KALASALINGAM ACADEMY OF REASERCH AND EDUCATION

Code Crunch 2024
(National Level Hackathon)

Low-Level Design

on

End-to-End Text Generation Pipeline

Team members:

B. Karthik
K. Uday Kiran
K. Pavani

Table of Contents

I. Introduction

1. 1.Scope of the Document

The main scope of the Document is to develop a text-to-text AI Bot using python, AI and to develop an Interface using the MERN Stack. Develop a web-based text generation application optimized for specific domains like legal, healthcare, or creative writing, using GPT-2 or GPT-3. The project involves data collection, model fine-tuning, and deploying the application on a MERN stack.

1. 2. Intended audience

A domain-optimized text generating application employing GPT-2 or GPT-3 may have different target users depending on the particular domain (e.g., legal, healthcare, or creative writing) for which it is optimized. Professionals, instructors, students, and businesses from a range of industries are among the target users of this program as they may gain from AI-assisted text production that is customized to meet their domain-specific requirements. The application's goals are to increase productivity, lower expenses, and provide written outputs in specialized sectors of higher quality.

Here is a thorough analysis of each domain's prospective audience.

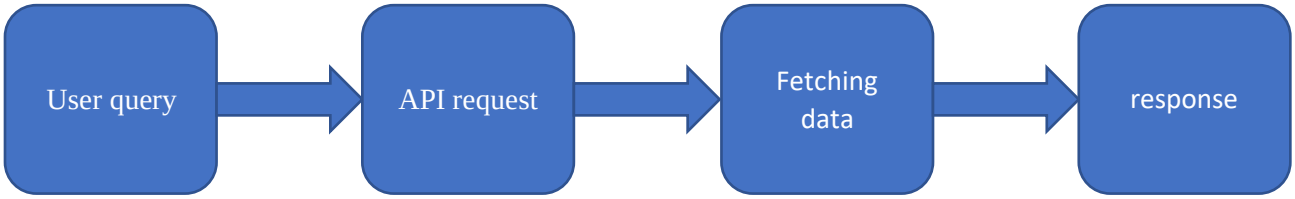
- **Accuracy and Reliability**: Especially critical in legal and healthcare, where generated content must adhere to high standards of accuracy.
- **Ethical Use**: Ensuring the generated text is used ethically, particularly in sensitive domains like legal and healthcare.
- **User Training**: Some users may require training to effectively utilize AI-generated text, especially for domain-specific applications.

1. 3. System overview

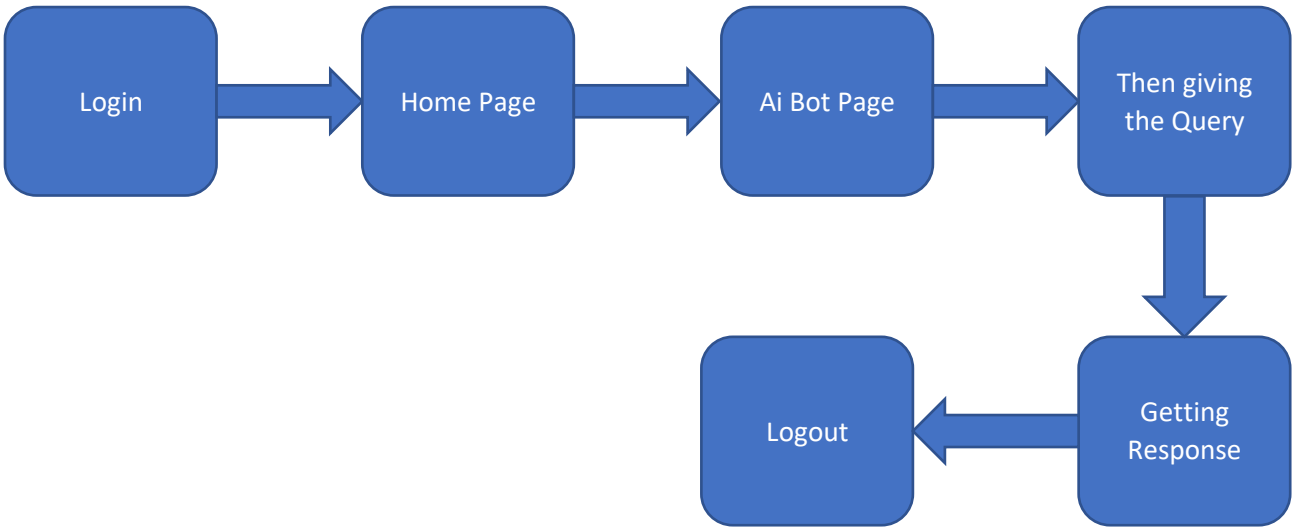
This is a thorough system overview that explains every part of the text creation program and how it fits into the larger architecture. The overview describes the main technologies used at each level, as well as how data moves through the system and interacts with its many subsystems. The modular architecture of the text generating application integrates front-end user interface design, backend APIs, model training, and data collecting. The principal objective is to establish a smooth and consistent interface for users to submit prompts and obtain comprehensible, domain-optimized text answers.

