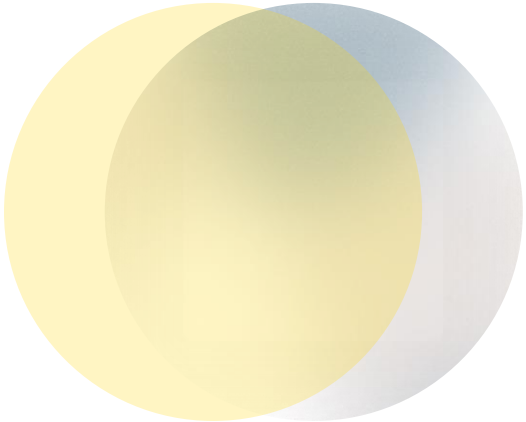


AI Chatbot Development

Katoo Roofthooft , Francesco Chrabieh , Egemen Alkan

Agenda

- Introduction to Chatbot Development
- Utilizing ChromaDB
- Groq API and Huggingface API
- Transition to Qdrant Database
- Replacing Langchain with LlamaIndex
- Content Agents and Upload Materials
- QAAgent
- Crew
- Voice Chat Integration
- Long-Term Memory Capabilities
- Introduction of Study Tools
- React Integration for Quizzes

Overview

Introduction to Chatbot Development

The Personalized Learning Assistant project aimed to create an interactive chatbot to assist learners with tailored support, summaries, and explanations, using the Groq API and Huggingface API.

Utilizing ChromaDB

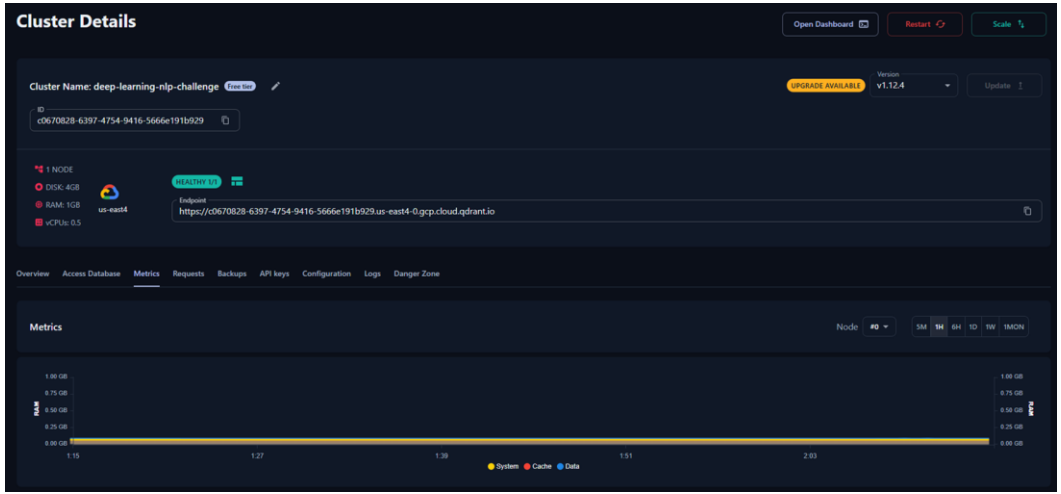
ChromaDB is a local vector database that stores embeddings, enabling efficient retrieval, accurate responses, and context retention by tracking user queries and interactions.

```python
from langchain_chroma import Chroma
from langchain_huggingface import HuggingFaceEmbeddings
import chromadb
from chromadb.config import Settings


class VectorStore:
    def __init__(self, llm_choice="Groq API", groq_api_key=None, hf_key=None):
        self.client = chromadb.Client(Settings(persist_directory="db"))

        if llm_choice == "Groq API":
            self.embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
        elif llm_choice == "HuggingFace API":
            self.embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2")
        else:
            raise ValueError(f"Unsupported LLM choice: {llm_choice}")

        self.vector_store = Chroma(
            client=self.client,
            embedding_function=self.embeddings,
            collection_name="learning_materials"
        )

    def add_documents(self, documents):
        self.vector_store.add_documents(documents)

    def as_retriever(self):
        return self.vector_store.as_retriever(search_kwargs={"k": 5})
```

Transition to Qdrant Database

Benefits of Qdrant

Qdrant's advanced features and optimized indexing improved retrieval speed and performance.



