
Voice Assistant - Computer Wake-Word Project

Dokument: LLM-Integration Architektur

Datum: 05. Dezember 2025

Seite: {page}

LLM-Integration Architektur

Planung für intelligente Konversationen

Inhaltsverzeichnis

1. [Übersicht](#)
 2. [Warum LLM-Integration?](#)
 3. [Architektur-Design](#)
 4. [LLM-Auswahl](#)
 5. [Command vs. Question Classification](#)
 6. [API-Integration](#)
 7. [Fallback-Strategie](#)
 8. [Code-Struktur](#)
 9. [Implementierungs-Roadmap](#)
-

Übersicht

Die LLM-Integration erweitert den Voice Assistant von einem einfachen Befehlsausführungs-System zu einem intelligenten Konversations-Partner. Statt nur

vordefinierte Befehle auszuführen, kann der Assistant komplexe Fragen beantworten, Recherchen durchführen und kontextbasierte Gespräche führen.

Aktueller Zustand (v3.0)

Der Voice Assistant funktioniert derzeit nach einem einfachen Pattern-Matching-Prinzip:

```
User: "Computer, öffne YouTube"
→ Pattern erkannt: "youtube" in command_lower
→ Aktion: webbrowser.open("https://www.youtube.com")
→ Response: "Öffne YouTube"
```

Limitierungen:

- Nur vordefinierte Befehle funktionieren
- Keine Fragen beantwortbar ("Wie wird das Wetter?")
- Keine Konversation möglich
- Keine Kontextspeicherung

Ziel-Zustand (v4.0 mit LLM)

Mit LLM-Integration wird der Assistant intelligent:

```
User: "Computer, wie wird das Wetter morgen in Berlin?"
→ Klassifizierung: FRAGE (kein Befehl)
→ LLM-Aufruf: Perplexity API
→ Response: "Morgen wird es in Berlin sonnig mit 18 Grad..."

User: "Computer, schreibe eine E-Mail an Max über das Meeting"
→ Klassifizierung: KOMPLEXE AUFGABE
→ LLM-Aufruf: ChatGPT API
→ Response: "Ich habe einen E-Mail-Entwurf erstellt: ..."
```

Warum LLM-Integration?

Vorteile

Natürliche Konversation: Nutzer können in natürlicher Sprache sprechen, ohne sich an spezifische Befehlsformate erinnern zu müssen. Statt “öffne YouTube” kann man fragen “Kannst du mir YouTube öffnen?” oder “Ich möchte Videos schauen” .

Intelligente Antworten: Der Assistant kann komplexe Fragen beantworten, die nicht vorprogrammiert sind. Wetter-Abfragen, Fakten-Recherche, Berechnungen, Übersetzungen und vieles mehr werden möglich.

Kontext-Verständnis: LLMs können den Kontext eines Gesprächs verstehen und darauf aufbauen. Folgefragen wie “Und übermorgen?” werden korrekt interpretiert.

Erweiterbarkeit: Neue Funktionen können durch Prompt-Engineering hinzugefügt werden, ohne Code zu ändern. Der Assistant lernt durch bessere Prompts.

Personalisierung: LLMs können sich an Nutzerpräferenzen anpassen und personalisierte Antworten geben.

Herausforderungen

Latenz: API-Aufrufe zu externen LLMs dauern 1-5 Sekunden, was die Reaktionszeit erhöht.

Kosten: ChatGPT und andere APIs sind kostenpflichtig (ca. \$0.001-0.01 pro Anfrage).

