensch. Für ihn stellt ein Bild lediglich eine Matrix aus Zahlenwerten dar, wobei jeder Wert die Intensität eines Pixels in verschiedenen Farbkanälen (meist Rot, Grün, Blau) repräsentiert. Diese Zahlenwerte werden von Algorithmen verarbeitet, um Muster und Strukturen zu erkennen, die für die Bilderkennung essenziell sind.

Der Prozess der Bilderkennung umfasst folgende Schritte:

1. **Bilderfassung:** Ein Bild wird in eine numerische Darstellung umgewandelt. Dabei werden Bildformate wie JPEG oder PNG in ein Raster aus Pixelwerten umgerechnet.
2. **Vorverarbeitung:** Das Bild wird normalisiert, skaliert und aufbereitet. Dies kann Schritte wie Rauschunterdrückung, Kontrastanpassung oder Farbnormalisierung umfassen.
3. **Merkmalsextraktion:** Wichtige Merkmale wie Kanten, Formen oder Farbverteilungen werden identifiziert. Hier können Methoden wie Histogramm-basierte Ansätze oder Kantendetektion (z. B. Sobel-Operator) angewendet werden.
4. **Klassifikation:** Basierend auf den extrahierten Merkmalen erfolgt eine Klassifikation des Bildes. Dies geschieht mithilfe von maschinellen Lernmodellen oder regelbasierten Systemen.

Raster aus Pixelwerten



2 Methoden

2.1 Traditionelle Methoden

Bei traditionellen Methoden müssen explizit Merkmale definiert werden, die als relevant gelten (z. B. Kanten, Farben, Texturen). Klassische Verfahren beinhalten Methoden wie:

- **Kantendetektion** mittels Sobel- oder Canny-Operator
- **Merkmalsvektoren** wie SIFT (Scale-Invariant Feature Transform) oder HOG (Histogram of Oriented Gradients)
- **Template Matching**, um spezifische Muster in Bildern zu finden

Diese Verfahren erfordern umfassendes domänenspezifisches Wissen und sind oft anfällig für Variationen in Beleuchtung, Perspektive oder Bildrauschen.

Merkmals-Filter: <https://editor.p5js.org/ralf.bendig.rb/full/zLXqj5u6f>

Merkmals-Filter-Anwendung: <https://editor.p5js.org/ralf.bendig.rb/full/Xi2uabjR9>

2.2 Deep Learning

Moderne Ansätze setzen auf neuronale Netze, insbesondere Convolutional Neural Networks (CNNs), die eigenständig lernen, welche Merkmale relevant sind. CNNs bestehen aus mehreren Schichten, die folgende Aufgaben erfüllen:

- **Faltungsschichten (Convolutional Layers)**: Extrahieren Merkmale durch das Anwenden von Filtern
- **Pooling-Schichten**: Reduzieren die Dimensionen und verallgemeinern die Merkmale

- **Voll verbundene Schichten (Fully Connected Layers):** Nutzen die extrahierten Merkmale zur Klassifikation

Deep-Learning-Modelle werden auf große Datensätze trainiert, wodurch sie eine hohe Generalisierungsfähigkeit erreichen und in der Lage sind, komplexe Muster autonom zu lernen.

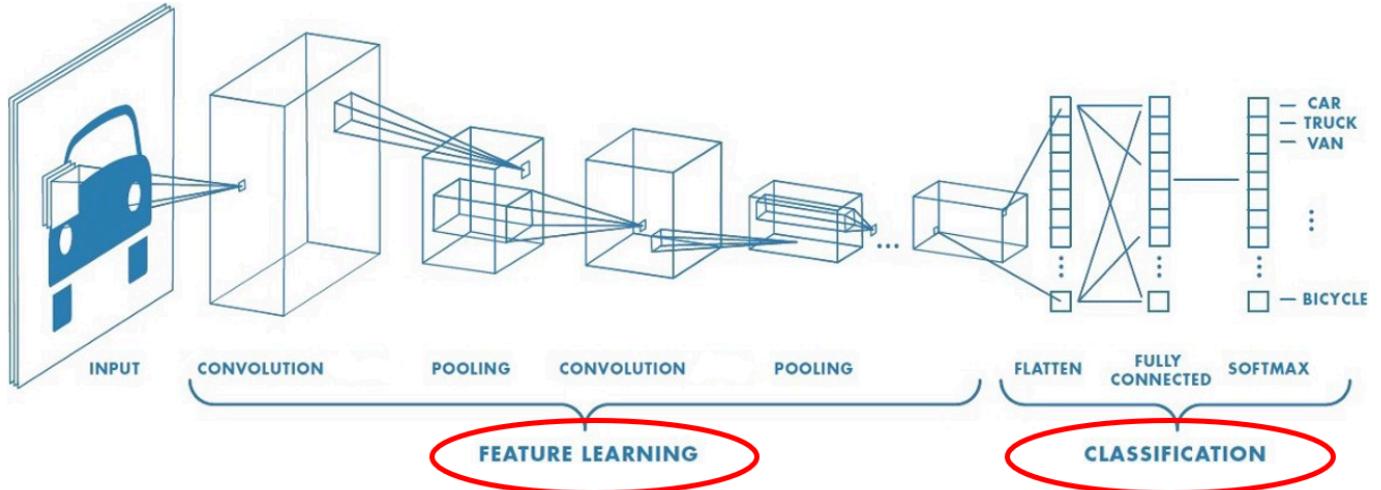


Bild: [A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | by Sumit Saha | Towards Data Science](#)

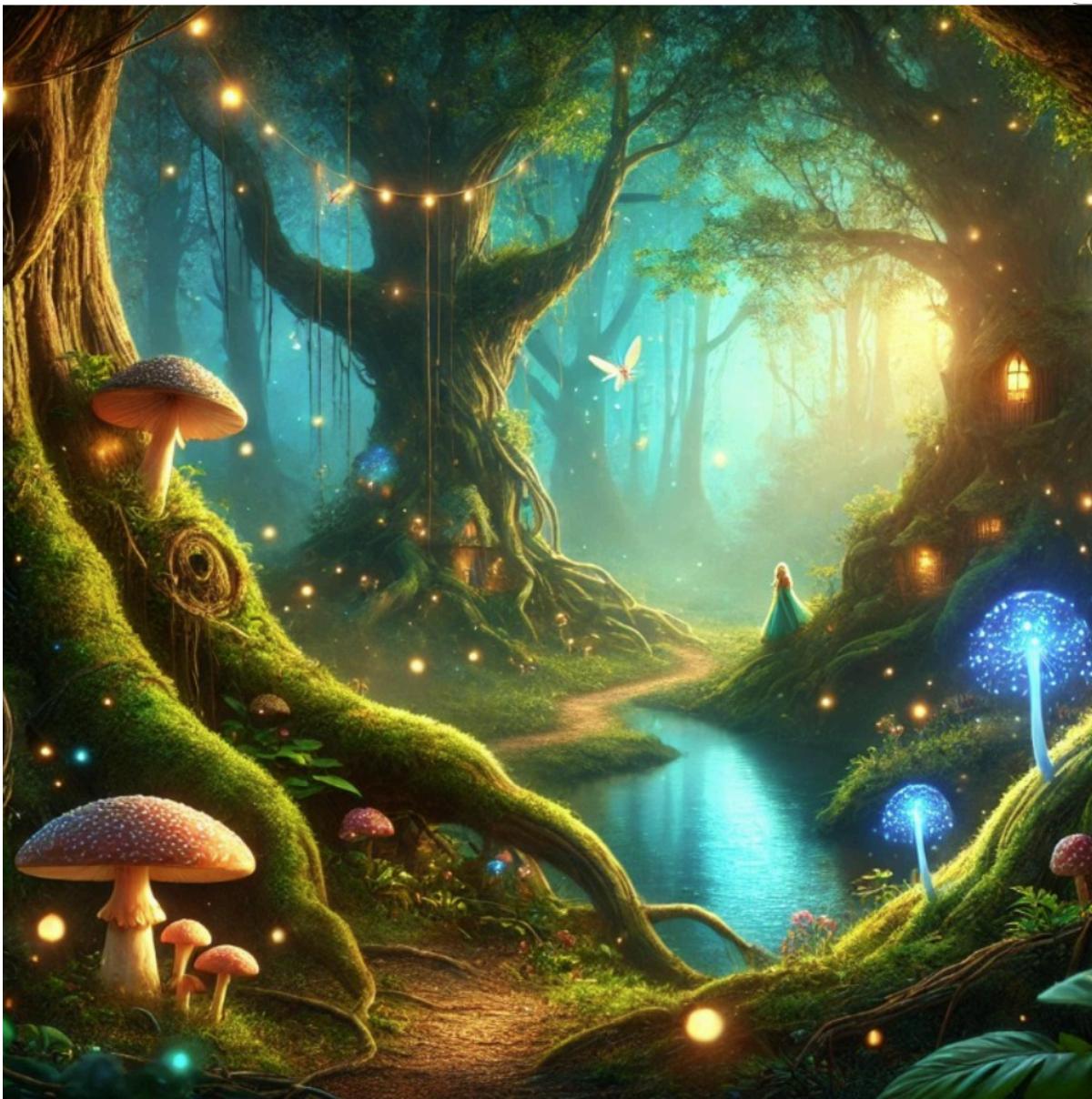
3 Bild-Modelle

3.1 Text-zu-Bild-Modelle

In diesem Abschnitt wird der Fokus auf Text-zu-Bild-Modelle gelegt, einem innovativen Bereich der künstlichen Intelligenz, der es Maschinen ermöglicht, Bilder basierend auf Textbeschreibungen zu erzeugen. Diese Modelle überbrücken die Kluft zwischen Sprache und visuellen Inhalten, indem sie eine natürliche Spracheingabe in eine vollständige Bilddarstellung umsetzen. Die Generierung von Bildern aus Text basiert auf Deep-Learning-Methoden, insbesondere durch die Kombination von natürlicher Sprachverarbeitung (NLP) und Computer Vision.

Ein zentrales Merkmal dieser Modelle ist ihre Fähigkeit, Zusammenhänge zwischen sprachlichen und visuellen Elementen zu erfassen. Durch das Training mit umfangreichen Datensätzen, die Texte mit den dazugehörigen Bildern verknüpfen, lernen sie, sprachliche Beschreibungen wie „eine Katze sitzt auf einem Fensterbrett“ mit relevanten visuellen Eigenschaften wie Formen, Texturen, Farben und räumlichen Anordnungen zu verbinden. Modelle wie DALL·E, MidJourney und Stable Diffusion haben das kreative Potenzial dieser Technologie eindrucksvoll demonstriert, indem sie sowohl fotorealistische als auch künstlerische Bilder direkt aus textlichen Vorgaben generiert haben.

Hier wird veranschaulicht, wie DALL·E ein Bild mit einem zauberhaften Märchenwald generiert:



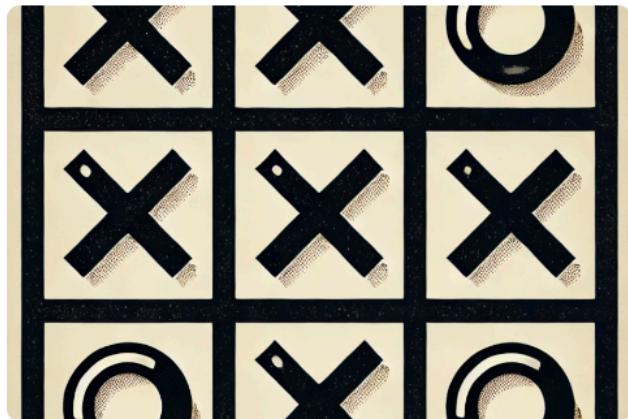
3.2 Multimodale Modelle

In diesem Abschnitt wird die spannende Welt multimodaler Modelle untersucht, die verschiedene Datentypen wie Text, Bilder, Audio und Video verarbeiten und zu einem ganzheitlichen Verständnis verknüpfen. Diese Modelle markieren einen bedeutenden Fortschritt in der KI, da sie Eingaben aus unterschiedlichen Quellen kombinieren und dadurch komplexere Aufgaben bewältigen können – ähnlich wie der Mensch Informationen aus verschiedenen Sinneseindrücken zusammenführt.

Während spezialisierte Modelle, etwa Text-zu-Bild-Systeme, bereits einzelne Verbindungen zwischen Modalitäten ermöglichen, gehen multimodale Modelle einen Schritt weiter. Sie analysieren und verstehen die Beziehungen zwischen verschiedenen Informationsarten, was vielfältige Anwendungen ermöglicht. Dazu zählen das Generieren von Bildunterschriften, das Beantworten visueller Fragen oder die Erstellung von Bildern auf Basis von Text und anderen Medieneingaben.

Ein anschauliches Beispiel für multimodale Fähigkeiten ist die Analyse eines handgezeichneten Tic-Tac-Toe-Bretts. Ein solches Modell kann ein Bild des Spiels

interpretieren, die Platzierung von X und O erkennen und basierend auf den Spielregeln den Gewinner bestimmen – ohne zusätzliche textliche Informationen über das Spielfeld zu benötigen.



Wer hat das Spiel gewonnen?

Das Bild zeigt ein Tic-Tac-Toe-Spielbrett mit einem 3x3-Raster. Die meisten Felder sind mit "X" markiert, während drei Felder ein "O" enthalten.

Um den Gewinner zu bestimmen, prüfen wir, ob eine der Parteien drei ihrer Symbole in einer Reihe, Spalte oder Diagonale angeordnet hat:

- Horizontale Reihen: Alle Reihen enthalten gemischte Symbole oder nur "X".
- Vertikale Spalten: Keine Spalte enthält ausschließlich "O".
- Diagonale Linien: Keine vollständige Diagonale besteht nur aus "O".

Da in keiner Reihe, Spalte oder Diagonale drei "O" erscheinen, aber eine horizontale Reihe oder Spalte vollständig mit "X" belegt ist, hat "X" das Spiel gewonnen.



Durch die Kombination verschiedener Datentypen ermöglichen multimodale Modelle eine fortschrittlichere und intuitivere Interaktion zwischen Mensch und KI. Dies macht sie zu einem vielseitigen Werkzeug mit breiten Anwendungsmöglichkeiten – von Gesundheitswesen und Bildung bis hin zu Unterhaltung und Design.

4 Image-Embeddings

Ähnlich wie bei Text-Embeddings, die Wörter oder Sätze in einer Weise kodieren, dass semantische Ähnlichkeiten erhalten bleiben, transformieren Image-Embeddings visuelle Merkmale in eine für Maschinen lernbare Form.

Mithilfe neuronaler Netze – typischerweise Convolutional Neural Networks (CNNs) oder Transformer-Modelle wie CLIP – werden hochdimensionale Bilddaten in kompakte Vektoren

umgewandelt. Diese Embeddings ermöglichen Aufgaben wie Bildähnlichkeitssuche, Clustering oder die Kombination von Bild- und Textdaten für multimodale Modelle.

Image-Embeddings

M14 - Multimodel Audio



Anwendung Generativer KI

Stand: 05.2025

1 Technische Grundlagen

Bevor wir in die praktische Anwendung von Audio-KI eintauchen, ist es wichtig, die grundlegenden technischen Konzepte zu verstehen, die hinter diesen Modellen stehen.

1.1 Von der Schallwelle zum digitalen Signal

Audio ist physikalisch betrachtet eine Schallwelle, die durch Druckschwankungen in der Luft entsteht. Um mit Computern verarbeitet zu werden, muss dieser analoge Schall in ein digitales Signal umgewandelt werden:

1. **Sampling (Abtastung)**: Der kontinuierliche Schall wird in regelmäßigen Zeitabständen gemessen. Die **Abtastrate** (Sampling Rate) gibt an, wie viele Messungen pro Sekunde durchgeführt werden. CD-Qualität verwendet z.B. 44.100 Messungen pro Sekunde (44,1 kHz).
2. **Quantisierung**: Jeder gemessene Wert wird in eine Zahl umgewandelt. Die **Bitttiefe** bestimmt, wie genau diese Umwandlung ist. 16-Bit-Audio kann 65.536 verschiedene Lautstärkewerte darstellen.

[Audio_Viz](#)

[MediaPipe](#)

1.2 Wie funktionieren Audio-KI-Modelle?

1.3 Speech-to-Text (Whisper)

OpenAI's Whisper nutzt eine **Encoder-Decoder-Architektur** mit Transformer-Technologie:

- Der **Encoder** wandelt das Audiosignal in eine kompakte Repräsentation um
- Der **Decoder** übersetzt diese Repräsentation in Text

Whisper wurde mit über 680.000 Stunden mehrsprachiger Audiodaten trainiert, wodurch es verschiedene Sprachen, Akzente und Umgebungsgeräusche verarbeiten kann.

1.4 Text-to-Speech (TTS-1)

TTS-1 verwendet ebenfalls eine komplexe neuronale Netzwerkarchitektur:

1. **Text-Encoder:** Wandelt Text in linguistische Merkmale um
2. **Prosody-Predictor:** Bestimmt Betonung, Rhythmus und Melodie
3. **Vocoder:** Erzeugt aus diesen Informationen naturgetreue Sprachsignale

Diese Komponenten arbeiten zusammen, um Text in natürlich klingende Sprache umzuwandeln, die Emotionen und Betonungen enthält.

1.5 Von der Audiowelle zum Verständnis

Wie "verstehen" KI-Modelle Audioinhalte? Der Prozess umfasst mehrere Schritte:

1. **Feature-Extraktion:** Aus dem Audiosignal werden charakteristische Merkmale extrahiert, z.B. durch Spektrogramme (visuelle Darstellungen der Frequenzanteile über Zeit)
2. **Musterkennung:** Neuronale Netze erkennen Muster in diesen Merkmalen
3. **Kontext-Analyse:** Durch Aufmerksamkeitsmechanismen wird der Kontext berücksichtigt
4. **Ausgabe-Generierung:** Erzeugung der Transkription oder der synthetisierten Sprache

Diese technischen Grundlagen erklären, warum moderne Audio-KI-Modelle so leistungsfähig sind und warum sie in der Lage sind, auch komplexe Audioinhalte zu verarbeiten und zu generieren.

2 Herausforderungen und Grenzen

Obwohl moderne Audio-KI-Systeme beeindruckende Ergebnisse erzielen, stoßen sie in bestimmten Situationen an ihre Grenzen. Diese Herausforderungen zu verstehen ist wichtig, um realistische Erwartungen zu setzen und die Qualität der Ergebnisse zu verbessern.

2.1 Herausforderungen bei Speech-to-Text (STT)

2.2 Sprachvariationen

- **Akzente und Dialekte:** Regionale Sprachvarianten können die Erkennungsgenauigkeit erheblich beeinflussen
- **Sprechgeschwindigkeit:** Sehr schnelles oder langsames Sprechen erschwert die korrekte Erkennung
- **Umgangssprache und Slang:** Informelle Ausdrücke werden oft nicht korrekt erkannt

2.3 Umgebungs faktoren

- **Hintergrundgeräusche:** Lärm, Musik oder andere Gespräche können die Qualität der Transkription beeinträchtigen
- **Halleffekte:** In halligen Räumen aufgenommene Sprache ist schwieriger zu transkribieren
- **Mikrofonqualität:** Niedrige Aufnahmegerätqualität führt zu schlechteren Transkriptionsergebnissen

2.4 Inhaltliche Komplexität

- **Fachbegriffe:** Spezialisierte Terminologie wird oft falsch transkribiert
- **Eigennamen:** Ungewöhnliche Namen werden häufig falsch erkannt
- **Homophone:** Wörter, die gleich klingen aber unterschiedlich geschrieben werden, führen zu Fehlern

2.5 Grenzen bei Text-to-Speech (TTS)

- **Emotionale Nuancen:** Subtile emotionale Ausdrücke sind schwer zu reproduzieren
- **Sprechpausen und Rhythmus:** Natürliches Sprechtempo ist eine Herausforderung
- **Aussprache seltener Wörter:** Ungewöhnliche oder fremdsprachige Begriffe werden oft falsch ausgesprochen
- **Kontextuelle Anpassung:** Die Anpassung an den inhaltlichen Kontext (z.B. Frage vs. Aussage) ist begrenzt

2.6 Strategie zur Verbesserung der Ergebnisse

2.6.1 Transkriptionen:

1. **Qualität der Aufnahme optimieren:** Ruhige Umgebung, gutes Mikrofon, angemessener Abstand zum Mikrofon
2. **Deutlich sprechen:** Gleichmäßiges Tempo, klare Aussprache
3. **Fachbegriffe bereitstellen:** Bei Bedarf eine Liste spezieller Begriffe vorbereiten

2.6.2 Sprachsynthese:

1. **Textformatierung anpassen:** Interpunktionszeichen für natürliche Pausen nutzen
2. **Aussprache-Hinweise:** Für schwierige Wörter phonetische Schreibweisen verwenden
3. **Stimme passend wählen:** Verschiedene Stimmen für unterschiedliche Inhalte

2.7 Ethische und praktische Grenzen

- **Stimmimitation:** Die Fähigkeit, Stimmen zu imitieren, wirft Fragen bezüglich Identitätsdiebstahl auf

- **Mehrsprachigkeit:** Die Qualität variiert stark zwischen verschiedenen Sprachen
- **Ressourcenverbrauch:** Hochwertige Audio-KI-Modelle benötigen erhebliche Rechenressourcen
- **Datenschutz:** Die Verarbeitung von Audiodaten erfordert besondere Sorgfalt im Umgang mit persönlichen Informationen

Das Bewusstsein für diese Herausforderungen hilft, Audio-KI-Technologien realistisch einzuschätzen und in geeigneten Kontexten effektiv einzusetzen.