Integration

Replacing Langchain with LlamaIndex

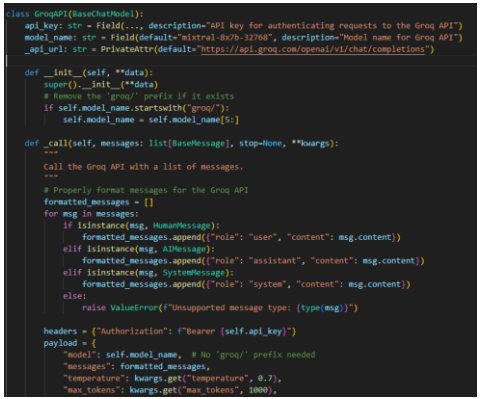
We explored LlamaIndex for better data organization but faced compatibility issues and we reverted to Langchain .

```
from crewai import Agent, Task
from llama_index.core import VectorStoreIndex, Document, Settings
from llama_index.readers.youtube_transcript import YoutubeTranscriptReader
from llama_index.readers.web import SimpleWebPageReader
from llama_index.readers.file import PptxReader
from qdrant_client import QdrantClient
from qdrant_client.http.models import Distance, VectorParams
import os
```

Groq API and Huggingface API

GroqAPI

- Integrates with Groq's language model for chat-based tasks.



```python
class GroqAPI(BaseChatModel):
    api_key: str = Field(..., description="API key for authenticating requests to the Groq API")
    model_name: str = Field(default="mixtral-8x7b-32768", description="Model name for Groq API")
    _api_url: str = PrivateAttr(default="https://api.groq.com/openai/v1/chat/completions")

    def __init__(self, **data):
        super().__init__(**data)
        # Remove the 'groq/' prefix if it exists
        if self.model_name.startswith("groq/"):
            self.model_name = self.model_name[5:]

    def _call(self, messages: list[BaseMessage], stop=None, **kwargs):
        """
        Call the Groq API with a list of messages.
        """
        # Properly format messages for the Groq API
        formatted_messages = []
        for msg in messages:
            if isinstance(msg, HumanMessage):
                formatted_messages.append({"role": "user", "content": msg.content})
            elif isinstance(msg, AIMessage):
                formatted_messages.append({"role": "assistant", "content": msg.content})
            elif isinstance(msg, SystemMessage):
                formatted_messages.append({"role": "system", "content": msg.content})
            else:
                raise ValueError(f"Unsupported message type: {type(msg)}")

        headers = {"Authorization": f"Bearer {self.api_key}"}
        payload = {
            "model": self.model_name,  # No 'groq/' prefix needed
            "messages": formatted_messages,
            "temperature": kwargs.get("temperature", 0.7),
            "max_tokens": kwargs.get("max_tokens", 1000),
```

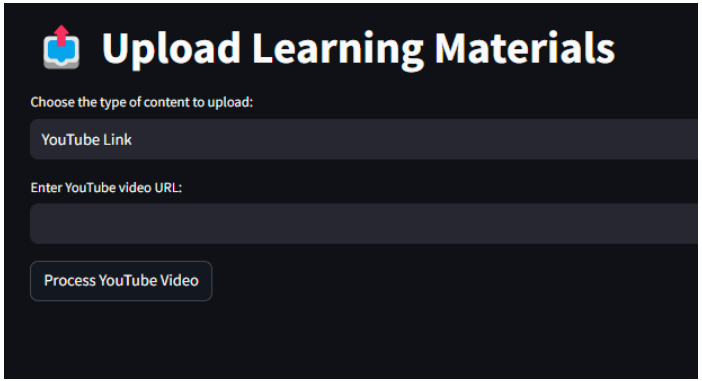
HuggingFaceAPI

- Connects to Hugging Face models for text generation.

```python
class HuggingFaceAPI(BaseChatModel):
    api_key: str = Field(..., description="API key for authenticating requests to the Hugging Face API")
    model_name: str = Field(default="tiiuae/falcon-7b-instruct", description="Model name for Hugging Face API")
    max_tokens: int = Field(default=1000, description="Maximum number of tokens to generate")
    temperature: float = Field(default=0.7, description="Sampling temperature")
    _api_url: str = PrivateAttr(default="https://api-inference.huggingface.co/models/")
    _client: InferenceClient = PrivateAttr()

    def __init__(self, **data):
        super().__init__(**data)
        load_dotenv()
        self.api_key = self.api_key or os.getenv("HF_API_KEY")
        if not self.api_key:
            raise RuntimeError("HuggingFace API key not found.")

        # Initialize the Inference Client
        self._client = InferenceClient(api_key=self.api_key)
        self.model_name = self.model_name

    def _call(self, messages: list[BaseMessage], context=None, stop=None, **kwargs):
        """
        Call the Hugging Face Inference API with optional context.
        """
        formatted_prompt = self._format_prompt(messages, context)
        headers = {"Authorization": f"Bearer {self.api_key}"}
        payload = {
            "inputs": formatted_prompt,
            "parameters": {
                "max_new_tokens": kwargs.get("max_tokens", self.max_tokens),
                "temperature": kwargs.get("temperature", self.temperature),
            },
        }
```

Content Agents and Upload Materials

We introduced content agents to process and index various educational materials, including PDFs, video transcripts, and presentations..