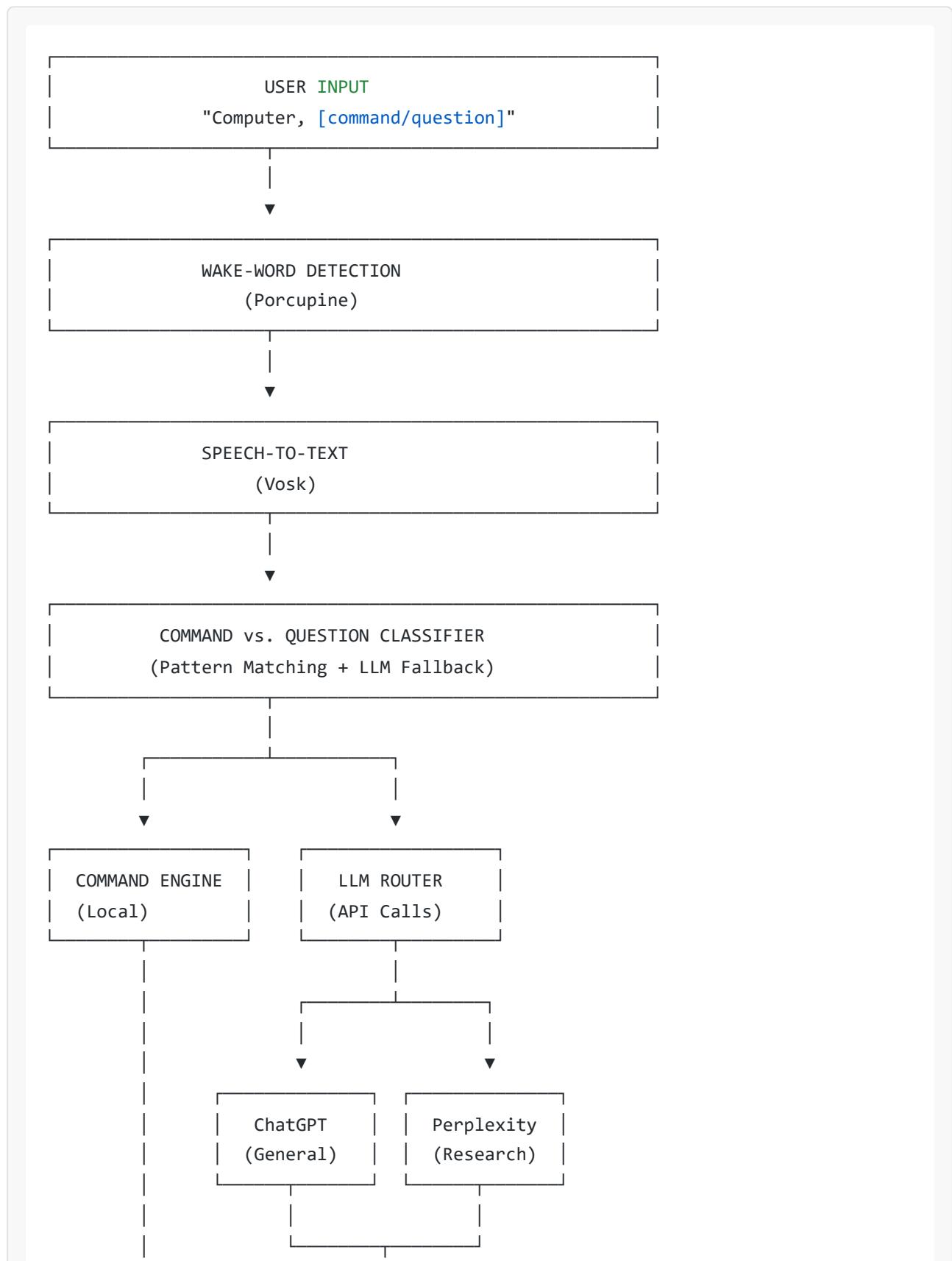
Internetabhängigkeit: Ohne Internet funktionieren Cloud-LLMs nicht.