3 Grundbegriffe für Einsteiger

Bevor wir uns mit der KI-basierten Audioverarbeitung beschäftigen, ist es wichtig, einige grundlegende Konzepte der digitalen Audioverarbeitung zu verstehen.

3.1 Was ist digitales Audio?

Audio besteht physikalisch aus Schallwellen – Druckschwankungen in der Luft, die unser Ohr wahrnimmt. Computer können jedoch nur mit digitalen Daten arbeiten, daher muss Schall für die Verarbeitung in Zahlen umgewandelt werden.

3.2 Der Digitalisierungsprozess

1. **Aufnahme:** Ein Mikrofon wandelt Schallwellen in elektrische Signale um
2. **Analog-Digital-Wandlung:** Diese kontinuierlichen Signale werden in diskrete Zahlenwerte umgewandelt
3. **Speicherung:** Die Zahlenwerte werden als Datei gespeichert
4. **Verarbeitung:** Die gespeicherten Werte können nun durch Programme verarbeitet werden

3.3 Wichtige Audio-Parameter

3.3.1 Abtastrate (Sampling Rate)

- **Definition:** Anzahl der Messungen pro Sekunde, gemessen in Hertz (Hz)
- **Typische Werte:**
 - 44.100 Hz (CD-Qualität)
 - 48.000 Hz (Professionelles Audio)
 - 8.000 Hz (Telefonie)
- **Auswirkung:** Höhere Abtastraten können höhere Frequenzen erfassen (gemäß dem Nyquist-Theorem)

3.3.2 Bittiefe (Bit Depth)

- **Definition:** Anzahl der Bits pro Abtastwert, bestimmt die Anzahl möglicher Lautstärkestufen
- **Typische Werte:**
 - 16 Bit (CD-Qualität, 65.536 Stufen)
 - 24 Bit (Professionelles Audio, über 16 Millionen Stufen)
- **Auswirkung:** Höhere Bittiefe verbessert den Dynamikumfang und reduziert Quantisierungsrauschen

3.3.3 Kanäle

- **Definition:** Anzahl der gleichzeitig aufgezeichneten Audiospuren
- **Typische Werte:**
 - Mono (1 Kanal)
 - Stereo (2 Kanäle)
 - Surround (5.1, 7.1, etc.)
- **Auswirkung:** Mehr Kanäle ermöglichen räumliches Audio

3.3.4 Audioformate

- **Unkomprimiert:** WAV, AIFF (verlustfreie Speicherung, große Dateien)
- **Komprimiert verlustbehaftet:** MP3, AAC, OGG (kleinere Dateien, etwas reduzierte Qualität)
- **Komprimiert verlustfrei:** FLAC, ALAC (reduzierte Dateigröße bei voller Qualität)

3.4 Audio-Eigenschaften

3.4.1 Amplitude

- **Definition:** Die Stärke des Audiosignals, entspricht der wahrgenommenen Lautstärke
- **Messung:** Dezibel (dB)

3.4.2 Frequenz

- **Definition:** Die Anzahl der Schwingungen pro Sekunde, bestimmt die Tonhöhe
- **Messung:** Hertz (Hz)
- **Menschliches Hören:** Etwa 20 Hz bis 20.000 Hz

3.4.3 Spektrum

- **Definition:** Die Verteilung der Energie über verschiedene Frequenzen
- **Darstellung:** Spektrogramm (Zeit-Frequenz-Darstellung)

3.5 Audioqualität und KI-Verarbeitung

Die Qualität der Audiodaten beeinflusst direkt die Ergebnisse der KI-Verarbeitung:

- **Hochwertige Aufnahmen:**
 - Klare Sprache ohne Hintergrundgeräusche
 - Angemessene Lautstärke (weder zu leise noch übersteuert)
 - Geeignete Abtastrate und Bitrate
- **Faktoren, die die Qualität beeinflussen:**
 - Mikrofonqualität und -platzierung
 - Akustik des Aufnahmeraums
 - Vermeidung von Übersteuerung und Verzerrung

4 Probleme & Hacks Audio-API

4.1 API-Fehler bei OpenAI

Symptome:

- Fehlermeldung: "Rate limit exceeded"
- Timeout-Fehler

Lösungsansätze:

```
import openai
import time
import backoff

# Exponential Backoff-Funktion für Wiederholungsversuche
@backoff.on_exception(backoff.expo,
                      (openai.RateLimitError, openai.APITimeoutError),
                      max_tries=5)
def transcribe_with_retry(file_path):
    with open(file_path, "rb") as audio_file:
        try:
            response = openai.audio.transcriptions.create(
                model="whisper-1",
                file=audio_file
            )
            return response.text
        except Exception as e:
            print(f"Fehler bei der Transkription: {e}")
            # Warten vor dem nächsten Versuch
            time.sleep(2)
            raise e
```

4.2 Unnatürliche Aussprache

Symptome:

- Falsche Betonung von Wörtern
- Abgehackte Sätze
- Falsche Aussprache von Fachbegriffen

Lösungen:

1. Interpunktionsanpassungen

- Kommas für kurze Pausen einfügen
- Punkte für längere Pausen verwenden

2. Aussprache-Hinweise verwenden

```
# Beispiel für Aussprache-Hinweise
text = "Der Patient leidet an Pneumonie (ausgesprochen: noi-mo-nie)."

# Alternative: Phonetische Schreibweise verwenden
text = "Python kann für verschiedene Aufgaben verwendet werden."
```

3. Satzstruktur vereinfachen

- Komplexe, verschachtelte Sätze in kürzere Sätze aufteilen

4.3 Fehlende Emotionalität

Lösungen:

1. Passende Stimme wählen

- Verschiedene Stimmen für unterschiedliche Stimmungen testen
- "Nova" für freundliche Inhalte, "Onyx" für ernstere Themen

2. Text mit Emotionshinweisen anreichern

```
# Emotionale Hinweise im Text
text = "Wow! Das ist eine fantastische Nachricht!" # Begeisterung

# Oder durch Beschreibungen
text = "[begeistert] Das ist eine fantastische Nachricht! [/begeistert]"
```

M16 - Fine-Tuning

Stand: 05.2025

Fine-Tuning ist eine bewährte Technik, um ein vortrainiertes KI-Modell gezielt auf spezifische Aufgaben oder Datensätze anzupassen. Dabei werden die bereits gelernten Strukturen und Muster eines bestehenden Modells genutzt und weiterverfeinert. Dieser Prozess spart nicht nur Rechenressourcen und Zeit, sondern führt auch bei kleineren Datenmengen zu beeindruckenden Ergebnissen.

In der Praxis bedeutet Fine-Tuning, dass Entwickler:innen lediglich einen Bruchteil der Rechenleistung und Datenmenge benötigen, die für das Training eines Modells von Grund auf erforderlich wären. Gleichzeitig wird die Möglichkeit geschaffen, eigene oder sensible Daten gezielt einzusetzen. Fine-Tuning ist somit nicht nur ein technisches Werkzeug, sondern auch ein strategischer Ansatz zur Effizienzsteigerung, Individualisierung und Qualitätssicherung von KI-Anwendungen.

Zudem ist Fine-Tuning ein zentraler Bestandteil des sogenannten Model Optimization Workflows. Dabei wird die Modellleistung durch eine Kombination aus **Evals**, **Prompt Engineering** und **Fine-Tuning** in einem iterativen Zyklus optimiert. Ziel ist es, die Qualität der Modellantworten durch Feedback und gezielte Anpassung kontinuierlich zu verbessern. Dieser sogenannte Optimierungs-Flywheel ermöglicht es, systematisch bessere Prompts, Trainingsdaten und daraus resultierende Modelle zu entwickeln.

1 Fine-Tuning-Ansätze

1.1 Transfer Learning

- **Grundprinzip:** Ein vortrainiertes Modell wird als Ausgangspunkt verwendet. Die allgemeinen Merkmale der frühen Schichten bleiben erhalten.
- **Vorgehen:** Die letzten Schichten werden ersetzt oder angepasst, um spezifische Aufgaben zu lösen.
- **Einsatzgebiete:** Bildklassifikation, Verarbeitung natürlicher Sprache (NLP), Computer Vision.
- **Vorteile:** Schneller Einstieg, geringer Datenbedarf, bewährte Basismodelle.
- **Vergleich zum Pre-Training:** Während das Pre-Training ein Modell mit großen Datenmengen (oft Billionen von Tokens) trainiert, um allgemeine Sprachmuster zu lernen, benötigt Fine-Tuning nur einen Bruchteil der Daten und Rechenkapazität für eine spezifische Aufgabe.

1.2 Parameter-effizientes Fine-Tuning (PEFT)

- **Prinzip:** Anpassung nur weniger Parameter, während das Basismodell unverändert bleibt.
- **Methoden:**
 - **LoRA (Low-Rank Adaptation):** Kompakte Matrizen für effiziente Gewichtsänderung. Reduziert den Rechenaufwand erheblich durch Low-Rank-Approximation der Gewichtsänderungen.
 - **QLoRA:** Eine Erweiterung von LoRA, die Gewichtsparameter auf 4-Bit-Präzision quantisiert, wodurch der Speicherbedarf weiter reduziert wird.
 - **DoRA (Weight-Decomposed Low-Rank Adaptation):** Zerlegt Gewichte in Größen- und Richtungskomponenten für präzisere Updates bei gleichbleibender Effizienz.
 - **Adapter:** Zusätzliche Module zwischen bestehenden Schichten.
 - **Prompt Tuning:** Anpassung durch trainierbare Prompts.
- **Vorteile:** Spart Ressourcen, wiederverwendbar, ideal für verschiedene Aufgaben.
- **Anwendungsbereich:** Besonders geeignet für ressourcenbeschränkte Umgebungen und für mehrere Spezialisierungsaufgaben mit demselben Basismodell.

1.3 Instruction Fine-Tuning

- **Ziel:** Das Modell lernt, auf klare, natürlichsprachliche Anweisungen zu reagieren.
- **Daten:** Input-Output-Paare mit expliziten Instruktionen.
- **Anwendungen:** Sprachassistenten, automatisierte Kommunikation, LLM-basierte Tools.
- **Formatbeispiel:** Oft in diesem Format: "##Human: < *InputQuery* > ##Assistant: < *GeneratedOutput* >"

1.4 Supervised Fine-Tuning (SFT)

- **Ansatz:** Feinabstimmung mit handverlesenen, hochwertigen Beispielen.
- **Zweck:** Optimierung für bestimmte Anforderungen oder Zielgruppen.
- **Typisch kombiniert mit:** Reinforcement Learning from Human Feedback (RLHF).
- **Datenerfordernisse:** Benötigt mindestens 10 Beispiele, empfohlen sind 50-100 qualitativ hochwertige Demonstrationen. Die Qualität der Daten ist entscheidender als die Menge.
- **Prozess:** Besteht aus Datenvorbereitung, Upload der Trainingsdaten, Erstellung eines Fine-Tuning-Jobs und anschließender Evaluierung des Modells.

2 Weitere Ansätze OpenAI

2.1 Direct Preference Optimization (DPO)

- **Vorgehen:** Training mit präferierten und abgelehnten Antwortpaaren.
- **Nutzen:** Verfeinert Nuancen wie Stil, Tonalität, Ausdruck.
- **Funktionsweise:** Ein effizienterer Weg als RLHF, um Modelle an menschliche Präferenzen anzupassen. Jedes Beispiel im Datensatz enthält einen Prompt, eine bevorzugte Ausgabe und eine nicht-bevorzugte Ausgabe.
- **Beta-Parameter:** Kann zwischen 0 und 2 konfiguriert werden, um zu steuern, wie streng das neue Modell am vorherigen Verhalten festhält versus sich an den neuen Präferenzen orientiert.

2.2 Reinforcement Fine-Tuning (RFT)

- **Prinzip:** Modell wird nicht mit festen Zielantworten, sondern anhand von Bewertungssignalen (Grader) trainiert.
- **Vorteile:** Besonders geeignet für komplexe, mehrdeutige Aufgaben.
- **Grader-Konzept:** RFT verwendet Grader, die die Modellantworten bewerten und ein numerisches Signal (zwischen 0 und 1) zurückgeben. Diese können als String-Check, Text-Similarity oder Model-Grader konfiguriert werden.
- **Anwendungsbereich:** Besonders effektiv bei Aufgaben, bei denen Experten in der Domäne sich über die richtigen Antworten einig sind und die Aufgabe eindeutig bewertbar ist.
- **Unterstützung:** Aktuell nur für reasoning-Modelle wie o4-mini verfügbar.

2.3 Vision Fine-Tuning

- **Zweck:** Anpassung von Modellen mit visuellen Eingaben (z. B. Bilder).
- **Anwendungen:** Bildklassifikation, visuelle Beschreibungen, Objektlokalisierung.
- **Technische Hinweise:** Unterstützt Base64-Bilder oder URLs, max. 10 Bilder pro Beispiel.
- **Besonderheiten:**
 - Bilder müssen im JPEG-, PNG- oder WEBP-Format vorliegen
 - Maximalgröße pro Bild: 10 MB
 - Bilder mit Menschen, Gesichtern, Kindern oder CAPTCHAs werden aus Datenschutzgründen ausgeschlossen
 - Der Detail-Parameter kann auf "low" gesetzt werden, um Trainingskosten zu reduzieren

2.4 Modell-Distillation

- **Konzept:** Nutzung der Ausgaben eines großen Modells, um ein kleineres Modell zu trainieren, das ähnliche Leistung für eine spezifische Aufgabe erzielt.
- **Vorteile:** Reduziert Kosten und Latenz erheblich, da kleinere Modelle effizienter sind.
- **Prozess:**

1. Speichern qualitativ hochwertiger Ausgaben eines großen Modells mit dem Parameter `store: true`
 2. Evaluierung der gespeicherten Antworten mit dem großen und kleinen Modell
 3. Auswahl relevanter Antworten als Trainingsdaten für das kleine Modell
 4. Fine-Tuning des kleinen Modells mit diesen Beispielen
 5. Evaluierung des fine-tuned kleineren Modells
- **Anwendungsbereich:** Besonders nützlich, wenn ein spezifischer, begrenzter Aufgabenbereich abgedeckt werden soll.

3 Fine-Tuning-Pipeline für LLMs

3.1 Datenvorbereitung

- Datensammlung aus verschiedenen Quellen
- Vorverarbeitung und Formatierung (z.B. JSONL-Format)
- Umgang mit unausgeglichenen Daten (Oversampling, Undersampling)
- Datensatzaufteilung (Training/Validierung/Test)

3.2 Modellinitialisierung

- Auswahl eines geeigneten vortrainierten Modells
- Einrichtung der Umgebung und Installation der Abhängigkeiten
- Laden des Modells in den Speicher

3.3 Trainingsumgebung

- Konfiguration von Hardwareressourcen (GPU/TPU)
- Definition von Hyperparametern (Lernrate, Batch-Größe, Epochen)
- Initialisierung von Optimierern und Verlustfunktionen

3.4 Fine-Tuning-Prozess

- Auswahl der Fine-Tuning-Technik (Voll, PEFT, etc.)
- Durchführung des Trainings mit regelmäßigen Validierungen
- Überwachung von Metriken und Verlustfunktionen

3.5 Evaluierung und Validierung

- Aufsetzen von Evaluierungsmetriken
- Analyse der Trainingsverlaufskurve
- Überwachung und Interpretation der Ergebnisse

3.6 Deployment

- Export des fine-tuned Modells
- Einrichtung der Infrastruktur
- API-Entwicklung für die Modellinteraktion

3.7 Monitoring und Wartung

- Kontinuierliche Überwachung der Modellleistung
- Aktualisierung des LLM-Wissens bei Bedarf
- Wiederholte Feinabstimmung bei veränderter Datenlage

4 Schlüsselkomponenten der Modelloptimierung

4.1 Evaluierungen (Evals)

- **Nutzen:** Systematische Tests zur Bewertung von Modellantworten.
- **Formate:** Multiple Choice, Klassifikation, Stringvergleich etc.
- **Grader-Typen:**
 - **String-Check-Grader:** Einfache String-Operationen (gleich, ungleich, enthält)
 - **Text-Similarity-Grader:** Bewertung der Ähnlichkeit zwischen Modellantwort und Referenz
 - **Model-Grader:** Nutzung eines separaten Modells zur Bewertung der Ausgaben
 - **Python-Grader:** Ausführung von Python-Code zur Bewertung
 - **Multi-Grader:** Kombination mehrerer Grader für komplexe Bewertungskriterien
- **Integrierter Prozess:** Evals sollten vor dem Fine-Tuning erstellt werden, um eine Baseline zu etablieren und den Fortschritt zu messen.

4.2 Prompt Engineering

- **Ziele:** Maximale Modellleistung ohne Training.
- **Methoden:** Klare Instruktionen, Kontextbereitstellung, Few-Shot-Beispiele.
- **Zusammenspiel mit Fine-Tuning:** Prompt Engineering kann Fine-Tuning ergänzen oder in manchen Fällen sogar ersetzen.
- **Beispiel:** Die Prompt-Konstruktion mit relevanten Beispielen (Few-Shot-Learning) kann die Leistung signifikant verbessern, ohne das Modell neu zu trainieren.

