**Interactive Text Generation with Large Language Models (LLMs)**

This project demonstrates the development of an interactive web application for text generation using a powerful Large Language Model (LLM) called LlamaCpp. The application leverages two technologies: Streamlit for a user-friendly interface and Flask for running the LLM model.

**System Architecture**

The system comprises two main components:

1. **Flask Backend Server:**
   * This server utilizes Flask, a Python web framework, to run the LlamaCpp LLM model.
   * The code (explained previously) initializes the model, defines a route (/generate\_text) for handling text generation requests, and performs inference with the LLM based on received prompts.
2. **Streamlit Frontend Application:**
   * This application is built using Streamlit, a Python library for creating web UIs.
   * The code (explained previously) creates a user interface with a text area for users to enter prompts and a button to trigger text generation.
   * Upon clicking the button, Streamlit sends a POST request to the Flask server's /generate\_text route, including the user's prompt as JSON data.
   * The application receives the generated text from the Flask server's response and displays it to the user.

**Technical Details**

* **Frontend Framework:** Streamlit
* **Backend Framework:** Flask
* **Large Language Model:** LlamaCpp
* **LLM Interface (on Backend):** Langchain libraries
* **Programming Language:** Python

**Functionalities**

* Users can enter prompts through a user-friendly text area.
* The application interacts with the LLM model to generate creative text formats based on the provided prompts.
* The generated text is displayed within the application.

**Future Enhancements**

* Implement functionalities beyond text generation (e.g., question answering, summarization) within the Streamlit app.
* Explore advanced UI elements in Streamlit for richer user interaction (e.g., progress bars, dropdowns).
* Integrate user authentication and authorization for secure access on the Flask server.

**Conclusion**

This project showcases the potential of combining Streamlit and Flask to create interactive web applications that leverage powerful LLMs. It provides users with a convenient way to explore text generation capabilities and experiment with creative prompts. The proposed enhancements can further extend the application's functionalities and user experience.

The libraries used in this combined project are:

**Frontend (Streamlit App):**

* streamlit as st: This library provides tools for building user interfaces (UIs) in Python using Streamlit.
* requests: This library enables making HTTP requests to external services (in this case, the Flask server).

**Backend (Flask Server):**

* Flask: This is a popular Python framework for building web applications.
* langchain.llms.LlamaCpp: This library provides an interface to the LlamaCpp LLM model.
* langchain\_core.callbacks: This library provides tools for managing model callbacks in Langchain (a framework for LLMs).
* langchain\_core.prompts: This library helps with defining prompts for the LLM model.

Show drafts

Draft 1

These two code snippets, the Flask application and the Streamlit application, work together to achieve the functionality of user-driven text generation using a large language model (LLM). Here's how t...

Draft 1

These two code snippets, the Flask application and the Streamlit application, work together to achieve the functionality of user-driven text generation using a large language model (LLM). Here's how they connect:

**1. Streamlit Frontend (User Interaction):**

* The Streamlit code creates a user interface with a text area where users can enter their prompts.
* When the user clicks the "Generate Text" button, the Streamlit application triggers a function that sends a POST request to the Flask server.

**2. Flask Backend (Model Interaction):**

* The Flask application defines a route (/generate\_text) that handles incoming POST requests.
* When a request arrives at this route, the Flask server extracts the user prompt data sent from the Streamlit application.
* The code utilizes the LlamaCpp LLM model (initialized earlier) to perform text generation based on the received prompt.
* The generated text is packaged as a JSON response with a key named "output".
* The Flask server sends this JSON response back to the Streamlit application.

**3. Streamlit Receives Response and Displays Output:**

* The Streamlit application receives the response from the Flask server.
* It checks the response status code. If successful (code 200), it extracts the generated text from the JSON response under the "output" key.
* Finally, the Streamlit application displays the generated text to the user within the application interface.

**In essence:**

* The Streamlit application acts as a user interface, collecting user input and sending it to the Flask server for processing.
* The Flask application functions as the backend, interacting with the LLM model and generating text based on the received prompts.
* The communication between these two applications happens through HTTP requests and responses, allowing users to seamlessly interact with the LLM model from the Streamlit interface.

Draft 2

These codes are connected to form a complete web application for generating text with a large language model (LLM). Here's how they work together:

1. \*\*Flask Backend Server (separate Python script):\*...