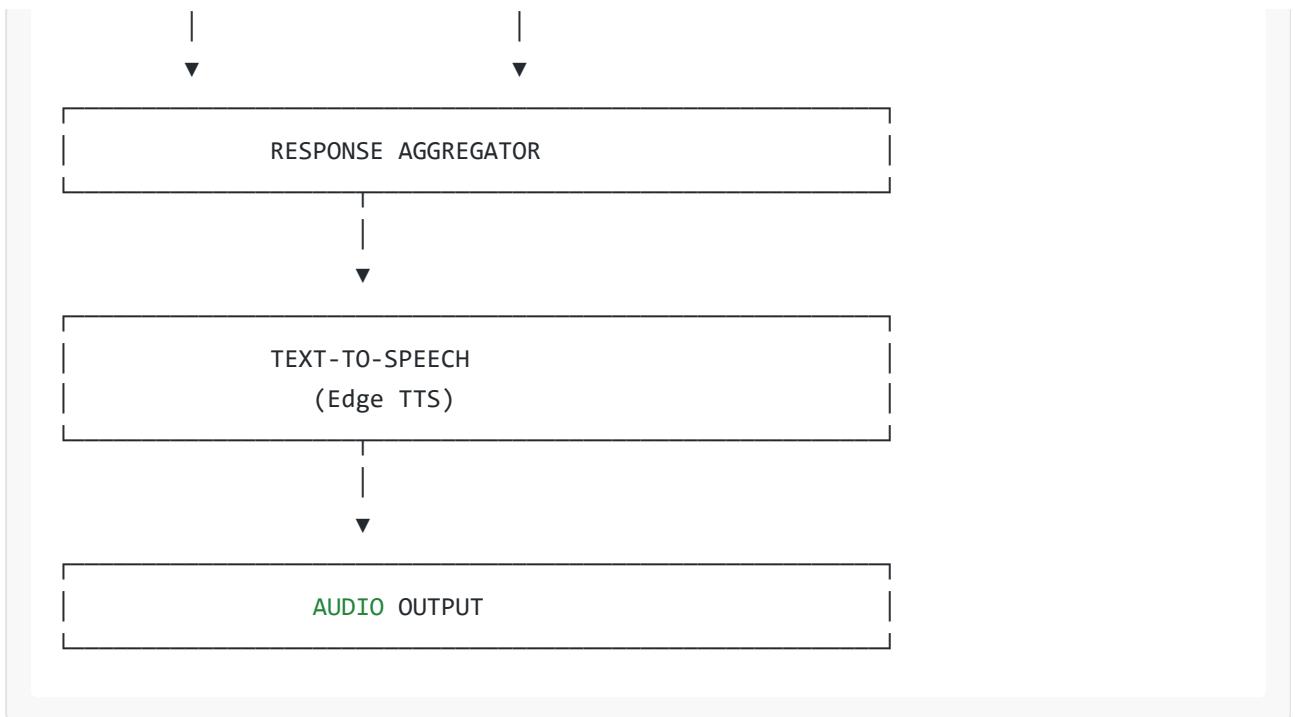
Datenschutz: Audio/Text wird an externe Server gesendet (OpenAI, Perplexity, etc.).

Zuverlässigkeit: API-Ausfälle oder Rate-Limits können die Funktionalität beeinträchtigen.

Architektur-Design

High-Level Architektur





Komponenten-Beschreibung

Wake-Word Detection: Aktiviert den Assistant (bereits implementiert mit Porcupine).

Speech-to-Text: Konvertiert Sprache zu Text (bereits implementiert mit Vosk).

Classifier: Entscheidet, ob Input ein Befehl oder eine Frage ist. Dies ist die zentrale neue Komponente.

Command Engine: Führt lokale Befehle aus (bereits implementiert).

LLM Router: Wählt das passende LLM basierend auf Anfrage-Typ.

LLM APIs: ChatGPT für allgemeine Fragen, Perplexity für Recherche, etc.

Response Aggregator: Kombiniert Antworten und formatiert sie für TTS.

Text-to-Speech: Spricht die Antwort (bereits implementiert mit Edge TTS).

