PROJECT TITLE: QuickGPT

Technology Stack: MERN Stack, OpenAI API, and ImageKit

Team: Manali Gawade (2035), Mrunmayee Shirodkar (2070), Divya Bhogale (2043), Vrushali Walve (2071)

SUMMARY

This report explains the development of a **QuickGPT application**. The project was inspired by OpenAI's ChatGPT and was built using the MERN stack, which includes MongoDB, Express.js, React.js, and Node.js.

The main objective of this project was to create a smart and responsive web application that can understand user input and generate human-like text responses. When a user enters a prompt, the request is sent to the backend server. The server connects to the GPT model through an API and receives a response. This response is then displayed to the user in real time.

The frontend of the application was developed using React.js. It provides a clean and interactive user interface where users can type messages and view AI-generated responses. The design is responsive, meaning it works properly on both desktop and mobile devices. The frontend is divided into different components such as chat box, message display, input field, and navigation bar to keep the code organized.

The backend was developed using Node.js and Express.js. It handles API requests, manages user authentication, processes chat data, and connects to the database. RESTful APIs were created to manage user information and conversation history.

MongoDB was used as the database to store user details and chat records. It allows flexible data storage and fast access to saved conversations. Collections were created for users, chats, and messages to maintain proper structure.

The project also integrates ImageKit for efficient image storage and delivery. It is used to manage user avatars and other interface images. This improves performance by loading images faster and reducing server load.

The system follows a client-server architecture. The React frontend acts as the client, and the Node.js backend acts as the server. The server communicates with both the database and the AI API. This structure makes the application scalable and easy to maintain.

1. PROJECT OVERVIEW & TECHNOLOGICAL REQUIREMENTS

1.1 Introduction

In recent years, the demand for intelligent AI-based applications has increased rapidly due to the development of Large Language Models (LLMs). These models can understand and generate human-like text, making them useful for chatbots, virtual assistants, and other smart systems.

The main aim of this project is to understand how AI models can be integrated into a normal web application. By building a “ChatGPT Clone,” developers can clearly understand how data flows inside the system. When a user types a message, it is sent from the frontend to a secure backend server. The backend then communicates with the AI model through an API. After processing the request, the AI sends back a response, which is displayed to the user on the frontend.

This project helps in learning how frontend, backend, database, and AI services work together to create a complete intelligent web application.

1.2 Technology Stack

Choosing the right tools is important for good performance, scalability, and smooth development. The following technologies were used in this project:

• **MongoDB:** A NoSQL database used to store user information and chat history. It stores data in a flexible JSON-like format, which makes it easy to manage and update.

• **Express.js:** A backend framework for Node.js used to build APIs and handle server-side routing. It helps manage requests and responses between the frontend and database.

• **React.js (with Vite):** A JavaScript library used to build the user interface. It allows developers to create reusable components and provides fast rendering for better user experience.

• **Node.js:** A runtime environment that allows JavaScript to run on the server side. It handles backend logic and API communication.

• **OpenAI API:** The AI engine that provides access to GPT-3.5 or GPT-4 models. It processes user prompts and generates intelligent text responses.

• **ImageKit:** A platform used to manage and optimize images. It improves performance by delivering images quickly using a Content Delivery Network (CDN).

• **Tailwind CSS:** A modern CSS framework used for designing the user interface. It helps in creating responsive and attractive layouts quickly.

2. BACKEND ENVIRONMENT SETUP & CONFIGURATION

2.1 Server Initialization

The development process starts by setting up the server environment. First, a Node.js project is created using the command **npm init**. This generates a package.json file that manages project details and dependencies.

After initialization, important packages are installed. These include:

- **Express** – to create the backend server and handle API requests.
- **CORS** – to allow communication between frontend and backend running on different ports.
- **Dotenv** – to manage environment variables securely.
- **Mongoose** – to connect and interact with the MongoDB database.
- **OpenAI** – to connect the application with the GPT model API.

These dependencies form the foundation of the backend system.

2.2 Directory Structure

A clean and organized folder structure is created to keep the project easy to manage and maintain. The main structure includes:

- **server.js** – The main entry point of the application. It starts the server and connects all components together.
- **models/** – This folder contains Mongoose schemas that define the structure of the database collections such as users and chats.
- **routes/** – This folder contains API route definitions. It manages different endpoints for handling user requests.
- **.env** – A file used to store sensitive information like database URLs and API keys.

Organizing files in this way improves readability and makes the project scalable.

2.3 Environment Variables

Security is very important in web applications. Instead of directly writing sensitive information in the code, a **.env file** is used.

The following variables are stored securely inside the .env file:

- **MONGO_URI** – The connection string for the MongoDB database.
- **PORT** – The port number where the server will run.
- **OPENAI_API_KEY** – The secret API key used to access the OpenAI GPT model.

Using environment variables protects sensitive data and makes the application safer and more professional.

3. DATABASE DESIGN & SCHEMA MODELING

3.1 Connecting to MongoDB

The application connects to MongoDB using **Mongoose**, which is a library that helps manage database operations in Node.js. The project uses **MongoDB Atlas**, which is a cloud-based database service.