II. Low Level System Design

1.1. Sequence Diagram



1.2 Navigation Flow/UI Implementation



1. 3 Screen validations, Defaults and Attributes

- **Asynchronous Calls**: Each request and response pair represents a call from one component to another, highlighting the asynchronous nature of the interactions.
- **Error Handling**: Not explicitly shown but should be incorporated to handle cases where the model fails to generate text or the servers are down.
- **Data Validation**: Input validation happens at the **Frontend** and **Python API** stages to ensure proper handling of data.
- **Modular Structure**: Each component is isolated, promoting easy debugging, maintenance, and updates without disrupting the entire system.

1.4 Client-Side Validation Implementation

Client-side validation is crucial for ensuring that the inputs provided by the user are correct and in the desired format before being sent to the server. This helps to reduce server load, improve user experience, and prevent malicious data from being processed.

- **Required Fields**: Ensure that the input field is not empty.
- **Input Length**: Set a minimum and maximum length for the input to avoid excessively short or long prompts.
- **Character Restrictions**: Disallow special characters if needed, based on the requirements of your model.
- **Input Format**: Check for any specific format requirements, like alphanumeric content only, depending on the domain.

1.5 Server-Side Validation Implementation

Server-side validation is critical for ensuring data integrity and security, even when client-side validation is already in place. Unlike client-side validation, which can be bypassed, server-side validation ensures that the backend checks all inputs rigorously before processing, which helps prevent malicious data from affecting the system.

- **Input Validation**: Use Pydantic models to define the expected input structure and apply validation rules.
- **Error Handling**: Provide meaningful error responses for invalid inputs.
- **Security**: Protect against malicious inputs, such as overly long prompts, special characters, or scripts.

1.6 Components Design Implementation

This component-based design ensures a clear separation of concerns, making the application scalable, maintainable, and easy to debug. Each component handles a specific task, from user interaction to model inference, ensuring a smooth flow of data and user experience.

- **User Input**: The user interacts with the frontend to enter prompts.
- **API Requests**: The Hugging Face API server forwards requests to the Fast API backend.
- **Model Inference**: The FastAPI backend processes the request, runs inference on the model, and returns generated text.
- **Display Output**: The generated text is displayed to the user on the front end.

1.7 Configuration/Settings

- **Security**: Protect sensitive information by using environment variables instead of hard-coded values.
- **Scalability**: Easily adapt settings for different environments, such as development, staging, and production.
- **Maintainability**: Centralize configurations to simplify updates and troubleshooting.
- **Portability**: Ensure that settings are consistent across different deployment environments.

This comprehensive configuration setup will ensure that your application runs smoothly, securely, and efficiently, no matter the environment.