LLM-Auswahl

ChatGPT (OpenAI)

Anwendungsfall: Allgemeine Konversation, Kreativität, Textgenerierung

Stärken:

- Sehr gute natürliche Sprache
- Kontextverständnis
- Kreative Antworten
- Multimodal (Text, Code, Reasoning)

Schwächen:

- Wissen nur bis Trainings-Cutoff (2023)
- Keine Echtzeit-Informationen
- Kann “halluzinieren” (falsche Fakten)

Kosten:

- GPT-4o: 0.005/1K input tokens, 0.015/1K output tokens
- GPT-4o-mini: 0.00015/1K input tokens, 0.0006/1K output tokens

API-Beispiel:

```
from openai import OpenAI
client = OpenAI(api_key="sk-...")

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": "Du bist ein hilfreicher Voice Assistant."},
        {"role": "user", "content": "Wie wird das Wetter morgen?"}
    ]
)
answer = response.choices[0].message.content
```

Empfehlung: Nutze für allgemeine Fragen, Kreativität, E-Mail-Entwürfe, Zusammenfassungen.

Perplexity

Anwendungsfall: Recherche, aktuelle Informationen, Fakten-Check

Stärken:

- Echtzeit-Websuche
- Zitiert Quellen
- Aktuelle Informationen (Wetter, News, etc.)
- Fakten-basiert (weniger Halluzinationen)

Schwächen:

- Weniger kreativ als ChatGPT
- Fokus auf Fakten, nicht Konversation
- Teurer als ChatGPT

Kosten:

- Sonar (Standard): \$0.005/1K tokens
- Sonar Pro: \$0.01/1K tokens

API-Beispiel:

```
import requests

url = "https://api.perplexity.ai/chat/completions"
headers = {
    "Authorization": "Bearer pplx-...",
    "Content-Type": "application/json"
}
data = {
    "model": "sonar",
    "messages": [
        {"role": "user", "content": "Wie wird das Wetter morgen in Berlin?"}
    ]
}

response = requests.post(url, headers=headers, json=data)
answer = response.json()["choices"][0]["message"]["content"]
```

Empfehlung: Nutze für Wetter, News, Fakten-Recherche, aktuelle Events.

Manus

Anwendungsfall: Komplexe Aufgaben, Multi-Step-Reasoning, Tool-Use

Stärken:

- Kann Tools nutzen (Browser, Code, etc.)
- Multi-Step-Reasoning
- Komplexe Aufgaben automatisieren
- Sehr gutes Verständnis

Schwächen:

- Langsamer (mehrere Sekunden)
- Komplexere Integration
- Nicht für einfache Fragen geeignet

Kosten: Abhängig von Manus-Subscription

Empfehlung: Nutze für komplexe Aufgaben wie “Erstelle eine Präsentation über KI” oder “Recherchiere und fasse zusammen” .

Lokale LLMs (Optional)

Anwendungsfall: Offline-Nutzung, Datenschutz

Optionen:

- Ollama (Llama 3, Mistral, etc.)
- LM Studio
- GPT4All

Stärken:

- Vollständig offline
- Keine API-Kosten
- Datenschutz (nichts verlässt PC)

Schwächen:

- Benötigt leistungsstarke GPU (8GB+ VRAM)
- Langsamer als Cloud-LLMs
- Weniger intelligent als GPT-4

Empfehlung: Für zukünftige Version als Fallback implementieren.

Command vs. Question Classification

Klassifizierungs-Strategie

Die Klassifizierung entscheidet, ob ein Input ein **Befehl** (lokal ausführbar) oder eine **Frage** (LLM-Anfrage) ist.

Methode 1: Pattern-Based (Schnell, Offline)

Befehl-Patterns:

```
COMMAND_KEYWORDS = [
    "öffne", "starte", "schließe", "zeige", "spiele",
    "öffnen", "starten", "schließen", "zeigen", "spielen",
    "mach", "mache", "erstelle", "lösche"
]

def is_command(text):
    text_lower = text.lower()
    for keyword in COMMAND_KEYWORDS:
        if keyword in text_lower:
            return True
    return False
```

Frage-Patterns:

```
QUESTION_KEYWORDS = [
    "wie", "was", "wer", "wann", "wo", "warum", "welche",
    "erkläre", "sage mir", "erzähle", "beschreibe"
]

def is_question(text):
    text_lower = text.lower()
    for keyword in QUESTION_KEYWORDS:
        if text_lower.startswith(keyword):
            return True
    if "?" in text:
        return True
    return False
```