Draft 2

These codes are connected to form a complete web application for generating text with a large language model (LLM). Here's how they work together:

1. **Flask Backend Server (separate Python script):**
   * This script runs the LLM model (LlamaCpp) and provides an API endpoint for text generation requests.
   * It uses libraries like Flask, langchain.llms, and langchain\_core to handle model initialization, prompt processing, and inference.
   * It defines a route (/generate\_text) that listens for incoming requests.
2. **Streamlit Frontend Application (separate Python script):**
   * This script builds the user interface (UI) for interacting with the LLM model.
   * It uses Streamlit to create a user-friendly interface with a text area for prompts and a button to trigger text generation.
   * When the user clicks the button, Streamlit sends a POST request to the Flask server's /generate\_text route.
   * The request includes the user's prompt data in JSON format.
3. **Connection:**
   * The Streamlit app acts as the client, interacting with the Flask server as the backend API.
   * When the Streamlit app sends a request, the Flask server processes it using the LLM model.
   * The Flask server generates text based on the prompt and sends the generated text back to the Streamlit app in a response.
   * Finally, the Streamlit app receives the response and displays the generated text to the user.

**In essence:**

* The Streamlit app provides a user interface for entering prompts.
* The Flask server acts as the middle layer, handling communication with the LLM model.
* The LLM model (LlamaCpp) resides within the Flask server and performs the actual text generation based on the prompts.
* The generated text is then sent back to the Streamlit app for display.

This combined approach allows for a clear separation of concerns:

* Streamlit focuses on building the user interface.
* Flask handles the backend logic and interaction with the LLM model.

This modular design makes the application easier to maintain and understand.

Draft 3

These codes are connected to work together and achieve text generation using a large language model (LLM). Here's how they interact:

**1. Flask Backend Server (separate Python script):**

* This serv...

Draft 3

These codes are connected to work together and achieve text generation using a large language model (LLM). Here's how they interact:

**1. Flask Backend Server (separate Python script):**

* This server runs the LLM model (LlamaCpp) and handles text generation requests.
* It utilizes libraries like Flask, langchain (for LLM access), and others.
* It defines a route (/generate\_text) that listens for incoming requests.

**2. Streamlit Frontend Application (separate Python script):**

* This application provides the user interface for interacting with the LLM.
* It utilizes Streamlit to create a user-friendly interface with a text area and a button.
* When the user enters a prompt and clicks the button:
  + Streamlit sends a POST request to the Flask server's /generate\_text route.
  + This request includes the user's prompt data in JSON format.

**Connection:**

* The Streamlit application acts as the client, initiating communication with the Flask server.
* The Flask server acts as the backend, processing the user's prompt and generating text using the LLM.
* The connection between them happens through HTTP requests and responses.
  + Streamlit sends a POST request with the prompt data.
  + The Flask server processes the request, performs LLM inference, and sends a response containing the generated text.
* Streamlit receives the response from the Flask server and displays the generated text to the user within the application.

**Essentially:**

* The user interacts with the Streamlit application.
* Streamlit communicates with the Flask server behind the scenes.
* The Flask server interacts with the LLM model to generate text.
* The generated text is sent back to Streamlit and displayed to the user.

This combined approach separates the user interface (Streamlit) from the LLM model processing (Flask), making the application more modular and maintainable.

refresh Regenerate drafts

volume\_up

These two code snippets, the Flask application and the Streamlit application, work together to achieve the functionality of user-driven text generation using a large language model (LLM). Here's how they connect:

**1. Streamlit Frontend (User Interaction):**

* The Streamlit code creates a user interface with a text area where users can enter their prompts.
* When the user clicks the "Generate Text" button, the Streamlit application triggers a function that sends a POST request to the Flask server.

**2. Flask Backend (Model Interaction):**

* The Flask application defines a route (/generate\_text) that handles incoming POST requests.
* When a request arrives at this route, the Flask server extracts the user prompt data sent from the Streamlit application.
* The code utilizes the LlamaCpp LLM model (initialized earlier) to perform text generation based on the received prompt.
* The generated text is packaged as a JSON response with a key named "output".
* The Flask server sends this JSON response back to the Streamlit application.

**3. Streamlit Receives Response and Displays Output:**

* The Streamlit application receives the response from the Flask server.
* It checks the response status code. If successful (code 200), it extracts the generated text from the JSON response under the "output" key.
* Finally, the Streamlit application displays the generated text to the user within the application interface.

**In essence:**

* The Streamlit application acts as a user interface, collecting user input and sending it to the Flask server for processing.
* The Flask application functions as the backend, interacting with the LLM model and generating text based on the received prompts.
* The communication between these two applications happens through HTTP requests and responses, allowing users to seamlessly interact with the LLM model from the Streamlit interface.