QAAgent

The QAAgent combines document retrieval, web search using GroqAPI or HuggingFaceAPI. It dynamically selects methods, validates context relevance, and generates accurate answers by integrating multiple data sources and tailored prompts for diverse question-answering needs.

```python
class QAAgent(Agent):
    _vector_store: PrivateAttr
    _llm: PrivateAttr
    _use_chain: PrivateAttr
    _retriever: PrivateAttr
    _qa_chain: PrivateAttr

    def __init__(self, vector_store, llm_choice, groq_api_key=None, hf_key=None):
        # Initialize LLM based on user choice
        if llm_choice == "Groq API":
            llm = GroqAPI(api_key=groq_api_key, model_name="mixtral-8x7b-32768")
            use_chain = True
        elif llm_choice == "HuggingFace API":
            llm = HuggingFaceAPI(api_key=hf_key, model_name="tiiuae/falcon-7b-instruct")
            use_chain = False
        else:
            raise ValueError(f"Unsupported LLM choice: {llm_choice}")

        # Initialize the Agent with CrewAI parameters
        super().__init__(
            role="Question Answering Expert",
            goal="Provide accurate and detailed answers based on available context",
            backstory="Expert at analyzing context and providing comprehensive answers",
            allow_delegation=True,
            llm=llm,
            verbose=True
        )
```

Crew

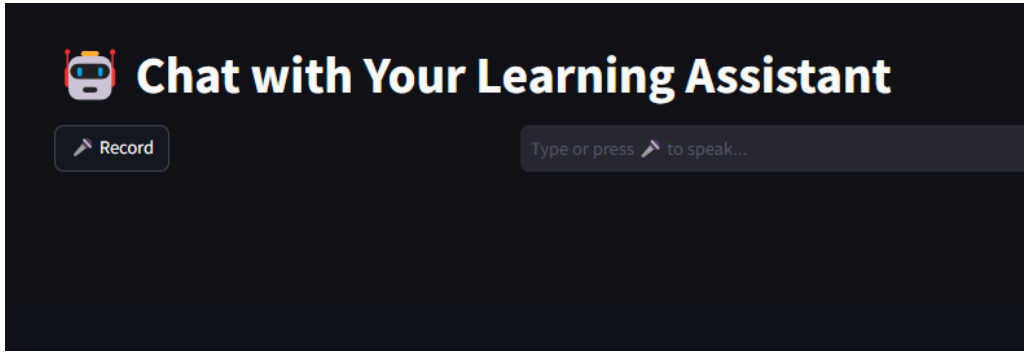
The ResearchCrew coordinates content and QA agents and it dynamically handles research-based and direct question-answering tasks, integrating multiple agents.

```
# Create crew
crew = Crew(
    agents=[self.content_agent, self.qa_agent],
    tasks=[research_task, qa_task],
    process=Process.sequential,
    verbose=True
)
```

Voice Chat

Voice Chat Integration

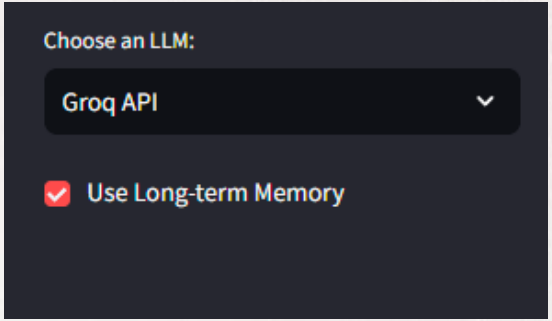
The project integrated OpenAI's Whisper model for accurate speech-to-text transcription and gTTS for text-to-speech.



