GenNarrate

INF-3993 Generative Artificial Intelligence
University of Tromsø

Opdahl, T and Wilhelmsen, M

June, 2025

I Introduction

This project explores the application of generative AI models to create an accessible, voice-driven interface for document comprehension.

This application, named **GenNarrate**, allows a user to learn about and explore a PDF document in an engaging way my being able to speak directly to an assistant that knows all about the document and is able to discuss it with a user.

The system achieves this though its retrieval-augmented generation (RAG) pipeline, which grounds the language model's responses in relevant parts of the original document.

The goal of this project is to demonstrate how recent advancements in generative AI—specifically LLMs, TTS, and ASR—can be composed into a coherent and multimodal user experience.

II Technical Background

Below we have provided a brief overview of the models and processes used in this project.

II.I Large Language Model (LLM)

This project uses the DeepSeek LLM with 7B parameters[3]. Although small and compact in comparison to state-of-the-art LLMs, this model is still capable of advanced reasoning and language understanding. The model follows a decoder-only transformer architecture, trained using causal language modeling to predict the next token based on previous context.

In this project, the model is used to generate responses to user queries, often enhanced with additional context retrieved via a retrieval-augmented generation (RAG) pipeline. This makes it suitable for open-ended dialogue and task-oriented conversation generation, especially when combined with contextual document embeddings.

II.II Retrieval-Augmented Generation (RAG)

A Retrieval-Augmented Generation (RAG) pipeline is implemented into this project to help the LLM to get valuable context to a user prompt. Given a user query, the system first encodes it using a sentence transformer model, specifically `all-MiniLM-L6-v2`[1]. It then performs a nearest-neighbor search over previously embedded documents stored in a ChromaDB vector database[2].

The retrieved document chunks are sent along with the user prompt, allowing it to ground its generation in more accurate and domain-specific information.

II.III Automatic Speech Recognition (ASR)

The model this project uses for speech recognition is Whisper from OpenAI. Whisper is a multilingual encoder-decoder transformer trained on 680,000 hours of audio data, making it robust across different

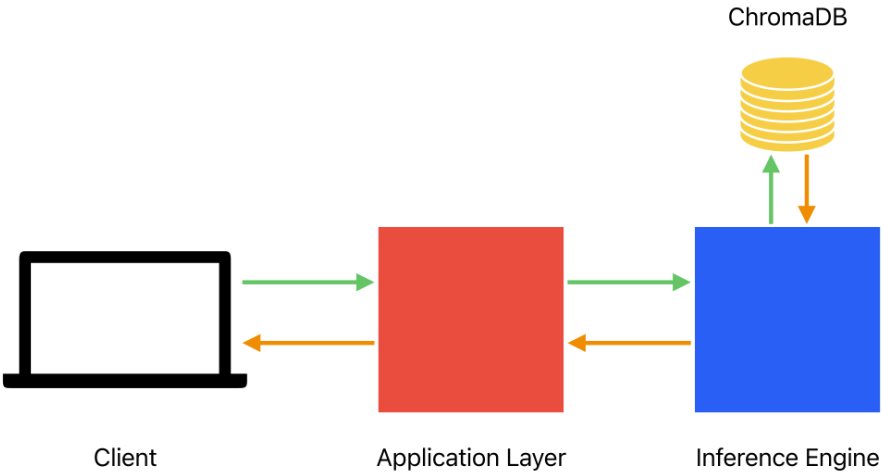

Figure 1: System architecture of GenNarrate, illustrating the interaction between the frontend, backend, and inference engine.

languages, accents, and recording conditions[6]. This project uses the largest variant of Whisper which is the `large-v3` variant.

Whisper processes incoming speech by first resampling it to 16kHz and then encoding the waveform before generating the corresponding text tokens. In our system, this transcribed text is passed to the language model for further processing. The use of Whisper enables hands-free interaction with the DeepSeek LLM, making the system multimodal.

II.IV Text-to-Speech (TTS)

To produce spoken responses, the project uses the Kokoro-82M TTS model[4]. This neural model converts generated text into speech using the `af_heart` voice and returns an audio stream in real time.

III Design

The system follows a three-tiered architecture with clearly separated frontend, backend, and inference layers. As visualized in Figure 1, the frontend is responsible for user interaction, the backend handles data processing and logic, and the inference engine performs the heavy computational tasks.