5 Best Practices

5.1 Datenstrategie

1. Datenqualität schlägt Datenmenge

- Beginnen Sie mit 50-100 hochwertigen Beispielen
- Verwenden Sie realistische Daten aus der Zielanwendung

- Stellen Sie sicher, dass die Daten repräsentativ für die Aufgabe sind

2. Vielfältige und realistische Beispiele wählen

- Decken Sie verschiedene Szenarien, Formulierungen und Nuancen ab
- Vermeiden Sie starke Verzerrungen in den Trainingsdaten
- Berücksichtigen Sie auch Randfall-Szenarien

3. Konsistente Formatierung (z. B. JSONL)

- Verwenden Sie das korrekte Format für Ihre Fine-Tuning-Methode
- Stellen Sie sicher, dass jede Zeile ein vollständiges JSON-Objekt enthält
- Validieren Sie Ihr Datenformat vor dem Training

5.2 Trainingsstrategie

4. Schrittweises Auftauen von Schichten

- Beginnen Sie mit dem Training der obersten Schichten
- Tauen Sie schrittweise tiefere Schichten auf
- Dies führt zu stabilerem Training und verhindert Overfitting

5. Kleine Lernraten zur Stabilisierung

- Verwenden Sie Lernraten zwischen 1e-4 und 2e-4 für stabile Konvergenz
- Ein Lernraten-Schedule mit Warmup und linearem Abfall kann hilfreich sein
- Experimentieren Sie mit verschiedenen Batch-Größen

6. Regelmäßige Evaluierung und Monitoring

- Setzen Sie vor dem Fine-Tuning Evaluierungen (Evals) auf
- Überwachen Sie Trainings- und Validierungsmetriken
- Implementieren Sie Early Stopping, um Overfitting zu vermeiden

5.3 Technische Exzellenz

7. Verwendung von Checkpoints und Modellversionierung

- Speichern Sie regelmäßig Zwischenstände (alle 5-8 Epochen)
- Vergleichen Sie die Leistung verschiedener Checkpoint-Modelle
- Behalten Sie die Versionshistorie bei, um Regressionen zu erkennen

8. Hyperparameter systematisch optimieren

- Nutzen Sie automatisierte Hyperparameter-Optimierung (Random Search, Grid Search, Bayesian)
- Fokussieren Sie auf Lernrate, Batch-Größe und Epochenzahl
- Dokumentieren Sie die Ergebnisse verschiedener Konfigurationen

9. Tools wie TensorBoard, W&B, MLflow einsetzen

- Visualisieren Sie Trainingsmetriken in Echtzeit
- Verfolgen Sie Experimente und deren Ergebnisse
- Vergleichen Sie verschiedene Trainingsläufe untereinander

5.4 Sicherheit und Effizienz

10. Datenschutzkonformität beachten

- Anonymisieren Sie sensible Daten vor dem Training
- Beachten Sie rechtliche Anforderungen beim Training mit personenbezogenen Daten
- Implementieren Sie Sicherheitsprüfungen für Modelleingaben und -ausgaben

11. Kosten im Blick behalten (Token-Effizienz)

- Überwachen Sie den Token-Verbrauch während des Trainings
- Optimieren Sie Prompts für kürzere Ausgaben, wo möglich
- Verwenden Sie Modell-Distillation für häufig genutzte Aufgaben

5.5 Spezifische Techniken für erweiterte Anwendungen

12. Multi-Task Learning

- Trainieren Sie das Modell für mehrere verwandte Aufgaben gleichzeitig
- Verwenden Sie spezifische Adapter für verschiedene Aufgaben
- Kombinieren Sie bei Bedarf mehrere Adapter für komplexe Anwendungen

13. Modell-Quantisierung

- Reduzieren Sie die Präzision der Modellparameter (z.B. von 32-bit auf 8-bit)
- Verwenden Sie QLoRA für effizientes Training mit quantisierten Modellen
- Testen Sie die Leistung quantisierter Modelle gründlich

14. Multimodale Integration

- Bei Vision Fine-Tuning: Achten Sie auf Bildqualität und -größe
- Verwenden Sie den "detail"-Parameter, um Trainingskosten zu optimieren
- Testen Sie verschiedene Kombinationen von Text- und Bildeingaben

6 Herausforderungen & Perspektiven

6.1 Skalierbarkeit

- **Rechnerische Ressourcen:** Fine-Tuning großer Modelle erfordert erhebliche Rechen- und Speicherkapazitäten
- **Memory-Effizienz:** Techniken wie LoRA, QLoRA und Half Fine-Tuning adressieren diese Herausforderungen
- **Zukünftige Entwicklungen:** Co-Design von Hardware und Algorithmen speziell für LLM-Training

6.2 Ethische Überlegungen

- **Bias und Fairness:** Trainingsdaten können Verzerrungen enthalten, die sich auf das Modell übertragen

- **Datenschutz:** Umgang mit sensiblen oder proprietären Daten während des Fine-Tunings
- **Transparenz und Nachvollziehbarkeit:** Dokumentation des Fine-Tuning-Prozesses und seiner Auswirkungen

6.3 Integration mit neuen Technologien

- **Internet of Things (IoT):** LLMs können IoT-Daten in Echtzeit analysieren und Entscheidungen optimieren
- **Edge Computing:** Fine-Tuned Modelle können direkt auf Edge-Geräten eingesetzt werden
- **Federated Learning:** Training über verteilte Datenquellen ohne zentrale Datenspeicherung

7 Fazit

Fazit

Fine-Tuning ist mehr als nur eine technische Maßnahme – es ist ein strategischer Hebel zur Anpassung von KI-Systemen an konkrete Anforderungen, Zielgruppen und Kontexte. Die Kombination aus fundierter Datenbasis, passenden Methoden und iterativer Evaluation bildet das Fundament für erfolgreiche KI-Projekte.

Mit Plattformen wie OpenAI lassen sich diese Prozesse effizient gestalten – von der Vorbereitung über das Training bis zur Bewertung. Erweiterte Verfahren wie DPO, RFT und Vision Fine-Tuning ermöglichen zusätzlich eine feinkörnige Kontrolle und Weiterentwicklung leistungsfähiger Modelle.

Die siebenstufige Fine-Tuning-Pipeline bietet einen strukturierten Ansatz vom Datenmanagement bis zur kontinuierlichen Überwachung, während parameter-effiziente Methoden die Ressourcennutzung optimieren. Durch den gezielten Einsatz dieser Techniken können Entwickler leistungsfähige KI-Lösungen erstellen, die genau auf ihre spezifischen Anforderungen zugeschnitten sind.

M17a - Modellauswahl

Stand: 05.2025

1 KI-Modelllandschaft: Ein Überblick

Die moderne KI-Landschaft bietet verschiedene spezialisierte Modelltypen für unterschiedliche Anwendungsfälle:

- **Reasoning-Modelle:** Spezialisiert auf logisches Denken und systematische Problemlösung (z.B. o3-mini) - diese Modelle lösen komplexe Aufgaben durch schrittweises, strukturiertes Denken.
- **Sprachmodelle:** Konzipiert für natürlichsprachliche Aufgaben wie Textgenerierung, Zusammenfassungen und Konversationen (z.B. GPT-4) - sie verstehen und erzeugen menschenähnliche Texte.
- **Codex-Modelle:** Optimiert für Codegenerierung und Programmieraufgaben - diese Modelle können Code schreiben, analysieren und debuggen.
- **Bildgenerierungsmodelle:** Erzeugen Bilder aus textlichen Beschreibungen (z.B. DALL-E) - sie wandeln Textanweisungen in visuelle Ergebnisse um.
- **Sprachverarbeitungsmodelle:** Spezialisiert auf Spracherkennung und -transkription (z.B. Whisper) - sie wandeln gesprochene Sprache in Text um.

2 Vergleich wichtiger Modelle

Die Wahl des richtigen Modells ist entscheidend für optimale Ergebnisse, Ressourcenschonung und maximale Effizienz. Hier ein Überblick der wichtigsten Modelle:

Modell	Hauptmerkmale	Empfohlene Anwendungsfälle
GPT-4o	Multimodales Allround-Modell: Versteht Text, Bilder und Audio, kann Bilder generieren. Sehr schnell und vielseitig.	Alltägliche Aufgaben, Brainstorming, Texterstellung, Content-Ideen, Bildanalysen, E-Mails, Konzepte. Gut für schnelle Dialoge und allgemeine Fragen.
GPT-4o Mini	Leichtere Version von GPT-4o: Verarbeitet Text und Bilder, ressourcenschonend und günstiger. Deutlich intelligenter als GPT-3.5-turbo.	Einfachere Aufgaben, Bildverarbeitung, schnelle und unkomplizierte Anwendungen, kostengünstige Chatbots.

Modell	Hauptmerkmale	Empfohlene Anwendungsfälle
o3-mini	Reasoning-Modell: Hohe Intelligenz bei niedrigen Kosten und geringer Latenz. Konzipiert für strukturiertes Denken.	Wissenschaftliche, mathematische und Programmieraufgaben, technische und logische Probleme, faktenbasierte Recherchen.
o4-mini	Kompaktes Reasoning-Modell: Optimiert für Geschwindigkeit und Kosteneffizienz. Stark in mathematischen, Programmier- und visuellen Aufgaben.	Komplexe Argumentationsstrukturen, technische Aufgaben, Programmierprojekte, visuelles Denken, wissenschaftliche Fragestellungen.
o3	Leistungsstärkster "Denker": Herausragend in Programmierung, Mathematik, Wissenschaft und visueller Analyse. Arbeitet mit verknüpften Einzelschritten ("Chain-of-Thought").	Komplexe Recherchen, anspruchsvolle Programmieraufgaben, Datenanalyse, strategische Planung, Code-Review und Debugging. Beste Wahl für höchste Präzision.

3 Modellauswahlprozess: Schritt für Schritt

Die Auswahl des optimalen KI-Modells erfordert einen strukturierten Prozess:

3.1 Anforderungsanalyse

- **Definition der Aufgaben:** Legen Sie fest, welche spezifischen Funktionen das Modell erfüllen soll (z.B. Textgenerierung, Fragebeantwortung).
- **Qualitätskriterien:** Bestimmen Sie, welche Qualitätsstandards (Kohärenz, Genauigkeit) erfüllt werden müssen.
- **Domänenkenntnisse:** Identifizieren Sie, welches Fachwissen für Ihre Aufgabe notwendig ist.
- **Antwortgeschwindigkeit:** Definieren Sie die akzeptable Reaktionszeit des Modells.
- **Budget:** Setzen Sie einen finanziellen Rahmen für Ihre KI-Lösung.

3.2 Bewertungskriterien

- **Verständlichkeit:** Wie klar und nachvollziehbar sind die Modellausgaben?
- **Effizienz:** Wie schnell verarbeitet das Modell Eingaben und liefert Ausgaben?
- **Skalierbarkeit:** Kann das Modell mit steigenden Anforderungen mitwachsen?
- **Kosten:** Wie hoch sind die Betriebs- und Nutzungskosten des Modells?

3.3 Recherche und Vorauswahl

- Analysieren Sie verfügbare Modelle anhand Ihrer festgelegten Kriterien und erstellen Sie eine Vorauswahl geeigneter Kandidaten.

3.4 Praktische Modellbewertung

- **Quantitative Methoden:** Verwenden Sie Benchmarks und Metriken, um die Leistung objektiv zu messen.
- **Qualitative Verfahren:** Sammeln Sie Nutzerfeedback zur praktischen Verwendbarkeit.
- **Testphase:** Erproben Sie die Modelle in einer realistischen Umgebung.

3.5 Finale Auswahl und Implementierung

- Treffen Sie eine fundierte Entscheidung für das am besten geeignete Modell und integrieren Sie es in Ihre Systeme.

[Beta-WebApp für Modellauswahl](#). 😊

4 Modellkaskade: Mehrere Modelle klug kombinieren

Die Modellkaskade kombiniert mehrere KI-Modelle, um ihre jeweiligen Stärken zu nutzen und Schwächen auszugleichen:

4.1 Beispiel für eine Modellkaskade

1. **Datenanalyse mit pandas:** Analysiert große Datensätze und erstellt statistische Zusammenfassungen
2. **Logische Strukturierung mit o3-mini:** Strukturiert die Ergebnisse und erstellt eine logische Gliederung
3. **Kreative Textgenerierung mit GPT-4o:** Verfasst ansprechende Texte basierend auf der Struktur
4. **Multimodale Präsentation:** Ergänzt den Text mit visuellen Elementen

4.2 Vorteile einer Modellkaskade

1. **Effizienzsteigerung:** Jedes Modell wird für seine Stärken optimal eingesetzt
2. **Kostenoptimierung:** Ressourcenschonende Modelle für einfache Aufgaben, teurere nur wo nötig
3. **Flexibilität:** Bearbeitung unterschiedlichster Anforderungen durch spezialisierte Modelle

5 Bewertungsmethoden für KI-Modelle

5.1 Wichtige Benchmarks

- **MMLU (Massive Multitask Language Understanding):** Standard-Benchmark über 57 Fachgebiete, der die Allgemeinbildung und Fachkenntnisse von Modellen misst.

Modell	MMLU-Score
GPT-4o	88,7%
Gemini 2.0 Ultra	90,0%
Claude 3 Opus	88,2%
Llama 3.1 405B	87,3%
gpt-4o-mini	70,0%

5.2 Bewertungsdimensionen

Die Bewertung von KI-Modellen umfasst verschiedene Aspekte:

1. Wissens- und Fähigkeitsbewertung:

- Wie gut beantwortet das Modell Fragen verschiedener Schwierigkeitsgrade?
- Wie zuverlässig ergänzt es fehlendes Wissen?
- Wie gut löst es logische und mathematische Probleme?
- Wie effektiv nutzt es externe Werkzeuge?

2. Alignment-Bewertung:

- Inwieweit stimmt das Modellverhalten mit menschlichen Werten überein?
- Wie ethisch und moralisch sind die Antworten?
- Wie fair und unvoreingenommen ist das Modell?
- Wie wahrhaftig sind die gelieferten Informationen?

3. Sicherheitsbewertung:

- Wie robust ist das Modell gegenüber Störungen und Angriffen?
- Welche potenziellen Risiken birgt die Nutzung des Modells?

5.3 Konkrete Bewertungsmethoden

5.4 Automatisierte Metriken

- **BLEU**: Misst die Übereinstimmung zwischen generiertem und Referenztext durch Vergleich von Wortgruppen.
- **ROUGE**: Bewertet die Qualität von Zusammenfassungen durch Analyse übereinstimmender Wortsequenzen.

5.5 Menschliche Bewertung

- Bewertung nach Kriterien wie Grammatik, Zusammenhang, Lesbarkeit und Relevanz
- Elo-System für den direkten Vergleich verschiedener Modelle (ähnlich wie bei Schach-Ratings)

5.6 KI-basierte Bewertung

- Einsatz leistungsfähiger Modelle zur Bewertung anderer Modelle
- Automatische Erkennung von Fehlinformationen in KI-Antworten

6 Praktische Anwendungsbereiche

Die Modellevaluierung und -auswahl findet in verschiedenen Szenarien Anwendung:

6.1 Kundenservice-Chatbots

- Auswahl eines schnellen Modells mit guter Verständlichkeit und Mehrsprachigkeit
- Bewertung nach Kundenzufriedenheit und Lösungsrate

6.2 Content-Erstellung

- Nutzung kreativer Modelle für Marketing, Social Media und Blogbeiträge
- Bewertung nach Originalität, Engagement und Konversionsraten

6.3 Technische Assistenz

- Einsatz von Reasoning-Modellen für Programmierung und Fehlerbehebung
- Bewertung nach Codequalität und Lösungsgeschwindigkeit

7 Fazit

Fazit

Zusammenfassend lässt sich sagen, dass die **Evaluierung von Large Language Models (LLMs) ein wichtiges Forschungsgebiet** ist, um ihre Fähigkeiten und Grenzen zu verstehen. Die Evaluierung umfasst verschiedene **Attribute wie Grammatikalität, Kohäsion, Gefallen, Relevanz, Flüssigkeit und Bedeutungserhalt**. Sowohl **menschliche Evaluatoren als auch LLMs selbst werden zur Bewertung eingesetzt**. Es gibt **spezifische Benchmarks und Datensätze** zur Bewertung von LLMs in verschiedenen Bereichen wie **Textgenerierung, Fragebeantwortung und Zusammenfassung**.

Ein wichtiger Aspekt der LLM-Evaluierung ist die **Sicherheitsbewertung**, die **Robustheit gegenüber adversarialen Angriffen** (manipulierte Eingaben, um LLM in die Irre zu führen) und die Identifizierung von **Risiken wie Bias und Toxizität** umfasst. Die Evaluierung kann auch auf **spezialisierte LLMs** in Bereichen wie Medizin, Recht und Finanzen zugeschnitten sein.

Verschiedene **Metriken**, darunter **Likert-Skalen** und der **BLEU-Score**, werden zur Quantifizierung der LLM-Leistung verwendet. Es gibt auch **Tools und Frameworks wie DeepEval**, die die Evaluierung erleichtern. Es ist wichtig zu beachten, dass **Evaluierungsbias existieren können**, beispielsweise eine Präferenz für längere Texte. Die **ethischen Aspekte** spielen ebenfalls eine Rolle bei der Entwicklung und Nutzung von LLMs.

8 A | Aufgabe

Die Aufgabestellungen unten bieten Anregungen, Sie können aber auch gerne eine andere Herausforderung angehen.

Anforderungsanalyse für ein KI-Projekt

Entwickeln Sie eine strukturierte Anforderungsanalyse für ein fiktives oder reales KI-Projekt.

Aufgabenstellung:

1. Wählen Sie einen konkreten Anwendungsfall (z.B. Kundenservice-Chatbot für eine Bank, Content-Generator für Social Media, oder Übersetzungstool für technische Dokumentation).
2. Definieren Sie:
 - Die primären Funktionen, die das KI-Modell erfüllen soll
 - Die spezifischen Anforderungen an das Sprachverständnis
 - Notwendige Fachkenntnisse in relevanten Domänen
 - Anforderungen an die Antwortgeschwindigkeit
 - Budget-Rahmenbedingungen
3. Erstellen Sie eine Prioritätenliste dieser Anforderungen (unbedingt erforderlich, wichtig, wünschenswert).
4. Beschreiben Sie, welche Kompromisse Sie bei konkurrierenden Anforderungen eingehen würden.

AbgabefORMAT:

Erstellen Sie ein Dokument mit Ihrer Anforderungsanalyse (1-2 Seiten).

Vergleichsanalyse bekannter KI-Modelle

Führen Sie eine vergleichende Analyse von mindestens drei verschiedenen KI-Modellen anhand vorgegebener Bewertungskriterien durch.

Aufgabenstellung:

1. Wählen Sie drei KI-Modelle aus der folgenden Liste aus:

- GPT-4o
- Claude 3 Opus
- Gemini 2.0 Ultra
- Llama 3.1
- Mistral 7B
- Ein anderes aktuelles KI-Modell Ihrer Wahl

2. Recherchieren Sie die Leistungsmerkmale dieser Modelle anhand der folgenden Kriterien:

- MMLU-Score oder vergleichbare Benchmark-Ergebnisse
- Kontextfenstergröße
- Antwortlatenz
- Kosten (pro Token oder alternativer Maßstab)
- Verfügbarkeit (API, Open-Source, etc.)
- Unterstützte Sprachen
- Multimodale Fähigkeiten (falls vorhanden)

3. Erstellen Sie eine Bewertungstabelle mit den recherchierten Informationen.

4. Verfassen Sie eine begründete Empfehlung, welches dieser Modelle sich für folgende Szenarien am besten eignen würde:

- Entwicklung eines kostengünstigen Chatbots für ein kleines Unternehmen
- Erstellung von KI-generierten Inhalten für ein internationales Nachrichtenportal
- Unterstützung bei der Software-Entwicklung

AbgabefORMAT:

Vergleichstabelle mit Bewertungen und einer Seite mit Ihren Empfehlungen.

Konzept für die qualitative Evaluation eines Sprachmodells

Entwickeln Sie ein strukturiertes Testverfahren zur qualitativen Bewertung eines Sprachmodells.

Aufgabenstellung:

1. Entwerfen Sie ein Bewertungsschema mit 5-7 qualitativen Kategorien, die für Ihre gewählte Anwendung relevant sind (z.B. Genauigkeit, Kreativität, Nützlichkeit der Antworten, Verständnis komplexer Anweisungen, Kulturelle Sensibilität).
2. Erstellen Sie für jede Kategorie:
 - Eine klare Definition, was in dieser Kategorie bewertet wird
 - Eine Bewertungsskala (z.B. 1-5 oder 1-10)
 - 2-3 konkrete Testfragen oder -aufgaben, die diese Kategorie prüfen
 - Bewertungskriterien: Was wäre eine ausgezeichnete (5/5) vs. eine unzureichende (1/5) Antwort?

3. Beschreiben Sie den Evaluationsprozess:

- Wie viele Bewerter sollten eingesetzt werden?
- Wie würden Sie die Bewertungen zusammenfassen?
- Welche Maßnahmen würden Sie ergreifen, um Bewertungsverzerrungen zu vermeiden?

4. Erläutern Sie, wie Sie die Ergebnisse dieser qualitativen Bewertung mit quantitativen Metriken (wie MMLU) kombinieren würden, um ein Gesamtbild der Modellleistung zu erhalten.

AbgabefORMAT:

Ein 2-3 seitiges Konzeptpapier mit Ihrem Evaluationsschema, den Testfragen und dem geplanten Prozess.