Imagine a two-part system:

1. **Website (Streamlit):** This is where users see things. They type their prompt in a text box and click a button. Behind the scenes, the website sends this prompt to a different program.
2. **Text Generator (Flask):** This program lives on a server and runs the powerful language model (LlamaCpp). It receives the prompt from the website, uses the model to create text, and sends that text back.

The website then displays the generated text to the user. So, the website lets users interact, while the hidden program does the heavy lifting with the model. They work together to create the text generation experience.

While an ER diagram isn't typically used for this kind of application structure, here's a simplified explanation using an Entity-Relationship approach:

**Entities:**

1. **User:** Represents the individual interacting with the application. (Has No Attributes in this case)
2. **Prompt:** Represents the text input provided by the user for text generation. (Has Attribute: prompt\_text - text)
3. **Generated Text:** Represents the creative text produced by the LLM model based on the prompt. (Has Attribute: generated\_text - text)

**Relationships:**

1. **Provides (User, Prompt):** A user provides a prompt for text generation. (One user can provide many prompts, one prompt can be provided by one user)
2. **Generates (Prompt, Generated Text):** A prompt is used to generate text by the LLM model. (One prompt can generate one generated text, one generated text is generated from one prompt)

**Explanation:**

* A user interacts with the application and provides a prompt through the Streamlit interface.
* This prompt is sent to the Flask server, which represents the "Generates" relationship.
* The Flask server utilizes the LLM model to generate creative text based on the received prompt.
* The generated text is sent back to the Streamlit application.
* The Streamlit application displays the generated text to the user, representing the "Provides" relationship back to the user.

**Note:**

This simplified explanation focuses on the core functionality of prompt submission and text generation. The actual implementation might involve additional entities and relationships depending on specific functionalities (e.g., storing user history, managing different LLM models).

The concept of this project is to create a user-friendly application that leverages a powerful large language model (LLM) for text generation. Here's a breakdown of the key aspects:

**1. User-driven Text Generation:**

* Users interact with the application through a Streamlit interface, providing prompts for the LLM.
* These prompts can be anything from simple phrases to complex instructions, allowing users to explore the creative capabilities of the model.

**2. Large Language Model (LlamaCpp):**

* The project utilizes the LlamaCpp LLM, a powerful language model trained on a massive dataset of text and code.
* This model can understand and respond to prompts in a creative and informative way, generating different text formats based on the user's input.

**3. Streamlit for User Interface:**

* Streamlit, a lightweight Python library, is used to build the user interface.
* This interface provides a simple and intuitive way for users to interact with the application, enter prompts, and view the generated text.

**4. Flask for Backend Processing:**

* Flask, a web framework, runs on a server and handles the backend logic.
* It interacts with the LlamaCpp model, processes user prompts, and generates text.
* The Flask server acts as a bridge between the user interface and the LLM model.

**5. Overall Functionality:**

* Users interact with the Streamlit interface and provide prompts.
* Streamlit sends the prompts to the Flask server.
* The Flask server utilizes the LlamaCpp model to generate text based on the prompts.
* The generated text is sent back to the Streamlit application.
* Streamlit displays the generated text to the user.

**In essence, the project combines user interaction, a powerful LLM, and web development frameworks to create a user-friendly platform for text generation.**

Sure, here's a user case diagram for this text generation project:

**Actors:**

* User

**Use Cases:**

* **Provide Prompt:** The user enters a prompt text into the Streamlit application's text area.

**System Responses:**

* The application displays a confirmation message indicating the prompt is received.

**Extends:** (Optional)

* The user can choose from pre-defined prompt templates (if implemented).

**Includes:** (Optional)

* The user can edit the provided prompt before submitting it.

**Notes:**

* This use case focuses on the core functionality of submitting a prompt.

**Diagram:**

Code snippet

user --> Provide Prompt: prompt\_text

system --> Displays confirmation message

Use code [with caution.](https://d.docs.live.net/faq#coding)

content\_copy

**Explanation:**

The user actor interacts with the system through the "Provide Prompt" use case. They enter their desired prompt text into the application's text area. The system responds by displaying a confirmation message indicating the prompt is received. The optional extensions and includes represent potential functionalities that can be added to enhance user experience, such as pre-defined templates or editing capabilities.

This user case diagram depicts the high-level interaction between the user and the system for submitting prompts for text generation.