III.I User Interface

The UI, also known as the frontend, is written in JavaScript with the React.js[5] framework. React.js is a widely adopted JavaScript library for developing user interfaces in a component-based manner. This part of the design provides a clean user-friendly experience, allowing users to interact with the system through a web browser. The UI includes features for text, voice and file input, as well as displaying the generated responses from the models.

III.II Application Layer

The App Layer, also known as the backend, is written in Python using the Flask framework[**?**]. Flask works as an API that handles requests from the user interface and processes data logic before sent elsewhere. It pre-processes user inputs and handles CPU-intensive tasks ranging from document chunking to conversation history management. The application layer acts as a bridge between the user interface and the infer-

ence engine, ensuring no direct communication between the two.

III.III Inference Engine

The inference engine, which is also a backend layer, uses the same framework as the application layer, Flask, though designed to run on separate nodes. It is responsible for computing GPU-intensive tasks such as running the LLM, ASR, TTS, and RAG pipelines. Also hosting the ChromaDB[2] vector database for storing and retrieving document embeddings.

IV Implementation

IV.I User Interface

The user interface consists of JavaScript functions for client-side data processing, and renders objects with HTML and CSS for styling. The JavaScript code cooperates with the HTML by using state variables to handle user actions and dynamically update the UI.

The functions for submitting audio, prompts and files share a similar structure.
When a user submits input to the interface, the functions appends relevant state variables to a form, and then this form is sent to the application layer through `HTTP POST` requests. In the case of submitting a file of the document or audio kind, the application layer will respond with a simple success message, while a text prompt will return a full response from an AI model. These responses are added to a state variable which contains the conversation history, and is cooperating with HTML to render the conversation in a chat-like format.

However, if the user submits a text or audio prompt with the text-to-speech state variable enabled, the function will render the audio stream returned from the application layer immediately to the user interface, allowing the user to listen to the response without waiting for the full response from the AI model.

The JavaScript also enables the user to record audio using a third-party recording library.

IV.II Application Layer

The application layer is an API connecting the user interface to the inference engine, pre-processing data and managing conversation history. See Figure 2 for an overview of the application layer endpoints.

`/upload/speech/` serves as the endpoint for uploading audio files, and demands the audio file, `history` as `JSON` and the text-to-speech variable. Using a custom made function, the `history` is formatted into a suitable string called `conversation` for the AI models on the inference engine. The audio file is then forwarded for transcription using the `/recognizeTextFromSpeech` endpoint of the inference engine. Depending on the text-to-speech variable, the response is either returned as a text string, or relayed as an audio stream to the user interface.

The `/upload/text/` endpoint takes the same parameters as the previous endpoint, but instead of audio, it takes a text prompt. After processing the `history` into `conversation`, it forwards the prompt to the corresponding endpoint from the text-to-speech variable. If the text-to-speech variable is enabled, the request is sent to the `/generateSpeechFromText` endpoint, which returns an audio stream. If the text-to-speech variable is disabled, the request is sent to the `/generateTextFromText` endpoint, which returns a text response.

Out of the three, the `/upload/file` endpoint handles the most intensive logic. After receiving a file, it will read the contents of the file as text, and chunk it into smaller pieces. The chunks are then sent to the `/indexEmbeddings` endpoint of the inference engine which uses a sentence transformer model to return an embedding for each chunk. These embeddings are indexed using FAISS, which is a library for efficient similarity search and clustering of dense vectors. The embeddings are stored in ChromaDB[2] under a common document identifier, allowing them to be retrieved later during inference. This process enables a Retrieval-Augmented Generation (RAG) pipeline that enhances the LLM's inference capability. This endpoint returns a simple success message to the user interface, indicating that the file has been successfully processed and indexed.