Vorteile: Schnell, offline, keine API-Kosten **Nachteile:** Ungenau, viele Edge-Cases

Methode 2: LLM-Based Classification (Genau, Online)

Nutze ein kleines, schnelles LLM (GPT-4o-mini) zur Klassifizierung:

```

def classify_intent(text):
    """Klassifiziert Input als COMMAND, QUESTION oder CONVERSATION."""

    prompt = f"""Klassifizierte folgende Nutzer-Eingabe:

Input: "{text}"

Kategorien:
- COMMAND: Befehl zum Ausführen einer Aktion (öffne, starte, etc.)
- QUESTION: Frage, die eine Antwort benötigt (wie, was, wann, etc.)
- CONVERSATION: Smalltalk oder Höflichkeit (hallo, danke, etc.)

Antworte nur mit: COMMAND, QUESTION oder CONVERSATION"""

    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": prompt},
        max_tokens=10,
        temperature=0
    )

    classification = response.choices[0].message.content.strip()
    return classification

```

Vorteile: Sehr genau, versteht Kontext **Nachteile:** Latenz (+500ms), API-Kosten

Methode 3: Hybrid (Empfohlen)

Kombiniere beide Methoden:

```
def classify_input(text):
    """Hybrid-Klassifizierung: Pattern-based first, LLM fallback."""

    # 1. Schnelle Pattern-Checks
    if is_command(text):
        return "COMMAND"

    if is_question(text):
        return "QUESTION"

    # 2. Spezielle Fälle
    if any(word in text.lower() for word in ["danke", "hallo", "tschüss"]):
        return "CONVERSATION"

    # 3. LLM-Fallback für unklare Fälle
    return classify_intent(text) # LLM-Aufruf
```

Vorteile: Schnell für klare Fälle, genau für unklare Fälle **Nachteile:** Etwas komplexer

API-Integration

ChatGPT API Integration

```
from openai import OpenAI
import os
from dotenv import load_dotenv

load_dotenv()

class ChatGPTAssistant:
    def __init__(self):
        self.client = OpenAI(api_key=os.getenv('OPENAI_API_KEY'))
        self.conversation_history = []

    def query(self, user_input):
        """Sendet Anfrage an ChatGPT und gibt Antwort zurück."""

        # Füge Nutzer-Input zu History hinzu
        self.conversation_history.append({
            "role": "user",
            "content": user_input
        })

        # System-Prompt
        messages = [
            {
                "role": "system",
                "content": "Du bist ein hilfreicher deutscher Voice Assistant
namens Computer."
                    "Antworte kurz und prägnant (max 2-3 Sätze). "
                    "Nutze natürliche, gesprochene Sprache."
            }
        ] + self.conversation_history

        # API-Aufruf
        try:
            response = self.client.chat.completions.create(
                model="gpt-4o-mini",
                messages=messages,
                max_tokens=150, # Kurze Antworten für Voice
                temperature=0.7
            )
        
```

```
answer = response.choices[0].message.content

# Füge Antwort zu History hinzu
self.conversation_history.append({
    "role": "assistant",
    "content": answer
})

# Limitiere History-Länge (letzte 10 Nachrichten)
if len(self.conversation_history) > 10:
    self.conversation_history = self.conversation_history[-10:]

return answer

except Exception as e:
    print(f"ChatGPT API Error: {e}")
    return "Entschuldigung, ich konnte keine Verbindung herstellen."
```