1.8 Interfaces to other components

The interfaces between components in your text generation application are crucial for seamless communication and data flow. Interfaces define how different parts of your system interact, ensuring that data is transferred correctly and efficiently between the frontend, backend, AI model, and any external services.

Here's an overview of the interfaces between the major components of your application:

- Frontend to Backend (React to Node.js Server)
- AI Model to GPT Model (Fast API to GPT-2/GPT-3)
- Embedded Space to AI Model (Hugging Face Space to Fast API)
- Deployment and Environment Interfaces

III. Details of other frameworks being used

1.1. Session Management

Session management is a critical aspect of any web application, including your text generation platform, as it ensures secure and reliable interactions between users and the system. Effective session management handles user authentication, maintains session state, and ensures data integrity during user interactions.

Here's a detailed guide on how to implement session management in your application:

Session Management Overview

Session management involves tracking user interactions with the system across multiple requests. It is essential for maintaining user authentication, storing temporary data, and managing user-specific settings.

- **Initialization**: Starts when a user logs in; assigns a unique session ID.
- **Authentication**: Verifies user credentials before creating a session.
- **Validation and Expiration**: Ensures sessions are valid and automatically expire after a set period.
- **Security Measures**: Implement secure cookies, session ID regeneration, and proper session termination to safeguard user data.

1.2 Caching

Caching is a powerful technique used to improve the performance and scalability of your text generation application by temporarily storing frequently accessed data, reducing the need for repeated computations or database queries. Properly implemented caching can significantly speed up response times, decrease server load, and enhance the overall user experience.

Caching Overview

Caching involves storing copies of data that can be retrieved quickly without having to regenerate or re-fetch the original data. In the context of a text generation application, caching can be used at various levels, such as storing AI model responses, user-specific settings, or even fine-tuned model parameters.

IV. UNIT TESTING

Unit testing is a crucial part of developing any software application, including your text generation platform, as it ensures that individual components of your code work as expected. By writing unit tests, you can catch bugs early, validate the correctness of your functions, and maintain code quality over time, especially when making updates or refactoring.

Unit Testing Overview

Unit testing involves testing individual units or components of an application independently from the rest of the system. In the context of a text generation application, this could involve testing functions for data validation, caching mechanisms, prompt handling, AI model response generation, and more.

Key Concepts of Unit Testing

1. Test Frameworks and Tools
2. Writing Unit Tests
3. Mocking and Stubbing
4. Test Coverage
5. Continuous Integration (CI) and Automation

1. Test Frameworks and Tools

Choosing the right test framework is essential for effective unit testing. Below are popular frameworks for testing different parts of your application:

- **Python**: `unittest`, `pytest`, `nose2`
- **JavaScript (react.js)**: Hugging Face API,
- **Integration with CI Tools**: GitHub Actions, Jenkins, Travis CI for automated test runs

2. Writing Unit Test

Unit tests should be focused, testing one small unit of code at a time, such as a function or a method. Each test typically follows a pattern:

- **Arrange**: Set up the environment and prepare the input.
- **Act**: Call the function or method under test.
- **Assert**: Check if the output matches the expected result.

3. Mocking and Stubbing

Mocking and stubbing are techniques used in unit testing to isolate the function under test from external dependencies, such as databases, APIs, or complex computations. This ensures that tests focus only on the logic of the unit.

- **Mocking**: Creating a fake version of an object, function, or service that simulates the behavior of real dependencies.
- **Stubbing**: Replacing a function or method call with a predefined response.

4. Test Coverage

Test coverage measures the extent to which your tests cover your codebase. High test coverage indicates that most of your code paths are tested, reducing the chances of undetected bugs.

- **Tools for Measuring Coverage**:
  - **Python**: `coverage.py`, integrated with `pytest`.
  - **JavaScript**: `Jest` with built-in coverage reporting.

5. Continuous Integration (CI) and Automation

To ensure code quality consistently, integrate unit tests into a CI pipeline. CI tools automatically run tests every time code is pushed or merged, ensuring that changes do not break existing functionality.