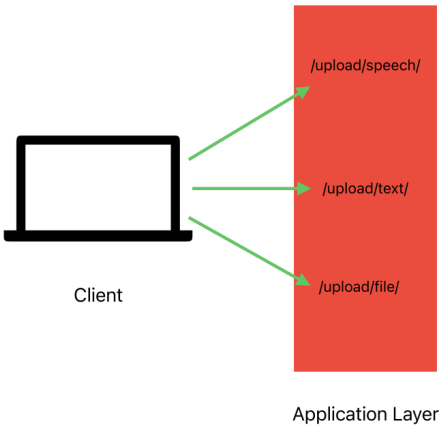

Figure 2: Overview of the application layer endpoints

IV.III Inference Engine

For this section, we will emphasize the models in order to reflect the course content. See Figure 3 for an overview of the inference engine and its connection to the application layer and the database.

IV.III.1 DeepSeek LLM

The inference engine loads the `deepseek-llm-7b-chat`[3] model using the Hugging Face `transformers` library. Its initialized with `float16` precision and moved to GPU memory on server startup for better efficiency. To avoid unessecary gradient computations, the model is paired with a corresponding tokenizer which is set to evaluation mode.
Each incoming request includes the conversation history and current prompt. Before generation, the system retrieves contextual information from ChromaDB. The context is always added to the prompt string if context is available. The fully assembled query is then tokenized and passed to the model, the generatd output is then decoded and returned as text.

IV.III.2 RAG with Chroma DB

The Retrieval-Augmented Generation (RAG) component uses the `all-MiniLM-L6-v2` model from SentenceTransformers to embed both user prompts and document chunks. On startup, the embedding model is loaded onto the GPU.

Document chunks are embedded and stored using ChromaDB[2], a persistent vector database. When a user prompt is received, it is embedded and used to query ChromaDB for the top-k most relevant chunks. These chunks are spliced together into a single context string and added to the context of the LLM input.
There are two endpoints in the inference engine that facilitates for storing files in the Chroma DB.

- **/generateEmbeddings:** endpoint receives raw text chunks and returns their vector embeddings using the MiniLM model.

- **/indexEmbeddings:** endpoint receives both the raw chunks and their precomputed embeddings, and stores them in ChromaDB under a common document identifier. Each chunk is assigned a unique ID to support retrieval later during inference.

IV.IV ASR with Whisper

The Automatic Speech Recognition (ASR) component uses OpenAI's `whisper-large-v3`[6] model to transcribe user audio input into text. The model is loaded onto the GPU at server startup and used through a Hugging Face pipeline for streamlined inference.

Incoming audio is received as a file upload and is then loaded into memory using `torchaudio`[**?**]. Since most microphones use 44.1kHz or 48kHz, all audio is resamples to meet Whisper's requirement of 16kHz before inference. If the audio is stereo, it is downmixed to mono. When the transcription is complete, the resulting text is then appended to the conversation history and passed on for further response generation for the LLM.

This component powers hands-free interaction via the `/recognizeTextFromSpeech` endpoint and enables the system to support voice-based dialogue.

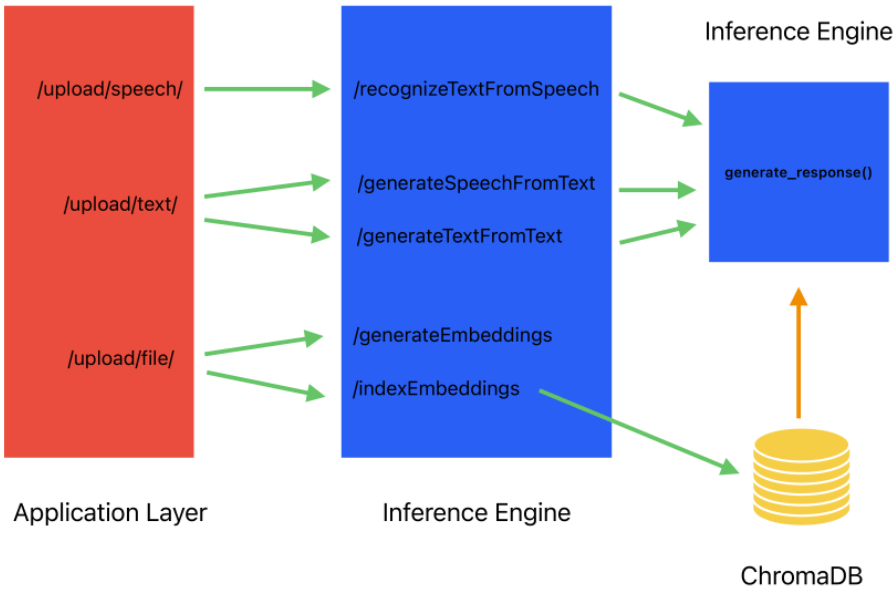

Figure 3: Overview of the inference engine and connection to the application layer

IV.IV.1 TTS with Kokoro

The Text-to-Speech (TTS) component uses the Kokoro-82M model[4] to synthesize audio from text. The model is accessed through the `KPipeline` interface, which produces audio in small chunks suitable for streaming.