M17b - Reasoning Modelle



Anwendung Generativer KI

Stand: 04.2025

Die rasante Entwicklung der künstlichen Intelligenz (KI) und des Natural Language Processing (NLP) hat in den letzten Jahren zu bemerkenswerten Fortschritten im Bereich der Sprachmodelle geführt. Insbesondere Large Language Models (LLMs) haben anfänglich durch ihre Fähigkeit zur Textgenerierung und Mustererkennung beeindruckt.¹ Nun zeichnet sich mit dem Aufkommen von sogenannten Reasoning Modellen eine neue Phase ab, in der die Fähigkeit zum logischen Denken und zur Problemlösung in den Vordergrund rückt.¹ Dieser Bericht zielt darauf ab, Reasoning Modelle zu definieren, sie von anderen Chat-Modellen zu unterscheiden, ihre Vor- und Nachteile zu analysieren, typische Anwendungsfälle beider Modelltypen zu beleuchten und aktuelle Forschungstrends im Bereich der Reasoning Modelle zu untersuchen. Die Evolution von reiner Textgenerierung hin zu einem strukturierten Denkprozess kennzeichnet einen bedeutenden Paradigmenwechsel in den Fähigkeiten von KI-Systemen, der über bloße Sprachausgabe hinausgeht und eine Form des „Denkens“ ermöglicht. Die Beobachtung, dass die Leistungssteigerung durch bloße Skalierung traditioneller LLMs an ihre Grenzen stößt, hat die Entwicklung von Reasoning Modellen als einen vielversprechenden neuen Weg für zukünftige Fortschritte in der KI vorangetrieben.¹

1 Was sind Reasoning Modelle?

Reasoning Modelle sind KI-Systeme, die Natural Language Processing mit strukturierten Denkfähigkeiten verbinden.³ Ihr Design zielt nicht nur auf die Generierung von Sprache ab, sondern darauf, Probleme mit größerer Tiefe und Präzision zu durchdenken.¹ Im Kern versuchen diese Modelle, logische Prozesse zu simulieren, um Schlussfolgerungen zu ziehen oder Entscheidungen zu treffen.⁵ Dabei greifen sie häufig auf explizite Wissensrepräsentationen und Inferenzmechanismen zurück.⁵ Obwohl der Begriff „Reasoning Modell“ nicht streng definiert ist, bezieht er sich im Allgemeinen auf Modelle, die explizit strukturierte Fähigkeiten zur Problemlösung demonstrieren.⁶ Zu diesen Fähigkeiten gehören logisches Schließen (deduktiv/induktiv), mehrschrittige Problemlösung (z. B. in Mathematik, Programmierung, Rätseln), Common-Sense-Denken (Verständnis impliziten Kontexts) und kausales Denken (Verknüpfung von Ursachen und Wirkungen).⁶ Moderne

Modelle erreichen dies durch architektonische Innovationen und Training auf Datensätzen, die mit Denkaufgaben angereichert sind.⁶ Der wesentliche Unterschied liegt in der Verlagerung von der reinen Vorhersage hin zu einem Prozess, bei dem Reasoning Modelle sich auf das „Wie“ und nicht nur auf das „Was“ der Antwortgenerierung konzentrieren.¹ Während traditionelle LLMs das wahrscheinlichste nächste Token vorhersagen, generieren Reasoning Modelle durch Techniken wie Chain-of-Thought Prompting explizit eine Abfolge von Denkschritten, die menschliche Problemlösung nachahmen. Dieser interne Prozess ermöglicht die Fehlererkennung und Selbstkorrektur. Die Entwicklung von Reasoning Modellen markiert somit einen Schritt hin zu einer KI, die Aufgaben bewältigen kann, die mehr als nur Mustererkennung und Informationsabruft erfordern.¹ Die Fähigkeit, logische Schlüsse zu ziehen, Kausalitäten zu verstehen und mehrschrittige Probleme zu lösen, deutet auf ein höheres kognitives Niveau im Vergleich zu Modellen hin, die primär auf Sprachgenerierung ausgerichtet sind.

2 Abgrenzung zu Chat-Modellen

Chat-Modelle sind Sprachmodelle, die eine Sequenz von Nachrichten als Eingabe verwenden und Nachrichten als Ausgabe zurückgeben.⁷ Ihr Hauptaugenmerk liegt auf der Generierung von menschenähnlichem Text für Konversationszwecke.⁴ Sie sind darauf ausgelegt, menschliche Sprache oder Schrift zu verstehen und darauf zu reagieren, wodurch sie menschenähnliche Gespräche nachahmen.⁸ Diese Modelle werden auf riesigen Textkorpora trainiert, um statistische Muster zu erlernen.⁶ Typische Aufgaben von Chat-Modellen umfassen Übersetzung, Textgenerierung, Sentimentanalyse, Zusammenfassung und Beantwortung von Fragen.⁶ Obwohl sie ein gewisses Maß an Denkfähigkeit zeigen können, ist dies oft implizit und nicht das primäre Ziel ihres Designs.⁴ Chat-Modelle zeichnen sich durch ihre Fähigkeit aus, flüssigen und kontextrelevanten Text zu generieren, es fehlt ihnen jedoch möglicherweise die strukturierte Denkweise und die Fähigkeit zum logischen Schlussfolgern von Reasoning Modellen.⁹ Die primäre Funktion eines Chat-Modells besteht darin, menschenähnliche Sprache zu produzieren. Obwohl fortgeschrittene Modelle Fragen beantworten und Texte zusammenfassen können, basiert ihr zugrunde liegender Mechanismus hauptsächlich auf Mustererkennung. Reasoning Modelle hingegen sind speziell darauf ausgelegt, logische Operationen durchzuführen und Probleme in einer Reihe von Schritten zu lösen. Der Aufstieg der generativen KI hat Chat-Modelle erheblich verbessert, sie menschenähnlicher gemacht und in die Lage versetzt, ein breiteres Spektrum von Anfragen zu bearbeiten.⁸ Generative KI-Techniken, die von LLMs angetrieben werden, haben es Chat-Modellen ermöglicht, über einfache regelbasierte Antworten hinauszugehen und personalisiertere und kontextbewusstere Antworten zu generieren. Dies entspricht jedoch nicht unbedingt dem eigentlichen Denken, wie es Reasoning Modelle einsetzen.

3 Architektonische Unterschiede

Reasoning Modelle bauen oft auf der Architektur von LLMs auf, führen aber neue Verhaltensweisen wie strukturiertes Denken ein.¹ Ein wesentlicher Unterschied liegt in der

Verwendung von Prompting-Techniken wie „Chain of Thought“ (CoT), „Tree of Thought“ (ToT) und „Graph of Thought“, um das Denken zu ermöglichen.¹ CoT fordert das Modell auf, eine Frage zu beantworten, indem es zunächst eine Kette von Denkschritten generiert.⁴ ToT verallgemeinert CoT, indem es das Modell anweist, einen oder mehrere „mögliche nächste Schritte“ zu generieren und das Modell dann auf jeden dieser Schritte anzuwenden.⁴ Graph of Thought stellt eine weitere Verallgemeinerung dar, bei der die Denkschritte einen gerichteten azyklischen Graphen bilden.⁴ Darüber hinaus nutzen Reasoning Modelle häufig Retrieval-Augmented Generation (RAG), um Informationen aus externen Wissensquellen zu integrieren und so ihre Denkfähigkeit zu verbessern.⁴ Die Möglichkeit zur Werkzeugnutzung, die es Modellen erlaubt, externe Methoden wie Taschenrechner oder Programminterprete aufzurufen, ist ein weiteres wichtiges architektonisches Merkmal.⁴ Moderne Reasoning Modelle verwenden auch Techniken wie spärliche Aufmerksamkeit (z. B. Mistral) oder Mixture-of-Experts-Ansätze (z. B. DeepSeek), um ihre Effizienz und Leistungsfähigkeit zu steigern.⁶ Im Gegensatz dazu konzentriert sich die allgemeine Architektur anderer Chat-Modelle primär auf Transformer-Netzwerke für Sequenz-zu-Sequenz-Aufgaben.⁹ Reasoning Modelle erweitern somit die Standardarchitektur von LLMs durch spezifische Mechanismen, die darauf ausgelegt sind, strukturierte Denkprozesse zu erleichtern.¹ Techniken wie CoT sind nicht inhärent in der Basisarchitektur von LLMs vorhanden, sondern werden durch Prompting oder Feinabstimmung angewendet. Diese explizite Anleitung ermutigt das Modell, einen Denkprozess zu simulieren, was ein wesentliches Unterscheidungsmerkmal zu Standard-Chat-Modellen darstellt, die primär auf den inhärenten Mustererkennungsfähigkeiten der Transformer-Architektur beruhen. Die Fähigkeit, externe Werkzeuge und Wissensdatenbanken zu nutzen, verbessert die Denkfähigkeiten dieser Modelle erheblich.⁴ Durch die Integration mit Werkzeugen wie Taschenrechnern oder Suchmaschinen können Reasoning Modelle Einschränkungen in ihrem internen Wissen und ihren Rechenfähigkeiten überwinden. RAG erweitert dies weiter, indem es ihnen ermöglicht, auf riesige Mengen externer Informationen zuzugreifen und diese zu verarbeiten, wodurch ihre Antworten auf faktischen Daten basieren.

4 Der Einfluss der Trainingsdaten

Reasoning Modelle werden oft auf Datensätzen trainiert, die mit Denkaufgaben angereichert sind, wie z. B. mathematischen Problemen, Logikrätseln und Programmieraufgaben.⁶ Um die Denkfähigkeiten weiter zu verbessern, werden häufig Supervised Fine-Tuning (SFT) und Reinforcement Learning (RL) eingesetzt.³ Beim RL spielen Belohnungsmodelle eine wichtige Rolle, um den Trainingsprozess zu steuern.⁴ Im Gegensatz dazu werden andere Chat-Modelle auf riesigen Datensätzen mit allgemeinem Text und Code trainiert, um breite Sprachmuster zu erlernen.⁶ Für Reasoning Modelle ist die Qualität kuratierter Datensätze von entscheidender Bedeutung.⁶ Die spezialisierten Trainingsdaten, die für Reasoning Modelle verwendet werden, sind entscheidend, um die Fähigkeit zur strukturierten Problemlösung zu vermitteln.⁶ Während allgemeine Sprachmodelle aus einer breiten Palette von Texten lernen, benötigen Reasoning Modelle die Auseinandersetzung mit spezifischen Datentypen, die explizit logisches Denken und Problemlösung demonstrieren. Dieses

gezielte Training ermöglicht es ihnen, die notwendigen Fähigkeiten für Aufgaben wie mathematische Inferenz oder logische Deduktion zu entwickeln. Techniken wie SFT und RL ermöglichen die Feinabstimmung der Modelle, um sich besser an das gewünschte Denkverhalten anzupassen und die Leistung bei denkintensiven Aufgaben zu verbessern.³ Vorab trainierte Sprachmodelle bieten eine starke Grundlage, aber das weitere Training durch SFT mit Beispielen für Denkprozesse und RL mit Belohnungssignalen, die korrekte Denkschritte belohnen, trägt dazu bei, die Denkfähigkeiten dieser Modelle spezifisch zu entwickeln und zu verfeinern.

5 Fähigkeiten im Vergleich

Reasoning Modelle zeichnen sich durch ihre Fähigkeiten im logischen Denken, in der Problemlösung und im Ziehen von Schlussfolgerungen aus.¹ Sie zeigen eine verbesserte Genauigkeit bei komplexen Aufgaben, die mehrschrittige Inferenz erfordern.¹ Zudem sind sie in der Lage, nuancierte Probleme zu bearbeiten und implizite Kontexte zu verstehen.¹ Ihre Fähigkeit zur besseren faktischen Fundierung und zur Reduzierung von Halluzinationen bei Denkaufgaben ist ein weiterer Vorteil.¹ Im Gegensatz dazu liegen die Stärken anderer Chat-Modelle in der Generierung kreativer Inhalte, der Führung natürlicher klingender Gespräche, der Textzusammenfassung und der Sprachübersetzung.¹ Diese Modelle glänzen bei Aufgaben, die auf Mustererkennung und flüssiger Textgenerierung beruhen.⁹

Merkmal	Reasoning Modelle	Andere Chat-Modelle
Primäres Ziel	Problemlösung, logisches Schließen	Flüssige und ansprechende Konversation, Informationsgenerierung
Schlüsseltechniken	Chain of Thought, Tree of Thought, Graph of Thought, Werkzeugnutzung, RAG	Transformer-Architektur, Mustererkennung
Stärken	Genauigkeit bei komplexen Aufgaben, Erklärbarkeit, Umgang mit nuancierten Problemen	Generierung kreativer Inhalte, natürliche Konversation, Zusammenfassung, Übersetzung
Schwächen	Höhere Rechenkosten, langsamere Reaktionszeiten, potenzielle Fehler im logischen Denken	Begrenzte logische Inferenz, potenzielle faktische Ungenauigkeiten bei komplexen Aufgaben
Typische Anwendungsfälle	Mathematische Probleme, komplexe Fragenbeantwortung, Code-Debugging, juristische Analyse	Kundenservice, virtuelle Assistenten, Inhaltserstellung, Sprachübersetzung

Die Erklärbarkeit, die Reasoning Modelle durch Techniken wie CoT bieten, ist ein bedeutender Vorteil, insbesondere in sensiblen Anwendungen, in denen das Verständnis der

Begründung einer KI-Ausgabe entscheidend ist.¹ Im Gegensatz dazu konzentrieren sich andere Chat-Modelle primär auf die Erzeugung von menschenähnlichem Text. Die Integration externer Werkzeuge erweitert die Fähigkeiten von Reasoning Modellen über ihr internes Wissen hinaus und ermöglicht es ihnen, ein breiteres Spektrum komplexer Probleme zu bewältigen.⁴ Durch die Möglichkeit, auf Ressourcen wie Taschenrechner, Datenbanken oder APIs zuzugreifen und diese zu nutzen, können Reasoning Modelle spezialisierte Werkzeuge für Aufgaben einsetzen, die mit ihren reinen Sprachverarbeitungsfähigkeiten nicht möglich wären.

6 Vorteile von Reasoning Modellen

Die verbesserte Fähigkeit zum logischen Denken, zur Problemlösung und zum Ziehen von Schlussfolgerungen ist ein zentraler Vorteil von Reasoning Modellen.¹ Sie bieten eine höhere Genauigkeit bei komplexen, mehrschrittigen Aufgaben.¹ Ein weiterer wesentlicher Vorteil ist die erhöhte Erklärbarkeit und Interpretierbarkeit, die durch die Generierung von Denkschritten ermöglicht wird.¹ Dies führt zu einer besseren faktischen Fundierung und einer potenziellen Reduzierung von Halluzinationen in denkintensiven Bereichen.¹ Die Fähigkeit, externe Werkzeuge zur Verbesserung der Problemlösung zu nutzen, ist ebenfalls ein bedeutender Vorteil.⁴ Dies macht sie besonders geeignet für Bereiche, die Präzision und überprüfbare Lösungen erfordern.⁴ Die Erklärbarkeit, die Reasoning Modelle bieten, ist besonders in sensiblen Anwendungen von Bedeutung, in denen das Verständnis der Gründe für die Ausgabe einer KI entscheidend ist.¹ In Bereichen wie Medizin oder Recht, in denen Entscheidungen schwerwiegende Folgen haben, kann die Fähigkeit eines Reasoning Modells, seinen Denkprozess zu artikulieren, für die Überprüfung, Auditierung und den Aufbau von Vertrauen in das KI-System von unschätzbarem Wert sein. Die Integration externer Werkzeuge erweitert die Fähigkeiten von Reasoning Modellen über ihr internes Wissen hinaus und ermöglicht es ihnen, ein breiteres Spektrum komplexer Probleme zu bewältigen.⁴ Durch die Möglichkeit, auf Ressourcen wie Taschenrechner, Datenbanken oder APIs zuzugreifen und diese zu nutzen, können Reasoning Modelle spezialisierte Werkzeuge für Aufgaben einsetzen, die mit ihren reinen Sprachverarbeitungsfähigkeiten nicht möglich wären.

7 Nachteile von Reasoning Modellen

Die potenziell höheren Rechenkosten und langsameren Reaktionszeiten aufgrund des Denkprozesses sind ein wesentlicher Nachteil von Reasoning Modellen.¹ Es besteht die Gefahr eines fehlerhaften Denkens, wenn die zugrunde liegende Logik oder Annahmen falsch sind.⁴ Im Vergleich zu Modellen, die speziell für offene Gesprächskontexte entwickelt wurden, kann die Generalisierbarkeit in solchen Szenarien begrenzt sein.⁴ Zudem ist die Notwendigkeit spezialisierter Trainingsdaten, die Denkschritte enthalten, ein weiterer Nachteil.⁴ Es besteht auch das Risiko erhöhter Halluzinationen in längeren Inferenzketten.¹ Selbst wenn der „Denkprozess“ angezeigt wird, kann eine gewisse Undurchsichtigkeit bestehen bleiben, da der Denkpfad plausibel, aber letztendlich falsch sein kann.¹ Die erhöhte Komplexität der Werkzeuge für Entwicklung und Einsatz ist ebenfalls ein Faktor.¹

Experimentelle Reasoning Modelle können auch bei einfacheren Aufgaben Inkonsistenzen aufweisen.⁶ Die erhöhten Rechenanforderungen von Reasoning Modellen können ein erhebliches Hindernis für ihre breite Akzeptanz darstellen, insbesondere in Anwendungen, die Echtzeitreaktionen erfordern.¹ Der mehrschrittige Charakter des Denkens erfordert oft mehr Rechenleistung und Zeit im Vergleich zur direkten Textgenerierung in anderen Chat-Modellen. Dies kann zu höheren Betriebskosten und längeren Latenzzeiten führen, was für nicht alle Anwendungsfälle akzeptabel ist. Trotz des Ziels einer verbesserten Genauigkeit kann die Komplexität des Denkens auch neue Wege für Fehler oder „Halluzinationen“ eröffnen, insbesondere in mehrschritten Inferenzprozessen.¹ Obwohl Reasoning Modelle darauf ausgelegt sind, zuverlässiger zu sein, können die längeren Denkketten Möglichkeiten für die Anhäufung von Fehlern schaffen, was zu falschen Schlussfolgerungen führt, selbst wenn einzelne Schritte logisch erscheinen.

Nachteil	Beschreibung
Höhere Rechenkosten	Der Denkprozess erfordert mehr Rechenleistung.
Langsamere Reaktionszeiten	Die Generierung und Auswertung von Denkschritten kann zeitaufwendiger sein.
Potenzielle Fehler im logischen Denken	Wenn der Denkprozess fehlerhaft ist oder auf falschen Annahmen beruht, ist das Ergebnis wahrscheinlich falsch.
Begrenzte Generalisierbarkeit in Gesprächen	Ihre Stärke liegt in Denkaufgaben, in offenen oder kreativen Gesprächsszenarien schneiden sie möglicherweise schlechter ab.
Spezialisierte Trainingsdaten erforderlich	Effektive Reasoning Modelle erfordern oft Datensätze, die nicht nur Antworten, sondern auch die beteiligten Denkschritte enthalten.
Potenzial für erhöhte Halluzinationsrisiken	Längere Inferenzketten können Fehler verstärken, wenn die ersten Schritte fehlerhaft sind.
Undurchsichtigkeit	Selbst wenn sie ihren „Denkprozess“ zeigen, können die bereitgestellten Denkpfade plausibel, aber letztendlich falsch sein.
Erhöhte Komplexität der Werkzeuge	Die Entwicklung und Bereitstellung von Anwendungen mit Reasoning Modellen erfordert eine sorgfältigere Verwaltung von Speicher, Token und Kosten.
Potenzielle Inkonsistenzen bei einfachen Aufgaben	Einige experimentelle Reasoning Modelle können bei einfacheren Aufgaben Inkonsistenzen aufweisen.