Implementation

Long-Term Memory Capabilities

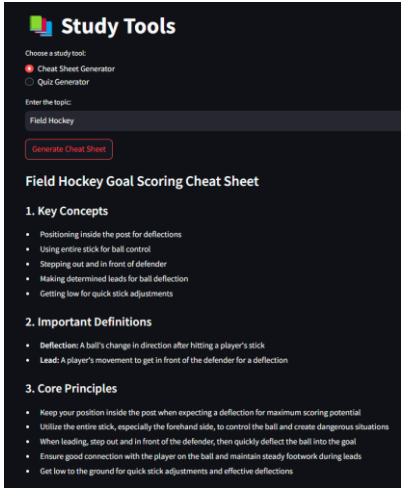
The long-term memory integration enhanced the learning assistant by enabling it to retain and utilize historical context through the MemoryStore class.

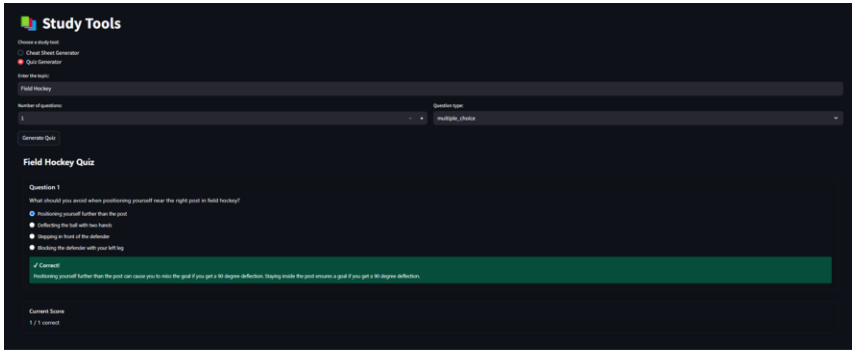


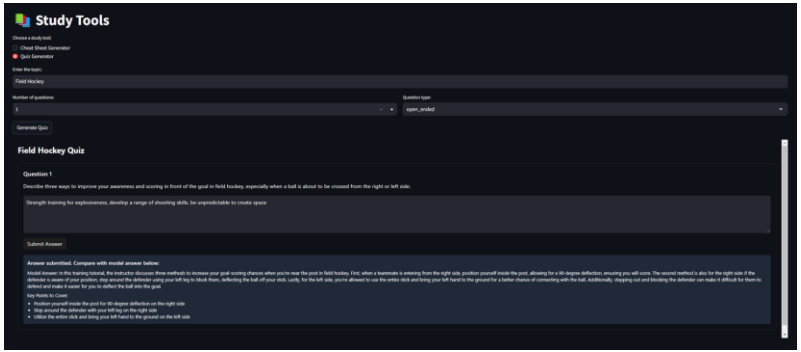
Study Tools

Introduction of Study Tools

The CheatSheetAgent generates concise summaries of key concepts, while the QuizAgent creates tailored quizzes with instant feedback.



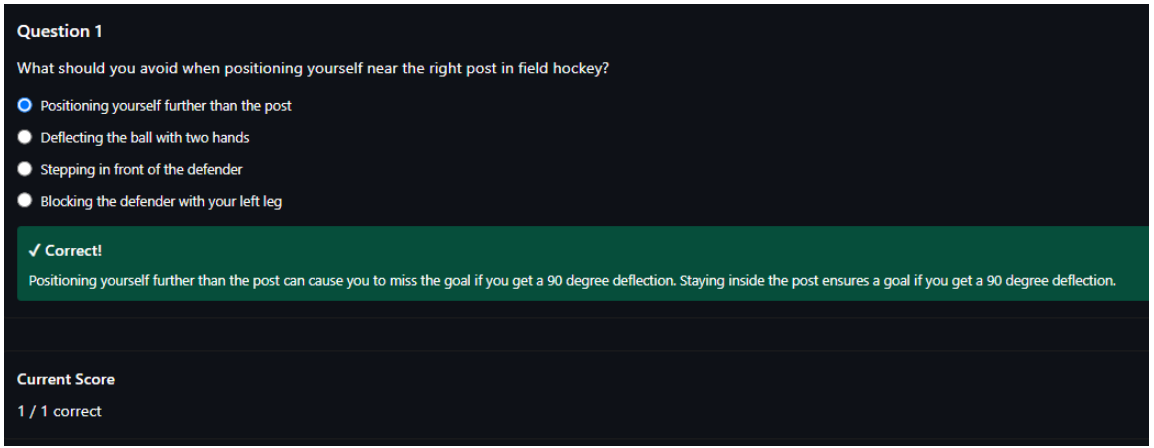




Integration

React Integration for Quizzes

Due to issues with Streamlit where quizzes disappeared upon answering the questions or submission, we used React to implement the quizzes, ensuring proper functionality and displaying results.



Thank you.