Perplexity API Integration

```
import requests
import os

class PerplexityAssistant:
    def __init__(self):
        self.api_key = os.getenv('PERPLEXITY_API_KEY')
        self.url = "https://api.perplexity.ai/chat/completions"

    def query(self, user_input):
        """Sendet Recherche-Anfrage an Perplexity."""

        headers = {
            "Authorization": f"Bearer {self.api_key}",
            "Content-Type": "application/json"
        }

        data = {
            "model": "sonar",
            "messages": [
                {
                    "role": "system",
                    "content": "Du bist ein Recherche-Assistent. "
                               "Gib kurze, faktische Antworten mit Quellen."
                },
                {
                    "role": "user",
                    "content": user_input
                }
            ],
            "max_tokens": 200
        }

        try:
            response = requests.post(self.url, headers=headers, json=data)
            response.raise_for_status()

            answer = response.json()["choices"][0]["message"]["content"]
            return answer

        except Exception as e:
            print(f"Perplexity API Error: {e}")
            return "Entschuldigung, ich konnte keine Informationen finden."
```


Fallback-Strategie

Multi-Tier Fallback

```
class LLMRouter:
    def __init__(self):
        self.chatgpt = ChatGPTAssistant()
        self.perplexity = PerplexityAssistant()

    def route_query(self, text, intent):
        """Routet Anfrage zum passenden LLM mit Fallback."""

        # 1. Bestimme primäres LLM
        if intent == "QUESTION":
            # Recherche-Fragen → Perplexity
            if any(word in text.lower() for word in ["wetter", "news", "aktuell"]):
                primary = self.perplexity
                fallback = self.chatgpt
            else:
                # Allgemeine Fragen → ChatGPT
                primary = self.chatgpt
                fallback = self.perplexity
        else:
            # Konversation → ChatGPT
            primary = self.chatgpt
            fallback = None

        # 2. Versuche primäres LLM
        try:
            answer = primary.query(text)
            if answer and "Entschuldigung" not in answer:
                return answer
        except Exception as e:
            print(f"Primary LLM failed: {e}")

        # 3. Fallback zu sekundärem LLM
        if fallback:
            try:
                answer = fallback.query(text)
                return answer
            except Exception as e:
                print(f"Fallback LLM failed: {e}")

        # 4. Offline-Fallback
```

```
        return "Entschuldigung, ich kann momentan keine Verbindung herstellen.  
Versuche es später erneut."
```

Code-Struktur

Integration in Voice Assistant

```
# In voice_assistant_computer.py  
  
from llm_integration import LLMRouter, classify_input  
  
# Initialisierung  
llm_router = LLMRouter()  
  
# In main loop, nach STT:  
command = result.get("text", "")  
  
if command:  
    print(f"📝 [STT] Erkannt: \'{command}\'")  
  
    # Klassifiziere Input  
    intent = classify_input(command)  
    print(f"🔍 [CLASSIFIER] Intent: {intent}")  
  
    if intent == "COMMAND":  
        # Lokale Befehlsausführung  
        execute_command(command)  
  
    elif intent in ["QUESTION", "CONVERSATION"]:  
        # LLM-Anfrage  
        print("🤖 [LLM] Sende Anfrage...")  
        answer = llm_router.route_query(command, intent)  
        print(f"💬 [LLM] Antwort: {answer}")  
        speak(answer)  
  
    else:  
        speak("Befehl nicht verstanden")
```

Implementierungs-Roadmap

Phase 1: Basis-Integration (1-2 Tage)

- [] OpenAI API-Key besorgen
- [] ChatGPT Integration implementieren
- [] Einfache Klassifizierung (Pattern-based)
- [] Testing mit 10 Beispiel-Fragen
- [] Dokumentation

Phase 2: Erweiterte Features (3-5 Tage)

- [] Perplexity API-Key besorgen
- [] Perplexity Integration implementieren
- [] LLM-Router mit Fallback
- [] Konversations-History
- [] Hybrid-Klassifizierung

Phase 3: Optimierung (1 Woche)

- [] Latenz-Optimierung (Streaming)
- [] Caching für häufige Fragen
- [] Kosten-Monitoring
- [] Erweiterte Fehlerbehandlung
- [] Performance-Tests

Phase 4: Erweiterte LLMs (Optional)

- [] Manus Integration
- [] Lokales LLM (Ollama) als Offline-Fallback
- [] Multi-Modal (Bilder, Dokumente)

Dokumentende

Seite {page}