8 Anwendungsfälle von Reasoning Modellen

Typische Anwendungsfälle für Reasoning Modelle umfassen das Lösen mathematischer Textaufgaben, die Beantwortung komplexer Fragen in spezifischen Bereichen wie Wissenschaft oder Recht, das Debuggen von Code und Aufgaben im Bereich des

Competitive Programming.⁴ Sie werden auch zur Analyse juristischer Dokumente, zur Strategieplanung, zur Diagnose komplexer technischer Probleme, zur Erkundung von Finanzmärkten und zum Verfassen analytischer Berichte eingesetzt.¹ Weitere Anwendungsbereiche sind die Unterstützung bei der Programmierung, die wissenschaftliche Forschung, das Lösen logischer Rätsel sowie komplexe Planungs- und Entscheidungsprozesse.⁶ Im Gesundheitswesen können sie bei der Diagnose und Behandlungsplanung helfen, in der Finanzbranche bei der Betrugserkennung und Risikoanalyse, und im Bereich Compliance bei der Einhaltung von Vorschriften.¹⁵ Reasoning Modelle sind besonders wertvoll in Bereichen, in denen Genauigkeit, logische Deduktion und die Fähigkeit, Komplexität zu bewältigen, von größter Bedeutung sind.¹ Die Fähigkeit von Reasoning Modellen, ihren „Denkprozess“ durch Techniken wie CoT darzustellen, macht sie auch für Bildungszwecke geeignet, da sie schrittweise Erklärungen liefern und das Lernen erleichtern können.¹⁴

9 Anwendungsfälle anderer Chat-Modelle

Andere Chat-Modelle finden typischerweise Anwendung in Kundenservice-Chatbots, virtuellen Assistenten für Terminplanung und Erinnerungen, der Generierung kreativer Texte, der Textzusammenfassung, der Sprachübersetzung und der Teilnahme an zwanglosen Gesprächen.¹ Sie werden auch zur Erstellung von Inhalten (Artikel, Blogbeiträge, Marketingtexte), zur Informationsbeschaffung und zur Lead-Generierung eingesetzt.¹ Weitere Anwendungsbereiche umfassen Mitarbeitersupport, kostenlose Übersetzung und 24/7-Support.¹⁶ Im Gesundheitswesen können sie bei der Symptomprüfung und Terminvereinbarung helfen, im Finanzwesen beim Abrufen von Kontoständen und bei Transaktionen, und im Einzelhandel bei Produktempfehlungen.¹⁷ Andere Chat-Modelle zeichnen sich durch ihre breite Anwendbarkeit aus, bei der flüssige und ansprechende Kommunikation im Vordergrund steht, und dienen oft als wertvolle Werkzeuge für die Automatisierung und Informationsverbreitung.¹ Die Vielseitigkeit anderer Chat-Modelle ergibt sich aus ihrem breiten Training auf diversen Textdaten, wodurch sie ein breites Spektrum an Themen und Aufgaben bearbeiten können.⁶

10 Aktuelle Forschungstrends

Die aktuelle Forschung im Bereich der Reasoning Modelle konzentriert sich auf die Verbesserung von Prompting-Techniken wie Tree of Thought (ToT) und Graph of Thought (GoT).⁴ Ein weiterer wichtiger Trend ist die Weiterentwicklung der Retrieval-Augmented Generation (RAG).⁴ Die Forschung zielt auch darauf ab, die Fähigkeit der Modelle zur Nutzung externer Werkzeuge zu verbessern.⁴ Supervised Fine-Tuning (SFT) mit Reasoning-Traces ist ein weiterer aktiver Forschungsbereich.⁴ Reinforcement Learning (RL) wird intensiv zur Verbesserung der Denkfähigkeiten eingesetzt.⁴ Darüber hinaus werden Techniken wie Guided Sampling und Self-Consistency Decoding untersucht.⁴ Die Entwicklung anspruchsvollerer Benchmarks zur Bewertung der Denkfähigkeiten ist ebenfalls ein wichtiger Forschungstrend.⁴ Im Bereich der Architektur werden Hybridmodelle (die schnelles und langsames Denken kombinieren) und agentische Frameworks erforscht,

ebenso wie die Verbesserung des kausalen Denkens, die Reduzierung von Halluzinationen, die Erhöhung der Transparenz und die Entwicklung spezialisierter Werkzeuge und domänenpezifischer Agenten.¹ Ein weiterer Fokus liegt auf der Effizienzsteigerung und Kostenreduktion.⁶ Die zunehmende Verfügbarkeit von Open-Source-Reasoning-Modellen ist ein wichtiger Trend.⁶ Schließlich wird an der Integration von neuro-symbolischer KI und Common-Sense-Wissen gearbeitet⁵, ebenso wie an dynamischen Lernmodellen, domänenübergreifendem Denken und dem Einsatz von Quantencomputing.¹⁵ Auch die Verbesserung der Erklärbarkeit und die Integration ethischer Überlegungen sind wichtige Forschungsziele.¹⁵ Die aktuellen Forschungstrends deuten stark darauf hin, dass die zukünftige Entwicklung von Reasoning Modellen auf die Steigerung ihrer Zuverlässigkeit, Effizienz und Erklärbarkeit abzielt, während gleichzeitig ihre Fähigkeiten durch die Integration von Werkzeugen und hybriden Ansätzen erweitert werden. Die zunehmende Verfügbarkeit von Open-Source-Reasoning-Modellen wird voraussichtlich die Innovation beschleunigen und den Zugang zu dieser fortschrittlichen Technologie demokratisieren.

11 Fazit

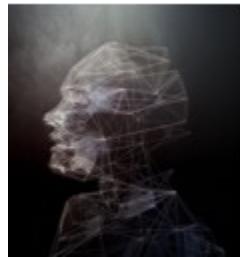
Zusammenfassend lässt sich festhalten, dass Reasoning Modelle und andere Chat-Modelle sich primär in ihrem Fokus und ihren Fähigkeiten unterscheiden. Während Reasoning Modelle auf logisches Denken und Problemlösung ausgerichtet sind, konzentrieren sich andere Chat-Modelle auf flüssige und ansprechende Konversationen. Die Vorteile von Reasoning Modellen liegen in ihrer verbesserten Genauigkeit bei komplexen Aufgaben, ihrer erhöhten Erklärbarkeit und ihrer Fähigkeit, nuancierte Probleme zu bearbeiten. Zu ihren Nachteilen zählen höhere Rechenkosten und langsamere Reaktionszeiten. Typische Anwendungsfälle für Reasoning Modelle finden sich in Bereichen, die logische Deduktion und die Fähigkeit zur Bewältigung komplexer Probleme erfordern, wie z. B. in der Wissenschaft, Technik und im Finanzwesen. Andere Chat-Modelle hingegen eignen sich hervorragend für Kundenservice, die Erstellung von Inhalten und die allgemeine Sprachinteraktion. Die aktuellen Forschungstrends im Bereich der Reasoning Modelle deuten auf eine kontinuierliche Weiterentwicklung hin, mit dem Ziel, ihre Leistungsfähigkeit, Effizienz und Anwendbarkeit in verschiedenen Domänen zu verbessern. Reasoning Modelle stellen somit einen bedeutenden Schritt in der Evolution der künstlichen Intelligenz dar und haben das Potenzial, neue Dimensionen der Problemlösungsfähigkeiten zu erschließen.

12 Referenzen

1. The Rise of Reasoning Models: Unlocking the Next Phase of AI, Zugriff am April 4, 2025, <https://opendatascience.com/the-rise-of-reasoning-models-unlocking-the-next-phase-of-ai/>
2. The Rise of Reasoning Models: Unlocking the Next Phase of AI | by ODSC - Medium, Zugriff am April 4, 2025, <https://medium.com/@odsc/the-rise-of-reasoning-models-unlocking-the-next-phase-of-ai-8d82683cb5ec>

3. en.wikipedia.org, Zugriff am April 4, 2025,
https://en.wikipedia.org/wiki/Reasoning_language_model#:~:text=Reasoning%20language%20models%20are%20artificial,initialized%20with%20pretrained%20language%20models.
4. Reasoning language model - Wikipedia, Zugriff am April 4, 2025,
https://en.wikipedia.org/wiki/Reasoning_language_model
5. What is the Difference Between Reasoning Models and Other AI ..., Zugriff am April 4, 2025, <https://blog.stackademic.com/what-is-the-difference-between-reasoning-models-and-other-ai-models-35d0cdbfb5ae>
6. Exploring Reasoning Models in AI Marketplace, Feb 25 | dasarpAI, Zugriff am April 4, 2025, <https://dasarpai.com/dsblog/exploring-reasoning-models>
7. Chat models | 🦜 LangChain, Zugriff am April 4, 2025,
<https://python.langchain.com/docs/integrations/chat/>
8. What are NLP chatbots and how do they work? - Zendesk, Zugriff am April 4, 2025, <https://www.zendesk.com/blog/nlp-chatbot/>
9. AI Reasoning Models Emerge as Key Differentiator for Business, Zugriff am April 4, 2025, <https://www.pymnts.com/artificial-intelligence-2/2025/ais-dual-nature-reasoning-models-emerge-as-key-differentiator-for-business/>
10. Advancing Reasoning in Large Language Models: Promising Methods and Approaches, Zugriff am April 4, 2025, <https://arxiv.org/html/2502.03671v1>
11. How does reasoning improve NLP models? - Milvus, Zugriff am April 4, 2025, <https://milvus.io/ai-quick-reference/how-does-reasoning-improve-nlp-models>
12. AI Reasoning Models: OpenAI o3-mini, o1-mini, and DeepSeek R1, Zugriff am April 4, 2025, <https://www.backblaze.com/blog/ai-reasoning-models-openai-o3-mini-o1-mini-and-deepseek-r1/>
13. NLP vs LLM: A Comprehensive Guide to Understanding Key ..., Zugriff am April 4, 2025, <https://medium.com/@vaniukov.s/nlp-vs-llm-a-comprehensive-guide-to-understanding-key-differences-0358f6571910>
14. Reasoning in large language models: a dive into NLP logic - Toloka, Zugriff am April 4, 2025, <https://toloka.ai/blog/reasoning-in-large-language-models-a-dive-into-nlp-logic/>
15. What is Reasoning in AI? Types and Applications in 2025 - Aisera, Zugriff am April 4, 2025, <https://aisera.com/blog/ai-reasoning/>
16. The Ultimate Guide to NLP Chatbots in 2025 - Botpress, Zugriff am April 4, 2025, [https://botpress.com/blog/nlp-chatbot](<https://botpress.com/blog/nlp-chatbot>)
17. A Comprehensive Guide to NLP Chatbots| Consensus, Zugriff am April 4, 2025, <https://www.consensus.com/blog/guide-to-nlp-chatbots/>

M17c - Architekturformen



Anwendung Generativer KI

Stand: 05.2025

Künstliche Intelligenz hat in den letzten Jahren enorme Fortschritte gemacht – nicht zuletzt durch den Einsatz spezialisierter Modellarchitekturen. Drei Architekturformen stechen dabei besonders hervor: **Transformer**, **MoE (Mixture of Experts)** und **Diffusionsmodelle**. Sie bilden das technologische Rückgrat vieler moderner Anwendungen in Sprachverarbeitung, Bildgenerierung, Code-Assistenz und multimodaler KI.

1 Transformer – Die Grundlage moderner Sprachmodelle

Die Transformer-Architektur wurde 2017 mit dem bahnbrechenden Paper "Attention is All You Need" eingeführt und revolutionierte die Verarbeitung von Sequenzdaten wie Text. Im Gegensatz zu rekurrenten Netzwerken (RNNs), die Daten schrittweise verarbeiten, kann der Transformer alle Elemente einer Eingabesequenz gleichzeitig betrachten. Dies geschieht durch den Einsatz des sogenannten Self-Attention-Mechanismus, der Kontextinformationen über die gesamte Sequenz hinweg berücksichtigt.

Ein wesentlicher Vorteil ist die Möglichkeit zum parallelen Training, was die Effizienz beim Training großer Modelle erheblich steigert. Transformer können zudem besser mit langen Abhängigkeiten in Texten umgehen und sind flexibler in der Architekturgestaltung.

Kernkomponenten:

- **Self-Attention:** Berechnet die Wichtigkeit jedes Tokens im Kontext der gesamten Eingabe.
- **Positionskodierung:** Da der Transformer keine inhärente Reihenfolge kennt, werden Positionsinformationen addiert.
- **Encoder-Decoder-Struktur** (für Aufgaben wie maschinelle Übersetzung), **reine Decoder-Modelle** (z. B. GPT für Textgenerierung) oder **reine Encoder-Modelle** (z. B. BERT für Klassifikation und Verständnis).

Transformer sind heute die dominierende Architekturform für Sprachverarbeitung, Codierung, multimodale Aufgaben und sogar für spezielle Anwendungsgebiete wie die Vorhersage von Proteinstrukturen. Ihre Flexibilität und Skalierbarkeit haben sie zur Grundlage moderner KI-Anwendungen gemacht.

2 MoE (Mixture of Experts) – Effiziente Skalierung durch Spezialisten

MoE-Modelle (Mixture of Experts) setzen auf eine modulare Architektur, in der viele spezialisierte Subnetzwerke – sogenannte "Experten" – zur Verfügung stehen. Für jede Eingabe entscheidet ein sogenanntes **Gating-Modul**, welche wenigen Experten (z. B. 2 von 64) tatsächlich aktiviert werden. Damit wird die Rechenlast reduziert, ohne die Gesamtkapazität des Modells zu verringern.

Diese Architektur ermöglicht es, extrem große Modelle mit hoher Parameteranzahl effizient zu betreiben, da nur ein Bruchteil der Parameter für eine einzelne Inferenz genutzt wird. MoE erlaubt zudem eine gewisse Spezialisierung: Manche Experten werden häufiger für bestimmte Datenarten oder Aufgaben aktiviert, was zu einer besseren Gesamtauslastung und differenzierterem Lernen führen kann.

Vorteile:

- **Effizienz:** Geringerer Rechenaufwand durch selektive Aktivierung von Teilnetzwerken.
- **Skalierbarkeit:** Modelle mit mehreren Billionen Parametern werden realisierbar.
- **Spezialisierung:** Experten lernen unterschiedliche Aufgaben oder Datenmuster.

MoE-Architekturen werden vor allem in großskaligen Sprach- und Multimodellen eingesetzt. Google entwickelte mit Switch Transformer und GShard zwei prominente Beispiele. Auch GPT-4 wird mit hoher Wahrscheinlichkeit als MoE-Modell betrieben. Herausforderungen bestehen in der ausgewogenen Nutzung der Experten (Load Balancing), der Stabilität im Training und der effizienten Verteilung über Hardware-Infrastruktur.

3 Diffusionsmodelle – Hochwertige Bildsynthese durch Rauschen

Diffusionsmodelle stellen eine neuartige Herangehensweise zur Generierung komplexer Daten dar. Sie sind besonders bekannt für ihre Anwendung im Bereich der Bildsynthese, finden aber zunehmend auch Einsatz in Audio-, Video- und 3D-Generierung. Ihr Grundprinzip basiert auf zwei Phasen: einem Vorwärtsprozess, in dem ein Bild schrittweise verändert wird, und einem Rückwärtsprozess, in dem ein neuronales Netzwerk lernt, dieses Rauschen wieder zu entfernen.

Dabei erzeugt das Modell aus reinem Rauschen ein hochauflösendes und detailreiches Bild, das einem realistischen Eingabebild sehr nahekommt. Der Trainingsprozess ist stabiler als bei anderen generativen Ansätzen wie GANs und lässt sich gut kontrollieren.

Typischer Ablauf:

1. **Vorwärtsprozess:** Das ursprüngliche Bild wird über viele Schritte mit immer mehr Rauschen überlagert.
2. **Rückwärtsprozess:** Ein Modell wird trainiert, diese Rauschschritte rückgängig zu machen und die ursprünglichen Inhalte zu rekonstruieren.

Vorteile:

- **Hervorragende Bildqualität**, oft besser als bei GANs.
- **Stabile Trainingsdynamik** ohne die typischen Instabilitäten adversieller Verfahren.
- **Hohe Flexibilität** für Anwendungen wie Inpainting, Text-zu-Bild-Generierung, Stilübertragungen oder sogar Animation.

Bekannte Modelle, die auf dieser Technik beruhen, sind **Stable Diffusion**, **DALL·E 2**, **Imagen** und viele neuere multimodale Generatoren. Diffusionsmodelle gelten als die vielversprechendste Richtung im Bereich generativer Medien.

4 Small Language Models (SLMs) – Kompakte KI für lokale Anwendungen

Small Language Models (SLMs) repräsentieren einen gegenläufigen Trend zur kontinuierlichen Skalierung der Modellgröße. Statt auf immer größere Parameteranzahlen zu setzen, fokussieren sich SLMs auf Effizienz, lokale Ausführbarkeit und spezifische Anwendungsfälle. Diese Modelle mit typischerweise weniger als 10 Milliarden Parametern bieten ein ausgewogenes Verhältnis zwischen Leistungsfähigkeit und Ressourcenbedarf.

Die Entwicklung dieser kompakten Modelle wurde durch Fortschritte in Kompressionstechniken, effizienteren Trainingsmethoden und architektonischen Optimierungen ermöglicht. Sie adressieren zentrale Herausforderungen großer Modelle wie hohe Betriebskosten, Latenzprobleme, Datenschutzbedenken und eingeschränkte Zugänglichkeit.

4.1 Wichtigste Techniken:

1. **Modellkompression:** Durch Verfahren wie Knowledge Distillation, Pruning und Quantisierung werden große Modelle kompakter gemacht, ohne signifikant an Leistung zu verlieren.
2. **Architekturoptimierung:** Anpassungen wie Sparse Attention, effiziente Aktivierungsfunktionen und optimierte Embedding-Schichten reduzieren den Rechenaufwand.
3. **Domänenspezifisches Training:** Fokussierung auf bestimmte Anwendungsgebiete statt Generalisierung über alle Domänen hinweg.
4. **Parameter-Effizienz:** Techniken wie LoRA (Low-Rank Adaptation) oder Adapter erlauben effiziente Feinabstimmung mit wenigen trainierbaren Parametern.

4.2 Vorteile:

- **Lokale Ausführung:** Können direkt auf Endgeräten ohne Cloud-Anbindung laufen.
- **Datenschutz:** Verarbeitung der Daten bleibt auf dem Gerät, was Privacy-by-Design ermöglicht.
- **Geringere Kosten:** Reduzierter Energie- und Ressourcenverbrauch bei Training und Inferenz.
- **Niedrigere Latenz:** Schnellere Antwortzeiten durch kompaktere Berechnungen.
- **Zugänglichkeit:** Demokratisieren KI-Technologie durch geringere Hardwareanforderungen.

Zu den bekanntesten Vertretern dieser Kategorie zählen TinyLlama (1,1B Parameter), Phi-2 (2,7B), Mistral-7B, Google Gemma (2B/7B) und FLAN-T5-Small. Diese Modelle zeigen, dass auch mit deutlich weniger Parametern beeindruckende Ergebnisse erzielt werden können, insbesondere bei spezifischen Aufgaben und nach effektivem Finetuning.

SLMs werden bereits erfolgreich in Bereichen wie mobilen Anwendungen, Embedded Systems, IoT-Geräten und in Umgebungen mit begrenzter Konnektivität oder hohen Datenschutzanforderungen eingesetzt. Sie stellen einen wichtigen Entwicklungszweig dar, der die praktische Anwendbarkeit von KI-Technologien wesentlich erweitert.