Audio chunks are streamed as they are created using Flask's `stream_with_context` and encoded as WAV using the `soundfile`[?] library. Doing this makes it so the system delivered speech responses with minimal delay.

IV.IV.2 Endpoints Overview

- **/recognizeTextFromSpeech:** Generates text audio sent by user to then pass on for further processing by the `generate_response()` function (see Section IV.IV.3).
- **/generateTextFromText:** Endpoint that receives prompt from user, and sends prompt on for further processing with the `generate_response()` function before feeding to the LLM.
- **/generateSpeechFromText:** Receives user prompt, processes full query and receives respons from LLM, then sends audio stream back with function `generate_audio_stream()` that creates audio from the text provided by the LLM.

IV.IV.3 Importiant Functions

- **generate_response(conversation, prompt):** This function handles the end-to-end logic for generating a text response to a user prompt. First, the prompt is embedded and used to query ChromaDB for relevant document chunks, forming a contextual grounding string. The prompt is then classified via a zero-shot classifier to determine intent.

  Based on the classification, the function selects the appropriate model and tokenizer (currently only the DeepSeek LLM is used). The conversa-

tion history and context are formatted into a full prompt and passed to the model for generation. The output is cleaned and returned as a plain string.

- **generate_audio_stream(text):** This function uses the Kokoror TTS pipeline to transform input text into speech. The Kokoro pipeline returns an iterator over audio chunks, which are encoded into WAV format. The function yields these chunks incrementally via a Flask stream, enabling real-time audio playback in the client.

V Experiments and Results

VI Discussion

VI.I Unfinished Features

In the implementation of the backend server, there is a implementation of a feature that is not yet finished as it does not communicate with the frontend. The features goal is to take a PDF file, and send this file to a GPT-4 model using OpenAI's API, so that this model can separate the PDF into narratable chunks that are contextually made, meaning that the document is split up in such a way that there is natural pauses between each chunk, so that each chunk stops at an appropriate point in the text and is not awkward, enhancing user experienceS.

This enables the user to have the application read the document to them, chunk by chunk, so that the user can ask questions about the chunk they just heard.

The reason for choosing to go for a larger LLM in this case is simple, its a different way of interacting with a LLM, and also the model on the inference server cannot handle very large documents, something GPT4 handles exceptionally better.

VII Conclusion

GenNarrate demonstrates how generative AI models can be combined to form an end-to-end multimodal system for document comprehension. By integrating a large language model, a TTS engine, and optional ASR functionality, the system allows users to consume and interact with written documents through speech.

The architecture was designed with clarity and modularity in mind, separating the inference engine from the application logic and user interface.

Although this project is built for a single-user research setting, it showcases the potential of deploying conversational AI beyond traditional chat interfaces.

VII..1

References

[1] all-minilm-l6-v2. `https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2`, 2021. Accessed: 2025-06-01.

[2] Chroma: the ai-native open-source embedding database. `https://www.trychroma.com/`, 2023. Accessed: 2025-06-01.

[3] DeepSeek. deepseek-ai/deepseek-llm-7b-chat. `https://huggingface.co/deepseek-ai/deepseek-llm-7b-chat`, 2024. Accessed: 2025-06-01.

[4] Hexgrad. Kokoro-82m. `https://huggingface.co/hexgrad/Kokoro-82M`, 2024. Accessed: 2025-06-01.

[5] Meta Platforms, Inc. React – a javascript library for building user interfaces. `https://react.dev/reference/react`, 2024. Accessed: 2025-06-01.

[6] Alec Radford, Jong Wook Gao, Greg Brockman, et al. Whisper: Robust speech recognition via large-scale weak supervision. `https://openai.com/index/whisper/`, 2022. Accessed: 2025-06-01.