- **Hugging Face Tools**: GitHub Actions,Spaces,API's
- **Benefits**: Automated testing on each commit, early detection of bugs, smoother deployments.

Unit testing is vital for maintaining the integrity of your text generation application, ensuring that each component functions correctly and integrates well with the rest of the system.

V. KEY NOTES

Key Notes for Building a Text Generation Application

1. System Architecture

- **Frontend**: React for user interface, handling input and displaying results.
- **Backend**: Hugging Face spaces API to communicate with FastAPI.
- **Model**: GPT-2 or GPT-3 for text generation, fine-tuned for specific domains.
- **Deployment**: Use the MERN stack (MongoDB, Express, React, Node.js) for a full-stack solution, or adapt based on your requirements.

2. Key Components

- **Frontend**:
  - **React Components**: `TextInput` for input handling, `DisplayOutput` for showing results.
  - **Validation**: Client-side checks for input length and content.
- **Backend**:
  - Hugging Face API Acts as a proxy between React and FastAPI, handles API routing and requests.
  - **FastAPI**: Manages model inference, input validation, and API endpoints.
- **Model**:
  - **Fine-Tuning**: Customize GPT-2 or GPT-3 on domain-specific data to improve relevance and accuracy.

3. Configuration and Settings

- **Environment Variables**:
  - Store sensitive data like API keys, database URLs, and configuration settings.
  - Use `.env` files and tools like `dotenv` for environment management.
- **Frontend Configuration**:
  - Centralize API endpoints and application settings in configuration files.
  - Manage React settings using `.env` files for environment-specific configurations.
- **Backend Configuration**:
  - Use environment variables for port settings, API URLs, and security configurations.
- **Model Configuration**:
  - Manage fine-tuning parameters and model paths using environment settings.

    - Load models and tokenizers based on configuration for flexibility.
- **Deployment**:
    - Set up different configurations for development, staging, and production environments.
    - Use Docker for consistent deployment across environments.

4. Validation and Security

- **Client-Side Validation**:
    - Check input length, format, and content before sending requests to the server.
- **Server-Side Validation**:
    - Ensure data integrity and security with rigorous validation using Pydantic in FastAPI.
    - Handle errors gracefully and provide clear error messages.
- **Security Best Practices**:
    - Protect sensitive data using environment variables.
    - Implement CORS and rate limiting to safeguard against abuse.
    - Use HTTPS for secure communication between frontend and backend.

5. Deployment Considerations

- **Scalability**:
    - Prepare for scaling by designing stateless components and considering load balancing.
- **Monitoring**:
    - Set up logging and monitoring to track application performance and errors.
- **Documentation**:
    - Maintain clear and up-to-date documentation for API endpoints, configurations, and deployment procedures.

6. Testing and Debugging

- **Testing**:
    - Write unit tests for individual components and integration tests for end-to-end functionality.
    - Use tools like Jest for frontend testing and Pytest for backend testing.
- **Debugging**:
    - Implement logging at different levels (info, warning, error) for effective

debugging.

- Utilize debugging tools and methods for both frontend and backend issues.

By keeping these key notes in mind, you can build a robust, secure, and efficient text generation application. Each component should be well-integrated, and configurations should be managed effectively to ensure a smooth development and deployment process.

VI. REFERENCES

- Hugging Face. (2023). *Transformers Documentation*. Hugging Face. https://huggingface.co/transformers/
- OpenAI. (2023). *GPT-3 API Documentation*. OpenAI. https://platform.openai.com/docs/guides/gpt
- React. (2024). *React Documentation*. React. https://reactjs.org/docs/getting-started.html
- Node.js. (2024). *Node.js Documentation*. Node.js. https://nodejs.org/en/docs/
- Fast API. (2024). *Fast API Documentation*. Fast API. https://fastapi.tiangolo.com/