5 Zusammenfassung

Architektur	Einsatzgebiet	Kernidee	Vorteile	Beispiele
Transformer	Text, Sprache, Codierung, Multimodalität	Self-Attention zur parallelen Verarbeitung von Sequenzen	Paralleles Training, lange Abhängigkeiten, flexibel	GPT-4, BERT, T5, LLaMA
MoE (Mixture of Experts)	Großskalige Sprach- und Multimodalmodelle	Spezialisierte Teilmodelle, nur wenige pro Abfrage aktiv	Effizienz, Skalierbarkeit, Spezialisierung	Switch Transformer, GShard, GPT-4
Diffusionsmodell	Bild-, Audio-, Video-, und 3D-Generierung	Rauschen schrittweise in sinnvolle Daten zurückverwandeln	Hohe Bildqualität, stabile Trainingsdynamik, vielseitig	Stable Diffusion, DALL·E 2, Imagen
Small Language Models	Mobile Anwendungen, Edge Computing, Privacy-kritische Bereiche	Kompakte, effiziente KI-Modelle für lokale Ausführung	Datenschutz, geringe Latenz, Ressourceneffizienz, Zugänglichkeit	TinyLlama, Phi-2, Mistral-7B, Gemma

Fazit

Diese vier Architekturformen – Transformer, MoE, Diffusionsmodelle und Small Language Models – prägen heute maßgeblich die Entwicklung und Anwendung moderner KI-Systeme. Sie bilden das Fundament für Anwendungen in Sprachverarbeitung, Codegenerierung, Bildsynthese und multimodaler künstlicher Intelligenz und werden in der Forschung wie in der Industrie intensiv weiterentwickelt. Während Transformer, MoE und Diffusionsmodelle die Grenzen des technisch Machbaren erweitern, sorgen Small Language Models für die praktische Anwendbarkeit von KI-Technologien in ressourcenbeschränkten Umgebungen und tragen so zur Demokratisierung dieser wichtigen Zukunftstechnologie bei.

M18a - Advanced_Prompt_Engineering



Anwendung Generativer KI

Stand: 04.2025

1 | Einführung Prompt-Engineering

Bei Prompt Engineering geht es darum, effektive Eingaben zu erstellen, um Sprachmodelle wie GPT-4 zur Generierung nützlicher und präziser Ergebnisse zu führen. Eine gute Beherrschung des Prompt Engineerings ist entscheidend, um präzise Informationen zu erhalten, überzeugende Kommunikation zu erstellen und fundierte Entscheidungen zu treffen.

Wichtige Aspekte des Prompt Engineerings

Spezifische und vollständige Anweisungen geben

Durch präzise Anweisungen versteht das KI-Modell genau, was benötigt wird. Dies reduziert Unklarheiten und erhöht die Relevanz der Antwort.

Ineffektiver Prompt:

Erkläre die Arten von Versicherungspolicen

Effektiver Prompt:

Als Versicherungsfachperson bitte die wichtigsten Unterschiede zwischen Kapitallebensversicherung und Risikolebensversicherung zusammenfassen, mit Fokus auf Leistungen, Laufzeit und typische Kundenprofile.

Ausnahmesituationen berücksichtigen

Durch Vorhersehen potenzieller Ausnahmen und Anweisungen zum Umgang mit diesen können unvollständige oder irreführende Antworten vermieden werden.

Liste die fünf besten Versicherungsanbieter in Deutschland nach Kundenzufriedenheit auf. Falls aktuelle Daten nicht verfügbar sind, bitte die neuesten Statistiken aus seriösen Quellen nennen und das Jahr der Daten angeben.

Ausgabeformat erklären

Die Definition des gewünschten Ausgabeformats erleichtert die Verwendung der Informationen und kann die Integration mit anderen Dokumenten oder Präsentationen erleichtern.

Erstelle eine Vergleichstabelle für Risikolebensversicherungen und Kapitallebensversicherungen mit den Spalten "Merkmale", "Vorteile" und "Ideal für". Präsentiere die Informationen im Markdown-Format.

Vorschläge zum eigenen Prompt einholen

Durch das Einholen von Feedback können Prompts verfeinert werden, was zu besseren Ergebnissen führt und die Fähigkeiten im Prompt Engineering verbessert.

Ich möchte eine E-Mail an einen Kunden verfassen, die die Vorteile einer Zusatzversicherung für schwere Krankheiten zu seiner Lebensversicherung erklärt. Mein aktueller Prompt ist: 'Schreibe eine E-Mail über die Ergänzung mit einer Zusatzversicherung.' Hast du Vorschläge zur Verbesserung dieses Prompts für detailliertere und überzeugendere Inhalte?

Voreingenommenheit berücksichtigen

Prompt Engineering kann strategisch eingesetzt werden, um Voreingenommenheit in KI-generierten Antworten zu reduzieren, indem Prompts sorgfältig gestaltet werden, die Fairness und Neutralität fördern.

2 | Few-Shot und Chain-of-Thought

Few-Shot-Prompting

Bei Few-Shot-Prompting werden dem Modell einige wenige Beispiele gegeben, um seine Antworten für neue Situationen zu steuern. Diese Methode ist besonders effizient für strukturierte Ergebnisse wie Kategorisierung, Zusammenfassung oder Schätzungen.

Beispiel: Klassifizierung von Antragstellern nach Risiko

Klassifizierte die folgenden Antragsteller basierend auf ihrer Krankengeschichte:

Antragsteller 1: 45 Jahre, Raucher, leichte Hypertonie.

Klassifizierung: Mittleres Risiko.

Antragsteller 2: 30 Jahre, Nichtraucher, keine bedeutende Krankengeschichte.

Klassifizierung: Niedriges Risiko.

Antragsteller 3: 55 Jahre, Diabetes und Herzerkrankungen in der Vorgeschichte.

Klassifizierung: Hohes Risiko.

Antragsteller 4: 50 Jahre, hoher Cholesterinspiegel.

Klassifizierung:

Beispiel: Zusammenfassung von Versicherungspolicen

Fasse diese Versicherungspolicen in einem einzigen Satz zusammen:

Police A: 500.000 € Deckung, 30-jährige Laufzeit, Prämien steigen nach 10 Jahren.

Zusammenfassung: Eine 30-jährige Laufzeitversicherung mit 500.000 € Deckung, wobei die Prämien nach den ersten 10 Jahren steigen.

Police B: 250.000 € Deckung, lebenslange Police mit garantierten Prämien.

Zusammenfassung: Eine lebenslange Police mit 250.000 € Deckung und garantiert stabilen Prämien.

Police C: 350.000 € Deckung, fondsgebundene Lebensversicherung.

Zusammenfassung:

Chain-of-Thought-Prompting

Bei Chain-of-Thought-Prompting wird das KI-Modell dazu angehalten, seine Überlegungen Schritt für Schritt zu artikulieren, bevor es zu einer Schlussfolgerung kommt. Dies verbessert die Genauigkeit der Antworten, indem der Denkprozess transparenter und logischer wird.

Beispiel: Risikobewertung

Beurteile das Risikoniveau für einen 52-jährigen männlichen Antragsteller mit Bluthochdruck

in der Vorgeschichte, der Nichtraucher ist, regelmäßig Sport treibt und eine familiäre Vorgeschichte von Herzerkrankungen hat. Bitte erkläre deine Überlegungen Schritt für Schritt.

3 | Persona- und Rollenmuster

Persona-Muster

Beim Persona-Muster wird der KI eine bestimmte Identität zugewiesen, sodass sie aus dieser Perspektive antworten kann.

Du bist ein erfahrener Versicherungsberater, spezialisiert auf Familienpolicen.
Erkläre einem neuen Kunden die Vorteile einer Kapitallebensversicherung.

Zielgruppen-Persona

Die Zielgruppen-Persona passt die Antwort der KI auf ein bestimmtes Publikum an.

Erkläre die Bedeutung einer Lebensversicherung für einen Hochschulabsolventen, der gerade ins Berufsleben eingestiegen ist.

Rollenspiel-Prompts

Rollenspiel-Prompts binden die KI in ein simuliertes Szenario ein.

Du bist ein Kunde, der an einer Lebensversicherung interessiert ist.
Beginne ein Gespräch mit einem Versicherungsberater, äußere deine Bedenken und frage nach Policienoptionen.

Umgedrehtes Interaktionsmuster

Das umgedrehte Interaktionsmuster kehrt die typischen Rollen um und fordert die KI auf, das Gespräch zu leiten.

Du bist mein Finanzberater. Stelle mir Fragen, um den besten Lebensversicherungsplan für meine Bedürfnisse zu ermitteln.

Spielmuster

Das Spielmuster verwandelt die Interaktion in ein Spiel.

Spielen wir ein Quiz über Begriffe der Lebensversicherung.
Stelle mir fünf Multiple-Choice-Fragen, um mein Wissen zu testen.

4 | Frage- und Prüfmuster

Fragenverfeinerungsmuster

Beim Fragenverfeinerungsmuster beginnt man mit einer allgemeinen Frage und verfeinert sie dann schrittweise.

Was sind die Hauptvorteile einer Lebensversicherung? Konzentriere dich nun speziell darauf,
wie eine Lebensversicherung als Anlagevehikel für die Altersvorsorge dienen kann.

Kognitives Prüfmuster

Das kognitive Prüfmuster fordert die KI auf, Informationen zu verifizieren oder zu überprüfen.

Erkläre, wie fondsgebundene Lebensversicherungspolicen funktionieren.
Verifiziere die Informationen, indem du die wichtigsten Merkmale und damit verbundenen Risiken zusammenfasst.

Muster für Faktencheckliste

Das Faktenchecklisten-Muster stellt Informationen im Format einer Checkliste bereit.

Liste die erforderlichen Schritte zur Umwandlung einer Risikolebensversicherung in eine Kapitallebensversicherung auf und erkläre jeden Schritt kurz.

Vergleichs-/Kontrastaufforderungen

Bei Vergleichs-/Kontrastaufforderungen werden Ähnlichkeiten und Unterschiede zwischen verschiedenen Elementen hervorgehoben.

Vergleiche und kontrastiere fondsgebundene Lebensversicherungen und Universal-Lebensversicherungen in Bezug auf Anlageoptionen und Risiko.

5 | Inhalt- und Struktur-Muster

Vorlagenmuster

Das Vorlagenmuster beinhaltet die Bereitstellung einer vordefinierten Struktur.

Erstelle anhand der folgenden Vorlage eine Zusammenfassung einer Lebensversicherungspolice:

Name der Police:

Art der Police:

Deckungssumme:

Prämiedetails:

Hauptvorteile:

Ausschlüsse:

Zusätzliche Bausteine:

Meta-Sprachmuster

Das Meta-Sprachmuster verwendet Platzhalter innerhalb des Prompts.

Entwerfe eine personalisierte E-Mail an [Kundenname], um ihn über die Vorteile einer Erweiterung seiner aktuellen Lebensversicherung um [Neues Feature/Baustein] zu informieren.

Gliederungserweiterungsmuster

Beim Gliederungserweiterungsmuster wird eine Gliederung zu einer detaillierten Darstellung ausgebaut.

Erweitere die folgenden Punkte zu einem umfassenden Artikel über Risikolebensversicherungen:

Definition der Risikolebensversicherung

Erschwinglichkeit und Einfachheit

Ideale Kandidaten für Risikolebensversicherungen

Vergleich mit Kapitallebensversicherungen

Wie wählt man die richtige Laufzeit

Eingabeaufforderungen zur Inhaltsgenerierung

Eingabeaufforderungen zur Inhaltsgenerierung fordern die KI auf, originelle Inhalte zu erstellen.

Schreibe einen Blogartikel mit dem Titel "Die 5 größten Mythen über Lebensversicherungen widerlegt", der sich an potenzielle Kunden richtet.

6 | Chat- und Reasoning-Modelle

Moderne KI-Modelle lassen sich in zwei Hauptkategorien einteilen: Chat-Modelle und Reasoning-Modelle. Die Herangehensweise beim Prompt Engineering unterscheidet sich je nach Modelltyp erheblich:

Prompting für Chat-Modelle

Chat-Modelle sind für konversationelle Interaktionen optimiert und eignen sich besonders für:

- Natürliche Gespräche
- Kundenservice
- Informationsanfragen
- Kreative Inhalte

Effektive Prompting-Strategien für Chat-Modelle:

1. Konversationeller Ton:

Als Versicherungsberater möchte ich einen Kunden über fondsgebundene Lebensversicherungen informieren. Wie würdest du in einfacher Sprache die Vor- und Nachteile erklären?

2. Rollenbasierte Interaktionen:

Du bist ein freundlicher Versicherungsberater. Ein Kunde fragt, warum er eine Risikolebensversicherung abschließen sollte. Wie würdest du antworten?

3. Klarere Strukturierung der Antworten durch Formatvorgaben:

Erkläre die Unterschiede zwischen Risiko- und Kapitallebensversicherung in einer übersichtlichen Tabelle mit maximal 5 Vergleichspunkten.

Prompting für Reasoning-Modelle

Reasoning-Modelle sind für komplexe Problemlösungen und analytisches Denken optimiert und eignen sich besonders für:

- Logisches Schlussfolgern
- Komplexe Berechnungen
- Mehrstufige Entscheidungsfindung
- Tiefgreifende Analysen

Effektive Prompting-Strategien für Reasoning-Modelle:

1. Aufforderung zur schrittweisen Analyse:

Analysiere die optimale Versicherungsstrategie für einen 45-jährigen Selbständigen mit zwei Kindern, der für seine Altersvorsorge und den Vermögensaufbau plant. Führe deine Überlegungen Schritt für Schritt durch und begründe jede Empfehlung.

2. Explizite Aufforderung zum kritischen Denken:

Beurteile kritisch die Vor- und Nachteile einer fondsgebundenen Rentenversicherung im Vergleich zu direkten ETF-Investments für die langfristige Altersvorsorge. Berücksichtige dabei steuerliche Aspekte, Kostenstrukturen und Flexibilität. Begründe jede Schlussfolgerung und betrachte verschiedene Szenarien.

3. Strukturierung komplexer Gedankengänge:

Entwickle ein Entscheidungsmodell für die Auswahl einer geeigneten Lebensversicherung. Strukturiere deine Analyse in folgende Schritte:

1. Identifizierte die relevanten Kundenfaktoren
2. Analysiere die verfügbaren Versicherungsoptionen
3. Bewerte Vor- und Nachteile jeder Option
4. Entwickle Entscheidungskriterien
5. Empfehle eine begründete Vorgehensweise

4. Verwendung von "Let's think step by step":

Ein 40-jähriger Familienvater möchte eine Risikolebensversicherung abschließen.

Welche Versicherungssumme wäre angemessen? Lass uns Schritt für Schritt überlegen.

Unterschiede in den Prompting-Techniken

- **Detailgrad der Anweisungen:** Reasoning-Modelle profitieren von detaillierteren, strukturierteren Anweisungen, die das Modell durch komplexe Gedankengänge führen.
- **Aufforderung zur Reflexion:** Bei Reasoning-Modellen sollte man explizit zur kritischen Überprüfung von Annahmen und Schlussfolgerungen auffordern:

Nachdem du die Empfehlung für die Versicherungslösung gegeben hast, hinterfrage kritisch deine eigenen Annahmen und diskutiere alternative Szenarien oder mögliche Schwachstellen in deiner Argumentation.

- **Mehrstufige Prompts:** Reasoning-Modelle können besonders gut mit mehrstufigen Prompts umgehen, die das Problem in Teilschritte zerlegen.

Durch die Anpassung der Prompting-Strategien an den jeweiligen Modelltyp können Anwender die Stärken jedes Modells optimal nutzen und präzisere, durchdachtere Antworten erhalten.

7 A | Aufgabe

Die Aufgabestellungen unten bieten Anregungen, Sie können aber auch gerne eine andere Herausforderung angehen.

Few-Shot-Prompting für Kundenfeedback-Analyse

Entwickeln Sie einen Few-Shot-Prompt, um Kundenfeedback zu kategorisieren.

Aufgabenstellung:

1. Formulieren Sie einen Prompt, der ein Sprachmodell anweist, Kundenfeedback zu einem KI-gestützten Produkt in folgende Kategorien einzurichten: "Benutzerfreundlichkeit", "Funktionsumfang", "Technische Probleme" und "Sonstiges".
2. Integrieren Sie mindestens drei Beispiele (Few-Shot-Ansatz), um dem Modell die gewünschte Struktur zu demonstrieren.

3. Testen Sie Ihren Prompt mit fünf fiktiven Kundenfeedbacks und bewerten Sie die Kategorisierungsgenauigkeit.

Erweiterte Anforderung:

Fügen Sie dem Prompt eine zusätzliche Anweisung hinzu, die das Modell auffordert, für jedes Feedback auch eine Stimmungsbewertung (positiv, neutral, negativ) abzugeben.

Chain-of-Thought für komplexe ML-Modellauswahl

Anwendung des Chain-of-Thought-Ansatzes für eine fundierte ML-Modellempfehlung.

Aufgabenstellung:

1. Erstellen Sie einen Prompt, der ein Sprachmodell dazu anleitet, für ein bestimmtes Datenproblem einen geeigneten ML-Algorithmus zu empfehlen.
2. Der Prompt soll das Modell explizit anweisen, seinen Entscheidungsprozess Schritt für Schritt darzulegen (Chain-of-Thought).
3. Definieren Sie ein komplexes Szenario, z.B.: "Ein Online-Händler möchte Kundensegmente identifizieren, Kaufverhalten vorhersagen und Produktempfehlungen optimieren."
4. Der Prompt soll das Modell auffordern, verschiedene Ansätze zu vergleichen und eine begründete Empfehlung auszusprechen.

Erweiterte Anforderung:

Fügen Sie eine zusätzliche Anweisung hinzu, dass das Modell auch potenzielle Fallstricke oder Herausforderungen bei der Implementierung des empfohlenen Algorithmus identifizieren soll.

Persona-basierte Dokumentation für unterschiedliche Zielgruppen

Erstellung von zielgruppenspezifischen Erklärungen eines ML-Konzepts mit Hilfe von Persona-Prompting.

Aufgabenstellung:

1. Wählen Sie ein komplexes Konzept des maschinellen Lernens (z.B. neuronale Netze, Ensemble-Methoden oder Transfer Learning).
2. Entwickeln Sie drei verschiedene Persona-Prompts für folgende Zielgruppen:
 - Technischer Entscheider ohne ML-Hintergrund
 - Data Scientist mit grundlegenden ML-Kenntnissen
 - Softwareentwickler, der ML-Komponenten integrieren möchte
3. Jeder Prompt soll das Modell anweisen, das Konzept auf eine für die jeweilige Persona geeignete Weise zu erklären.

4. Vergleichen Sie die Ergebnisse und analysieren Sie, wie sich die Erklärungen in Bezug auf Fachsprache, Detailtiefe und Praxisbezug unterscheiden.

Erweiterte Anforderung:

Erweitern Sie jeden Prompt um die Anweisung, ein konkretes Anwendungsbeispiel zu liefern, das für die jeweilige Zielgruppe besonders relevant ist, und fügen Sie eine Aufforderung hinzu, typische Missverständnisse zu adressieren, die bei der jeweiligen Zielgruppe auftreten könnten.

M18b - Langchain Hub



Anwendung Generativer KI

Stand: 05.2025

1 | Einführung

Der LangChain Hub ist eine zentrale Plattform für die Verwaltung, Versionierung und gemeinsame Nutzung von Prompts in LangChain-Anwendungen. Diese Funktion ermöglicht es Entwicklern, ihre Prompts zu standardisieren, zu optimieren und effizient in Teams zusammenzuarbeiten.

2 | Hauptfunktionen

2.1 Zentrale Prompt-Verwaltung

Der LangChain Hub ermöglicht die zentrale Speicherung und Verwaltung von Prompts, was besonders in größeren Teams oder Projekten vorteilhaft ist, um Konsistenz zu gewährleisten.

```
from langchain_hub import pull

# Einen bestimmten Prompt aus dem Hub abrufen
qa_prompt = pull("yourusername/qa_prompt")
```

2.2 Prompt-Versionierung

Ähnlich wie bei Git können Prompts versioniert werden, um Änderungen nachzuverfolgen und bei Bedarf zu früheren Versionen zurückzukehren.

```
# Eine bestimmte Version eines Prompts abrufen
qa_prompt_v2 = pull("yourusername/qa_prompt:v2")
```

2.3 Prompt-Sharing und Kollaboration

Teams können Prompts einfach teilen und gemeinsam daran arbeiten, was die Zusammenarbeit und Wiederverwendung von bewährten Prompt-Mustern fördert.

```
from langchain_hub import push
from langchain.prompts import PromptTemplate

# Einen neuen Prompt erstellen
my_prompt = PromptTemplate.from_template(
    "Analysiere folgendes Dataset: {dataset_description}"
)

# Prompt zum Hub hochladen
push(my_prompt, "yourusername/data_analysis_prompt", new_version=True)
```

2.4 Prompt-Bibliotheken und Community

Der Hub bietet Zugang zu einer wachsenden Sammlung von Community-erstellten Prompts für verschiedene Anwendungsfälle, die als Ausgangspunkt oder Inspiration dienen können.

```
# Einen Community-Prompt für Zusammenfassungen verwenden
summarization_prompt = pull("community/summarization:latest")
```

3 | Integration in LangChain-Anwendungen

Die Hub-Prompts lassen sich nahtlos in bestehende LangChain-Anwendungen integrieren:

```
from langchain_hub import pull
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI

# Prompt aus dem Hub abrufen
classification_prompt = pull("myteam/sentiment_classification:v3")

# LLM definieren
llm = ChatOpenAI(model="gpt-4")

# Chain erstellen und ausführen
chain = LLMChain(llm=llm, prompt=classification_prompt)
result = chain.run(text="Das neue Produkt übertrifft alle meine Erwartungen!")
```

4 | Best Practices

4.1 Standardisierte Benennung

Entwickeln Sie ein konsistentes Benennungsschema für Ihre Prompts, das den Anwendungsfall und die Version klar kommuniziert.

```
teamname/anwendungsfall_prompttyp:version
```

Beispiel: dataengineers/customer_feedback_classification:v2

4.2 Dokumentation von Prompts

Fügen Sie jedem Prompt Metadaten und Dokumentation hinzu, um seinen Zweck, die erwarteten Eingaben und das Ausgabeformat zu beschreiben.

```
from langchain.prompts import PromptTemplate
from langchain_hub import push

prompt = PromptTemplate(
    template="Klassifiziere das folgende Kundenfeedback: {feedback}",
    input_variables=["feedback"],
    metadata={
        "description": "Prompt zur Klassifizierung von Kundenfeedback in Kategorien",
        "expected_output": "Eine der Kategorien: Positiv, Neutral, Negativ",
        "example_inputs": {"feedback": "Ihr Produkt funktioniert großartig!"}
    }
)

push(prompt, "myteam/feedback_classification", new_version=True)
```

4.3 A/B-Tests und Prompt-Experimente

Nutzen Sie den Hub für A/B-Tests verschiedener Prompt-Formulierungen, um die effektivsten Prompts für Ihre Anwendungsfälle zu identifizieren.

```
# Variante A abrufen und testen
prompt_a = pull("myteam/product_summary:v1")
# Variante B abrufen und testen
prompt_b = pull("myteam/product_summary:experimental")

# Vergleich der Ergebnisse beider Prompts
```

5 | Fortgeschrittene Anwendungen

5.1 Prompt-Chaining mit dem Hub

Komplexe Reasoning-Ketten können durch die Kombination spezialisierter Prompts aus dem Hub erstellt werden.

```
from langchain_hub import pull
from langchain.chains import SequentialChain
from langchain.chains import LLMChain
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4")

# Spezialisierte Prompts aus dem Hub abrufen
extraction_prompt = pull("myteam/data_extraction:v2")
analysis_prompt = pull("myteam/data_analysis:v1")
recommendation_prompt = pull("myteam/recommendations:v3")

# Chains erstellen
extraction_chain = LLMChain(llm=llm, prompt=extraction_prompt,
output_key="extracted_data")
analysis_chain = LLMChain(llm=llm, prompt=analysis_prompt,
output_key="analysis_results")
recommendation_chain = LLMChain(llm=llm, prompt=recommendation_prompt,
output_key="recommendations")

# Sequentielle Chain erstellen
sequential_chain = SequentialChain(
    chains=[extraction_chain, analysis_chain, recommendation_chain],
    input_variables=["raw_data"],
    output_variables=["recommendations"],
    verbose=True
)

results = sequential_chain({"raw_data": customer_data})
```

5.2 Dynamische Prompt-Auswahl

Implementieren Sie eine dynamische Auswahl von Prompts basierend auf Eingabedaten oder Kontextinformationen.

```
from langchain_hub import pull

def select_prompt(input_data):
    if "technical" in input_data.lower():
        return pull("myteam/technical_support:latest")
```

```

    elif "billing" in input_data.lower():
        return pull("myteam/billing_support:latest")
    else:
        return pull("myteam/general_support:latest")

user_query = "Ich habe ein technisches Problem mit der API-Integration."
selected_prompt = select_prompt(user_query)
# Verwenden des ausgewählten Prompts für die Antwortgenerierung

```

6 | Hub-Prompts mit Reasoning-Modellen

Für komplexe Reasoning-Aufgaben können spezialisierte Prompts aus dem Hub besonders wertvoll sein:

```

from langchain_hub import pull
from langchain_openai import ChatOpenAI
from langchain.chains import LLMChain

# Reasoning-Prompt aus dem Hub abrufen
reasoning_prompt = pull("myteam/step_by_step_reasoning:v2")

# Modell mit erweitertem Reasoning verwenden
reasoning_model = ChatOpenAI(model="gpt-4", temperature=0.2)

# Chain erstellen
reasoning_chain = LLMChain(llm=reasoning_model, prompt=reasoning_prompt)

# Komplexes Problem lösen mit strukturiertem Reasoning
problem = """
Ein Datensatz enthält 5000 Kundenrezensionen. 20% der Rezensionen enthalten
technische Fragen,
35% beziehen sich auf Preisgestaltung, und der Rest betrifft Lieferprobleme.
Welcher ML-Ansatz wäre optimal für die Klassifizierung dieser Daten und
warum?
"""

analysis = reasoning_chain.run(problem=problem)

```

7 | Template Repositories

LangChain Hub bietet vorgefertigte Template Repositories, die schnell in eigenen Projekten eingesetzt werden können:

```

# Community-Template für Few-Shot-Prompting abrufen
few_shot_template = pull("community/few_shot_classification:latest")

```

```
# Anpassen mit eigenen Beispielen
custom_few_shot = few_shot_template.format(
    examples=[
        {"input": "Das System reagiert nicht.", "output": "Technisches Problem"}, 
        {"input": "Die Kosten sind zu hoch.", "output": "Preisbeschwerde"}, 
        {"input": "Tolle Funktionen!", "output": "Positives Feedback"}]
)
```

8 | Praktische Anwendungsfälle

8.1 Mehrsprachige Prompts

Verwalten Sie Prompts in verschiedenen Sprachen für internationale Anwendungen:

```
# Sprachspezifische Prompts abrufen
de_prompt = pull("myteam/customer_service:de")
en_prompt = pull("myteam/customer_service:en")
es_prompt = pull("myteam/customer_service:es")

# Dynamische Sprachauswahl
def get_language_prompt(language_code):
    return pull(f"myteam/customer_service:{language_code}")
```

8.2 Domänenspezifische Prompts

Erstellen Sie spezialisierte Sammlungen von Prompts für verschiedene Geschäftsbereiche:

```
# Prompts für verschiedene ML-Aufgaben
classification_prompt = pull("ml_team/text_classification:v2")
sentiment_prompt = pull("ml_team/sentiment_analysis:latest")
entity_extraction_prompt = pull("ml_team/entity_extraction:stable")
```

9 | Zukunft des Prompt-Engineerings

Der LangChain Hub entwickelt sich kontinuierlich weiter und bietet Entwicklern leistungsstarke Werkzeuge für das Prompt-Engineering:

- **Automatisierte Prompt-Optimierung:** Zukünftige Versionen werden voraussichtlich Tools für automatisierte A/B-Tests und Optimierung von Prompts enthalten.
- **Erweiterte Prompt-Templates:** Komplexere Template-Strukturen mit bedingter Logik und dynamischer Formatierung.
- **KI-gestützte Prompt-Vorschläge:** Intelligente Vorschläge zur Verbesserung von Prompts basierend auf Erfolgsmetriken.

Der LangChain Hub bietet eine robuste Infrastruktur für die Verwaltung und Optimierung von Prompts, die besonders bei der Zusammenarbeit in Teams und bei der Entwicklung komplexer KI-Anwendungen wertvoll ist. Durch die Standardisierung und Versionierung von Prompts können Entwickler konsistenter und effektivere Ergebnisse erzielen.

M19a - EU AI Act



Anwendung Generativer KI

Stand: 04.2025

1 Einleitung & Zielsetzung

Der EU AI Act (Verordnung (EU) 2024/1689) ist das weltweit erste umfassende Gesetz zur Regulierung von Künstlicher Intelligenz (KI). Ziel ist es, ein vertrauenswürdiges KI-Ökosystem in Europa zu schaffen, das Sicherheit, Grundrechte und europäische Werte wahrt, gleichzeitig aber Innovation und Investitionen fördert. Kern des Gesetzes ist ein risikobasierter Ansatz, der KI-Systeme in vier Risikoklassen einteilt: inakzeptabel, hoch, begrenzt und minimal. Für jede Risikoklasse gelten abgestufte Anforderungen. Der Gesetzgebungsprozess wurde durch globale politische Entwicklungen, technologische Dynamik und zunehmende gesellschaftliche Debatten beschleunigt. Der AI Act ist Teil der Digitalstrategie der EU und steht im Kontext internationaler Regulierungsbemühungen.

2 Umsetzung & Anwendbarkeit

Das Gesetz trat am 1. August 2024 in Kraft. Die Anwendung erfolgt gestaffelt bis 2030. Erste Verbote (z. B. manipulative KI, Social Scoring) gelten seit Februar 2025. Für Hochrisiko-KI-Systeme sind lange Übergangsfristen vorgesehen. Mitgliedstaaten müssen zuständige Behörden benennen, Sandkästen einrichten und über notwendige Ressourcen verfügen. Bisher zeigen sich Unterschiede im Umsetzungsstand. Auf EU-Ebene übernimmt das European AI Office zentrale Aufgaben, insbesondere bei General Purpose AI (GPAI). Die nationale Umsetzung variiert stark, was Risiken für Fragmentierung birgt. Sandkästen sollen Innovation fördern, insbesondere bei KMU.

Wichtige Meilensteine bei der Umsetzung der KI-Regulierung



Made with Napkin

3 Risikoklassen im Detail

Der risikobasierte Ansatz ist das zentrale Steuerungsinstrument des EU AI Acts. KI-Systeme werden in vier Klassen eingestuft, je nach potenzieller Auswirkung auf Sicherheit, Grundrechte und gesellschaftliche Werte:

- **Inakzeptables Risiko:** KI-Systeme, die gegen Werte der EU verstößen oder erhebliche Risiken für Individuen bergen, sind verboten. Beispiele: Social Scoring durch Behörden, Echtzeit-Gesichtserkennung im öffentlichen Raum (außer in engen Ausnahmefällen).
- **Hohes Risiko:** Systeme in sensiblen Bereichen wie Bildung, Justiz, Strafverfolgung, Gesundheit oder kritischen Infrastrukturen. Sie unterliegen umfangreichen Anforderungen wie Risikomanagement, Dokumentationspflichten, menschlicher Aufsicht und hoher Datenqualität. Diese Systeme müssen vor dem Einsatz ein Konformitätsbewertungsverfahren durchlaufen.
- **Begrenztes Risiko:** Systeme mit potenziellen Transparenzrisiken, wie Chatbots oder Deepfake-Generatoren. Hier bestehen Informationspflichten gegenüber Nutzer:innen, z. B. Kennzeichnung, dass Inhalte KI-generiert sind.
- **Minimales Risiko:** Die Mehrheit aller KI-Anwendungen (z. B. Spamfilter, Empfehlungssysteme für Streaming-Dienste) fällt in diese Kategorie. Es gelten keine verpflichtenden Anforderungen, freiwillige Verhaltenskodizes werden jedoch gefördert.

Diese Einteilung ermöglicht eine differenzierte Regulierung: Statt alle KI-Technologien über einen Kamm zu scheren, werden Anforderungen gezielt dort angesetzt, wo die potenziellen Gefahren für Individuum und Gesellschaft am größten sind.

KI-Risikokategorien



Made with Napkin

4 Herausforderungen & Kritikpunkte

Zentrale Herausforderungen:

- **Unklare Begriffsdefinitionen:** Begriffe wie "KI-System", "systemisches Risiko" oder "enge verfahrenstechnische Aufgabe" sind interpretationsbedürftig.
- **Risikoklassifizierung:** Die Einstufung als Hochrisiko-System ist teils unklar. Die Ausnahmeregelung nach Art. 6(3) wird als potenzielles Schlupfloch kritisiert.
- **GPAI-Regulierung:** Neue Regeln für Basismodelle wie GPT-4 sind komplex. Kritik gibt es an Schwellenwerten, Transparenzpflichten und Umsetzbarkeit.
- **Grundrechtlücken:** Zivilgesellschaftliche Organisationen bemängeln Ausnahmen für Sicherheitsbehörden und unzureichenden Schutz z. B. im Migrationskontext.
- **Durchsetzbarkeit:** Behörden mangelt es oft an Ressourcen und Expertise. Die komplexe Governance-Struktur erhöht die Anforderungen an Koordination.
- **Wirtschaftliche Sorgen:** Unternehmen kritisieren hohe Compliance-Kosten, vage Formulierungen und potenzielle Innovationshemmnisse.

5 Potenziale & Chancen

Trotz der Kritik birgt der AI Act bedeutende Potenziale:

- **Rechtssicherheit:** Einheitliche Regeln erleichtern die Planung und Investition für Unternehmen.

- **Marktchancen:** Es entsteht ein Markt für vertrauenswürdige KI-Produkte, der als Qualitätsmerkmal dienen kann.
- **Neue Dienstleistungen:** Nachfrage nach Compliance-Tools, Auditierung, Governance-Frameworks und Ethikberatung steigt.
- **Standardisierung:** Harmonisierte technische Normen könnten auch international prägend wirken.
- **Weltweite Vorreiterrolle:** Der "Brussels Effect" – die globale Strahlkraft europäischer Regulierung – könnte KI-Regelsetzung weltweit beeinflussen.

6 Best Practices & Empfehlungen

Empfohlene Maßnahmen für die Umsetzung:

- **Frühzeitige Systeminventur:** Unternehmen sollten ihre KI-Systeme erfassen und frühzeitig risikobasiert einstufen.
- **Governance-Modelle etablieren:** Aufbau interner Compliance-Strukturen, idealerweise unter Einbeziehung bestehender DSGVO-Frameworks.
- **Transparenz und Dokumentation:** Verfahrensdokumentation, Modellkarten, Bias-Tests und menschliche Aufsicht sind essenziell.
- **Nutzung von Sandkästen:** Besonders für KMU eine Möglichkeit, Innovation rechtskonform zu testen.
- **Offene Kommunikation:** Stakeholder-Dialoge und partizipative Gestaltung erhöhen Akzeptanz und Wirksamkeit.

7 Ausblick

Der langfristige Erfolg des EU AI Acts wird davon abhängen, ob es gelingt, die Balance zwischen Regulierung und Innovationsförderung zu halten. Die praktische Durchsetzbarkeit, die Kohärenz mit anderen EU-Gesetzen (z. B. DSGVO, Data Act, DSA) sowie die internationale Anschlussfähigkeit werden entscheidend sein. Wichtig wird auch sein, wie flexibel der AI Act auf technologische Entwicklungen reagieren kann und ob er Vertrauen in KI langfristig stärkt.

M19b - Ethik und Generative KI



Anwendung Generativer KI

Stand: 04.2025

1 Ethische Dimensionen

Definition & Abgrenzung:

- Generative KI (GenAI) erzeugt neue Inhalte (Texte, Bilder, Musik, Code) auf Basis gelernter Muster und stellt damit eine kreative, wenn auch nicht bewusste, Nachbildung menschlicher Ausdrucksformen dar.
- Abgrenzung zu anderen KI-Typen:
 - *Analytische oder prädiktive KI* analysiert bestehende Daten, um Vorhersagen zu treffen (z. B. Kredit-Scoring). Ethikfragen betreffen hier v. a. Fairness und Nachvollziehbarkeit der Entscheidungskriterien.
 - *Regelbasierte oder symbolische KI* folgt festen, menschenkodierten Entscheidungsregeln (z. B. Expertensysteme) und ist meist gut erklärbar.
 - *AGI* (Artificial General Intelligence) beschreibt eine hypothetische KI mit menschenähnlicher, allgemeiner Intelligenz. Sie existiert aktuell nicht, ist aber zentrales Thema in der KI-Ethik-Forschung.

Die Abgrenzung ist wichtig, da **generative KI besonders intransparente, kreative Outputs erzeugt**, was neue ethische Fragestellungen aufwirft – etwa zur Originalität, Verantwortung und Manipulationsgefahr.

Zentrale ethische Prinzipien:

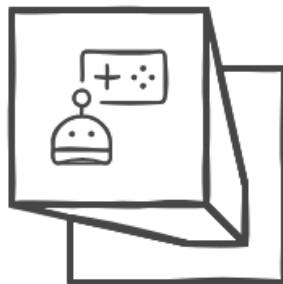
- **Verantwortung:** Wer haftet bei Fehlentscheidungen? Eine klare juristische und ethische Zuweisung ist meist schwierig.
- **Fairness:** Gefahr der Reproduktion sozialer Ungleichheiten durch Daten-Bias; bedarf systematischer Überprüfung.
- **Transparenz:** Undurchschaubarkeit der Modelle verhindert Vertrauen und kontrollierte Anwendung.
- **Datenschutz:** Besonders problematisch bei sensiblen Daten wie Gesundheitsdaten oder intimen Nutzereingaben.

- **Autonomie:** Nutzer:innen dürfen nicht entmündigt werden; Systeme müssen überschreibbar bleiben.
- **Sicherheit:** Technisch wie gesellschaftlich müssen Risiken minimiert werden, etwa durch Missbrauchsprävention.
- **Akteure:** Technologieunternehmen (oft marktgetrieben), Forschung (wissensgetrieben), Politik (regulierend), Zivilgesellschaft (wertorientiert), Nutzer:innen (praktisch-orientiert) prägen gemeinsam das öffentliche Verständnis und die Entwicklungspfade von KI.

Ethische Prinzipien in der KI

Autonomie

Autonomie ist einfach umzusetzen, hat aber großen Einfluss auf Nutzer.



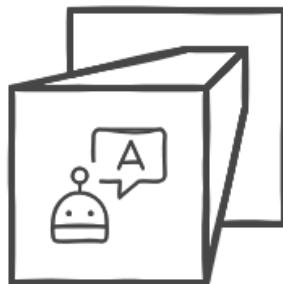
Datenschutz

Datenschutz erfordert komplexe Lösungen mit hoher gesellschaftlicher Wirkung.



Transparenz

Transparenz ist leicht zu erreichen, aber mit geringer Wirkung.



Verantwortung

Verantwortung ist komplex, aber mit begrenzter direkter Auswirkung.



Made with Napkin

2 Rahmenwerke & Praxis

Regulatorische Grundlagen:

- Der **EU AI Act** ist das weltweit erste umfassende Gesetz zur Regulierung von KI und unterteilt Systeme in vier Risikokategorien. Besonders generative KI mit hohem Einfluss auf Meinungsbildung und Kreativbereiche steht dabei unter besonderer Beobachtung.
- **OECD- und UNESCO-Richtlinien** setzen wichtige normative Standards, die Fairness, Erklärbarkeit und Rechenschaftspflicht als universelle Prinzipien formulieren.

Umsetzung in der Industrie:

- Unternehmen wie OpenAI, Google und Meta haben ethische Leitlinien entwickelt, die Aspekte wie Moderation, Red Teaming und Prompt-Engineering einschließen.

- Tools wie SynthID oder Wasserzeichen-Lösungen fördern Transparenz und Fälschungsschutz.
- Trotzdem bleibt die Selbstverpflichtung oft hinter regulatorischen Anforderungen zurück.

Organisatorische Umsetzung:

- Interne Ethikboards, Compliance-Beauftragte und Prozesse zur Ethikfolgenabschätzung gewinnen an Bedeutung.
- Wichtig ist nicht nur die Existenz, sondern die Integration ethischer Reflexion in agile Entwicklungsprozesse.

Rolle von Bildung & Forschung:

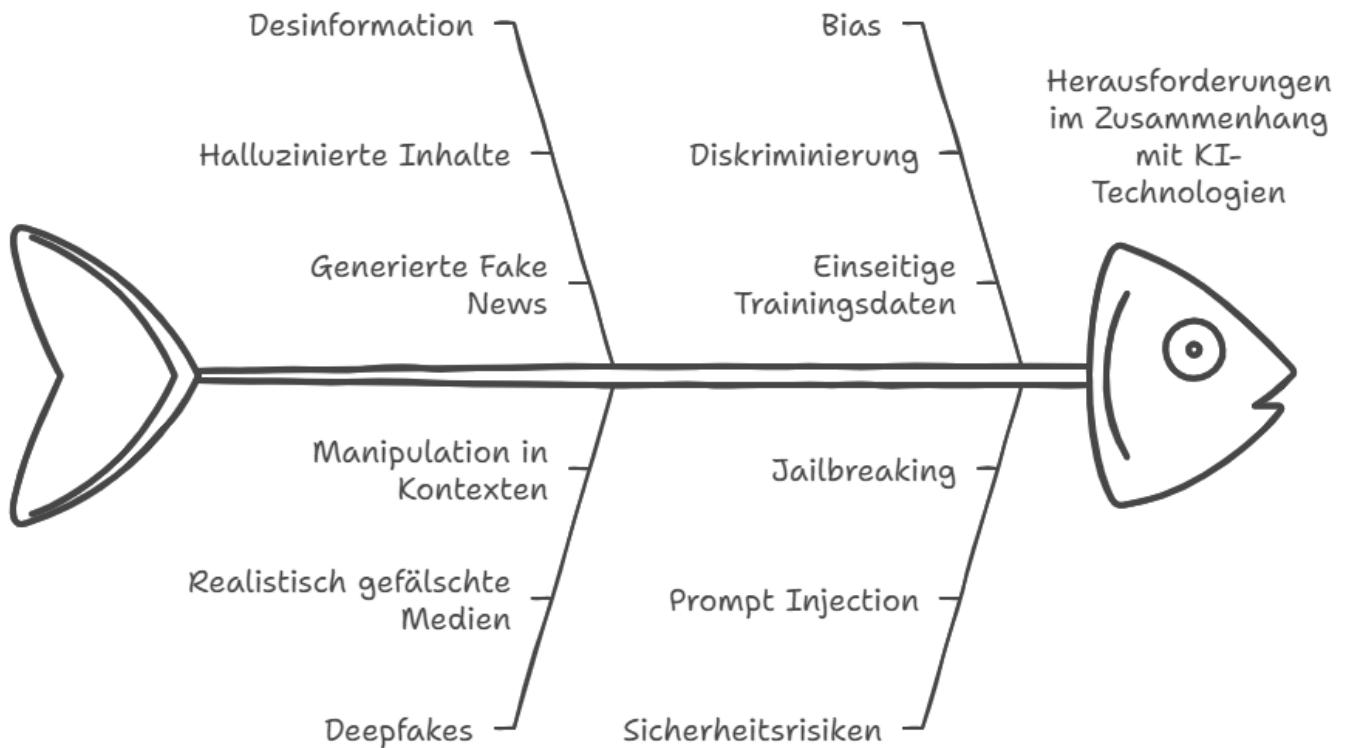
- Hochschulen und Ausbildungsinstitutionen entwickeln spezialisierte Curricula zu KI-Ethik.
- Praxisnahe Formate wie Fallanalysen, Planspiele oder interdisziplinäre Projektarbeit sind besonders wirksam.

3 Risiken & Fehlerquellen

Kernrisiken:

- **Desinformation:** Halluzinierte Inhalte und generierte Fake News untergraben Vertrauen in Informationen.
- **Deepfakes:** Realistisch gefälschte Medien können zur Manipulation in politischen oder wirtschaftlichen Kontexten führen.
- **Bias:** Diskriminierung aufgrund einseitiger Trainingsdaten ist ein zentrales Problem.
- **Rechtsunsicherheit:** Unklarheiten bei Urheberrecht und Datenschutz hemmen klare Verantwortungszuweisung.
- **Sicherheitsrisiken:** Prompt Injection, Jailbreaking oder Training mit vergifteten Daten sind reale Angriffsvektoren.

Risiken im Zusammenhang mit KI



Made with Napkin

Ethische Spannungsfelder:

- Innovation vs. Regulierung
- Transparenz vs. Datenschutz oder geistiges Eigentum
- Open Source vs. Missbrauch
- Automatisierung vs. Arbeitsplatzverlust

Fehlerquellen im Lebenszyklus:

- *Daten*: Verzerrung durch schlechte oder einseitige Quellen.
- *Modell*: Fehlende Robustheit, Halluzinationen, Black-Box-Verhalten.
- *Prozess*: Mangelnde Tests, nicht-diverse Teams, unklare Verantwortlichkeiten.
- *Nutzung*: Missbrauch für Desinformation, unreflektierte Übernahme von KI-Outputs.

Verantwortung:

- Verteilte Verantwortung in Wertschöpfungsketten erschwert Haftung und Governance.
- Neue regulatorische Ansätze wie der AI Act definieren Rollen und Pflichten neu.

4 Chancen & Potenziale

Gesellschaftlicher Mehrwert:

- **Bildung:** Automatisierte Lernpfade, intelligente Nachhilfe und adaptive Lernumgebungen.
- **Barrierefreiheit:** Text-zu-Sprache, visuelle Erkennung, einfache Sprache für mehr Teilhabe.
- **Wissenschaft:** Hypothesengenerierung, Datenanalyse, automatisierte Literaturauswertung.
- **Kreativität:** Unterstützung für künstlerische Prozesse und Demokratisierung kreativer Mittel.
- **Wirtschaft:** Automatisierung von Routineaufgaben, Entlastung von Fachkräften.
- **Nachhaltigkeit:** Umweltmonitoring, Klimamodellierung, Analyse von ESG-Daten.

Ethics by Design:

- Ethische Reflexion bereits in der Designphase einbetten.
- Interdisziplinäre Teams, Impact Assessments und diverse Perspektiven sind zentral.

Gemeinwohlorientierte KI:

- Open-Source-Modelle, öffentliche KI-Infrastrukturen (z. B. EU AI Factories), transparente Standards.
- Ziel: Technologische Souveränität und gerechter Zugang zu KI.

5 Best Practices

Technische Maßnahmen:

- **Explainable AI (XAI):** Methoden wie LIME, SHAP, RAG-basierte Erklärungen fördern Transparenz.
- **Bias-Mitigation:** In allen Phasen (Pre-, In-, Postprocessing), begleitet von Fairness-Audits.
- **Sicherheit:** Red Teaming, Input-Validierung, Zugriffskontrollen, Inhaltsfilter.
- **Datenschutz:** Anonymisierung, Pseudonymisierung, differenzielle Privatsphäre.
- **Transparenz:** Dokumentation, Wasserzeichen, klare Kommunikation der KI-Nutzung.

Organisatorische Strategien:

- Etablierung klarer Verantwortlichkeiten für KI im Unternehmen.
- Entwicklung und Pflege von KI-Ethik-Kodizes.
- Integration von ethischer Reflexion in Produktentwicklung und -bewertung.

Bildungs- und Schulungsinitiativen:

- Schulung für Entwickler (z. B. Bias, Datenschutz, XAI).
- Aufklärung von Anwendern über Grenzen, Risiken und verantwortungsvollen Umgang mit KI.

Rahmenwerke und Tools:

- Nutzung internationaler Standards (z. B. NIST AI RMF, EU AI Act, ISO 42001).
- Checklisten für ethisch orientierte Entwicklung und Deployment.

M20 - KI-Challenge



Anwendung Generativer KI

KI-Challenge

1 | Überblick KI-Challenge

Die KI-Challenge dient als praktische Anwendung und Integration der in den Kursmodulen erlernten Konzepte und Techniken. Ziel ist es, eine funktionsfähige KI-Anwendung zu entwickeln, die mehrere Aspekte der generativen KI kombiniert und einen praktischen Nutzen bietet.

1.1 Lernziele

- Integration mehrerer Technologien aus den Basismodulen
- Praktische Anwendung von LLM-basierten Lösungen
- Entwicklung einer vollständigen End-to-End-Anwendung
- Präsentation und Dokumentation der eigenen Lösung

1.2 Voraussetzungen

- Abschluss der Basismodule (Module 1-12)
- Module aus dem Bereich Erweiterung
- Kenntnisse in Python und LangChain
- Zugriff auf API-Keys (OpenAI, Hugging Face)
- Grundlegende Vertrautheit mit Gradio für UI-Entwicklung

2 | Projektoptionen

Zur Auswahl stehen vier verschiedene Projekttypen, die jeweils unterschiedliche Aspekte der generativen KI betonen. Wählen Sie eine Option aus oder kombinieren Sie Elemente verschiedener Optionen.

2.1 Dokumentenanalyse-Assistent

Beschreibung: Ein System, das PDF-Dokumente, Word-Dateien oder Textdateien verarbeitet und intelligente Zusammenfassungen, Antworten auf Fragen oder strukturierte Analysen liefert.

Kernelemente:

- RAG-Pipeline mit Vektordatenbank (ChromaDB)
- Dokumentenverarbeitung und Chunking
- Intelligentes Prompting für die Analyse
- Benutzeroberfläche mit Gradio

Erwartete Module:

- Modul 4 (LangChain)
- Modul 7 (Output Parser)
- Modul 8 (RAG)
- Modul 11 (Gradio)

2.2 Multimodaler Assistent

Beschreibung: Ein Assistent, der Bild, Text und optional Audio verarbeiten kann, um komplexe Aufgaben zu erfüllen oder Informationen zu analysieren.

Kernelemente:

- Integration von Bild- und Texterkennung
- Multimodale Prompt-Strategien
- Kontextbewusste Antworten
- Interaktive Benutzeroberfläche

Erwartete Module:

- Modul 5 (LLMs und Transformer)
- Modul 6 (Chat und Memory)
- Modul 9 (Multimodal Bild)
- Modul 14 (optional: Multimodal Audio)

2.3 Agentenbasiertes System

Beschreibung: Ein System mit mehreren spezialisierten Agenten, die zusammenarbeiten, um komplexe Aufgaben zu lösen oder Workflow-Prozesse zu automatisieren.

Kernelemente:

- Multi-Agenten-Architektur
- Werkzeugintegration (APIs, Datenbanken)
- Planung und Zielverfolgung
- Benutzerinteraktion und Transparenz

Erwartete Module:

- Modul 3 (Codieren mit GenAI)
- Modul 10 (Agents)
- Modul 12 (Lokale Modelle)
- Modul 18 (optional: Advanced Prompt Engineering)

2.4 Domänen Fachexperte

Beschreibung: Ein spezialisierter Assistent für ein bestimmtes Fachgebiet (z.B. Recht, Medizin, Finanzen, Marketing), der tiefgreifendes Fachwissen bereitstellt und domänenspezifische Aufgaben löst.

Kernelemente:

- Fachspezifische Wissensdatenbank
- Spezialisierte Prompts und Output-Strukturen
- Benutzeroberfläche für Fachexperten
- Optional: Feinabstimmung eines bestehenden Modells

Erwartete Module:

- Modul 2 (Grundlagen Modellansteuerung)
- Modul 8 (RAG)
- Modul 16 (optional: Fine-Tuning)
- Modul 19 (optional: EU AI Act/Ethik)

3 | Projekt-Setup

Hier finden Sie den Code für das grundlegende Setup Ihres Projekts, ähnlich wie in den Kursmodulen.

```
#@title
#@markdown <p><font size="4" color='green'> Colab-Umfeld</font> </br></p>
# Installierte Python Version
import sys
print(f"Python Version: ",sys.version)
# Installierte LangChain Bibliotheken
```

```

print()
print("Installierte LangChain Bibliotheken:")

!pip list | grep '^langchain'
# Unterdrückt die "DeprecationWarning" von LangChain für die Memory-
Funktionen
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.filterwarnings("ignore", category=UserWarning,
module="langsmith.client")

```

```

#@title
#@markdown <p><font size="4" color='green'> SetUp API-Keys
(setup_api_keys)</font> </br></p>
def setup_api_keys():
    """Konfiguriert alle benötigten API-Keys aus Google Colab userdata"""
    from google.colab import userdata
    import os
    from os import environ

    # Dictionary der benötigten API-Keys
    keys = {
        'OPENAI_API_KEY': 'OPENAI_API_KEY',
        'HF_TOKEN': 'HF_TOKEN',
        # Weitere Keys bei Bedarf
    }

    # Keys in Umgebungsvariablen setzen
    for env_var, key_name in keys.items():
        environ[env_var] = userdata.get(key_name)

    return {k: environ[k] for k in keys.keys()}

# Verwendung
all_keys = setup_api_keys()
# Bei Bedarf einzelne Keys direkt zugreifen
# WEATHER_API_KEY = all_keys['WEATHER_API_KEY']

```

4 | Projektstruktur

Ein erfolgreiches Abschlussprojekt sollte folgende Komponenten enthalten:

4.1 Problemdefinition und Anforderungen

- Klare Beschreibung des Problems oder der Aufgabe

- Definition der Anforderungen und Erfolgskriterien
- Abgrenzung des Projektumfangs

4.2 Datenstrukturen und Modellauswahl

- Auswahl und Begründung der verwendeten Modelle
- Datenstrukturen und Datenvorbereitung
- Embedding-Strategien (bei RAG-Anwendungen)

4.3 Kernfunktionalität

- LangChain-Pipelines oder -Ketten
- Prompt-Engineering und Templates
- Integration mit externen APIs oder Datenquellen

4.4 Benutzeroberfläche und Interaktion

- Gradio-Interface für die Interaktion
- Benutzerführung und Feedback
- Fehlerbehandlung und Robustheit

4.5 Evaluation und Tests

- Testfälle für verschiedene Szenarien
- Bewertung der Modellleistung
- Benutzerfeedback und Verbesserungen

4.6 Dokumentation und Präsentation

- Projektdokumentation (Markdown oder PDF)
- Code-Kommentare und Erklärungen
- Präsentation der Ergebnisse

5 | Bewertungskriterien

Die KI-Challenge wird anhand folgender Kriterien bewertet:

Kriterium	Beschreibung	Gewichtung
Funktionalität	Die Anwendung erfüllt die definierten Anforderungen und funktioniert zuverlässig	30%

Kriterium	Beschreibung	Gewichtung
Integration	Erfolgreiche Kombination mehrerer Technologien und Module aus dem Kurs	25%
Code-Qualität	Sauberer, lesbarer und gut strukturierter Code mit angemessenen Kommentaren	15%
Innovation	Kreative Lösungsansätze und eigenständige Weiterentwicklung der Konzepte	15%
Dokumentation	Vollständige und verständliche Dokumentation des Projekts	15%

6 | Beispielprojekt: Doku-Assi

Als Orientierung dient hier ein vereinfachtes Beispiel für einen Dokumentenanalyse-Assistenten:

```
# Import der benötigten Bibliotheken
import os
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Chroma
from langchain.chat_models import ChatOpenAI
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain.chains import ConversationalRetrievalChain
import gradio as gr

# API-Keys einrichten
os.environ["OPENAI_API_KEY"] = "Ihr-OpenAI-Key"

# Funktion zum Laden und Verarbeiten von Dokumenten
def load_and_process_document(file_path):
    """
    Lädt ein PDF-Dokument und bereitet es für die Verarbeitung vor

    Args:
        file_path: Pfad zur PDF-Datei

    Returns:
        Chroma-Vektordatenbank mit den Dokumentenchunks
    """
    # PDF laden
    loader = PyPDFLoader(file_path)
    pages = loader.load()
```

```
# Text in Chunks aufteilen
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=100
)
chunks = text_splitter.split_documents(pages)

# Embeddings erstellen und Vektorstore initialisieren
embeddings = OpenAIEMBEDDINGS()
vectorstore = Chroma.from_documents(
    documents=chunks,
    embedding=embeddings
)

return vectorstore

# Chat-Modell und Retrieval-Kette initialisieren
def setup_qa_chain(vectorstore):
    """
    Erstellt eine Konversations-Retrieval-Kette für Frage-Antwort-
    Interaktionen

    Args:
        vectorstore: Chroma-Vektordatenbank

    Returns:
        ConversationalRetrievalChain für QA
    """
    # LLM initialisieren
    llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo")

    # Retriever mit Multi-Query-Strategie
    retriever = MultiQueryRetriever.from_llm(
        vectorstore.as_retriever(search_kwargs={"k": 3}),
        llm
    )

    # QA-Kette erstellen
    qa_chain = ConversationalRetrievalChain.from_llm(
        llm=llm,
        retriever=retriever,
        return_source_documents=True
    )

    return qa_chain

# Gradio-Interface für die Benutzerinteraktion
def create_interface():
    """
```

Erstellt ein Gradio-Interface für die Benutzerinteraktion

Returns:

```

    Gradio-Interface
"""

# Zustandsvariablen
state = {
    "qa_chain": None,
    "chat_history": []
}

# PDF-Upload-Funktion
def upload_pdf(file):
    try:
        vectorstore = load_and_process_document(file.name)
        state["qa_chain"] = setup_qa_chain(vectorstore)
        state["chat_history"] = []
        return "Dokument erfolgreich geladen und verarbeitet!"
    except Exception as e:
        return f"Fehler beim Laden des Dokuments: {str(e)}"

# Frage-Antwort-Funktion
def ask_question(question):
    if state["qa_chain"] is None:
        return "Bitte laden Sie zuerst ein Dokument hoch."

    try:
        result = state["qa_chain"](
            {"question": question, "chat_history":
state["chat_history"]})
    )

        # Chat-Historie aktualisieren
        state["chat_history"].append((question, result["answer"]))

        # Quellen hinzufügen
        sources = set()
        for doc in result["source_documents"]:
            page_content = doc.page_content[:150] + "..." if
len(doc.page_content) > 150 else doc.page_content
            sources.add(f"Quelle (Seite {doc.metadata.get('page',
'N/A')}) : {page_content}")

        sources_text = "\n\n".join(sources)
        full_response = f"{result['answer']}\n\n---\n\nVerwendete
Quellen:\n{sources_text}"

    return full_response
    except Exception as e:
        return f"Fehler bei der Verarbeitung der Frage: {str(e)}"
```

```
# Gradio-Interface erstellen
with gr.Blocks(title="Dokumentenanalyse-Assistent") as interface:
    gr.Markdown("# Dokumentenanalyse-Assistent")
    gr.Markdown("Laden Sie ein PDF-Dokument hoch und stellen Sie Fragen dazu.")
    with gr.Row():
        with gr.Column():
            file_input = gr.File(label="PDF-Dokument hochladen")
            upload_button = gr.Button("Dokument verarbeiten")
            status_text = gr.Textbox(label="Status", interactive=False)

        with gr.Column():
            question_input = gr.Textbox(label="Ihre Frage zum Dokument",
placeholder="Stellen Sie eine Frage zum Inhalt des Dokuments...")
            answer_output = gr.Textbox(label="Antwort",
interactive=False, lines=15)
            ask_button = gr.Button("Frage stellen")

    # Ereignisbehandlung
    upload_button.click(upload_pdf, inputs=[file_input], outputs=[status_text])
    ask_button.click(ask_question, inputs=[question_input], outputs=[answer_output])

return interface

# Hauptfunktion
def main():
    interface = create_interface()
    interface.launch(share=True)

# Ausführung
if __name__ == "__main__":
    main()
```

7 | Ressourcen und Hilfestellung

Folgende Ressourcen können bei der Entwicklung des Abschlussprojekts hilfreich sein:

- **Dokumentation:**
 - [LangChain Dokumentation](#)
 - [OpenAI API Dokumentation](#)
 - [Hugging Face Dokumentation](#)

- [Gradio Dokumentation](#)

- **Beispielprojekte und Tutorials:**

- LangChain Cookbook im GitHub-Repository
- Beispiel-Implementierungen aus den Kursmodulen
- Hugging Face Spaces für Beispielanwendungen

- **Online-Tools:**

- GenAI Tutor
- ChatBots, wie ChatGPT, Gemini, ...
- ...

Bei Fragen oder Problemen während der Projektentwicklung können Sie das Kurs-Forum nutzen.



Viel Erfolg!