MongoDB Atlas allows the database to be hosted online, making it accessible from anywhere. It also provides scalability, meaning the database can handle more users and data as the application grows.

By connecting through the **MONGO_URI** stored in the .env file, the server securely establishes a connection with the database before handling any user requests.

3.2 Schema Definitions

To organize the data properly, Mongoose schemas are created. A schema defines the structure of the data stored in the database.

Two main data models are used in this project:

User Schema:

This schema stores user account information. It includes fields such as name, email, password, and creditBalance. The creditBalance field is used to manage the number of AI requests a user can make.

Chat Schema:

This schema stores conversation details. It includes the userId (to identify which user the chat belongs to) and an array of messages. Each message contains two fields:

- role (user or assistant)
- content (the actual message text)

This structure allows the application to store and retrieve complete conversation history for each user.

4. OPENAI API INTEGRATION & LOGIC

4.1 OpenAI API Integration

The backend imports the **OpenAI** library to connect the application with the GPT model. The API key stored in the **.env file** is used to configure the OpenAI client.

By using environment variables, the API key remains secure and is not exposed in the source code. This setup creates a secure OpenAI client instance on the server side.

Whenever a user sends a message, the backend uses this client to communicate with the AI model and generate a response.

4.2 The Chat Completion Endpoint

A new API route is created, usually named **/api/chat**. This route handles POST requests from the frontend.

When a user sends a prompt, the frontend sends the message to this endpoint. The server receives the request and prepares a message array that includes the user's input.

Then, the server calls the method:

openai.chat.completions.create()

using the model **gpt-3.5-turbo**.

The AI processes the message and generates a response. This response is then sent back to the frontend and displayed to the user in the chat interface.

This endpoint is the core part of the application because it connects user input with AI-generated output.

5. FRONTEND ARCHITECTURE (REACT & TAILWIND CSS)

5.1 Project Initialization with Vite

The frontend development starts by creating a React project using **Vite**. Vite is chosen because it is faster and more efficient than Create React App.

One of the main advantages of Vite is **Hot Module Replacement (HMR)**. This feature allows changes in the code to appear instantly in the browser without reloading the entire page. It makes development quicker and smoother.

Using Vite helps improve performance and reduces build time during development.

5.2 Tailwind CSS Setup

To design the user interface, **Tailwind CSS** is installed and configured. Tailwind is a utility-first CSS framework that allows styling directly using predefined classes.

Instead of writing separate CSS files, developers can use classes like:

- **flex** – for flexible layouts
- **p-4** – for padding
- **bg-[#343541]** – for background color

With Tailwind, it becomes easy to create a clean, dark, and modern design similar to ChatGPT. It also helps in building responsive layouts that work well on different screen sizes.

6. COMPONENT BREAKDOWN & UI IMPLEMENTATION

6.1 Sidebar Component

The Sidebar component is placed on the left side of the screen. It contains the **"New Chat"** button and a list of recent chat conversations.

Users can click on "New Chat" to start a fresh conversation. The list of previous chats allows users to quickly switch between different conversations.

The sidebar has a dark theme to match the overall design of the application. On mobile devices, it becomes responsive, meaning it adjusts its size or can be toggled for better viewing.

6.2 Main Component

The Main component is the main area where the active conversation is displayed. It acts as a container for different sections of the chat interface.

It includes:

- **Header** – Displays the application name or user details.
- **Chat Container** – A scrollable section where messages appear. Users can scroll up to see previous messages.
- **Bottom Box** – A fixed input area at the bottom where users type their messages and send prompts.

This structure ensures a clean and organized chat layout.

6.3 Message Component

The Message component is responsible for displaying individual chat messages in the conversation.

User messages are aligned to the **right side** and have a different background color to make them easily identifiable.

AI-generated messages are aligned to the **left side** and include the GPT icon. This visual difference helps users clearly understand who sent each message.

This design makes the chat interface simple, readable, and user-friendly.

7. STATE MANAGEMENT USING CONTEXT API

7.1 The Need for Global State

In this project, React's **Context API** is used to manage global state. Global state means data that needs to be shared between multiple components.

For example, values like:

- **input** (current user message)
- **recentPrompt** (latest question asked)
- **prevPrompts** (previous chat history)
- **loading** (to show when AI is generating a response)

Instead of passing these values from one component to another using props (called prop drilling), the Context API allows all components to access shared data directly.

This makes the code cleaner, easier to manage, and more scalable.

7.2 Typing Effect Logic

One important feature of the application is the **typing effect**, which makes the AI response appear gradually, similar to real ChatGPT.

To create this effect, a JavaScript function is used. The function takes the full response text and loops through it character by character. Each character is added to the screen one at a time with a small delay.

This creates a smooth streaming effect and improves the user experience by making the chat feel more interactive and realistic.

8. IMAGEKIT INTEGRATION & ASSET MANAGEMENT

8.1 Role of ImageKit

ImageKit is used in this project to manage and deliver images efficiently. It stores application assets such as logos and the default ChatGPT avatar.

One of the main advantages of ImageKit is that it automatically resizes and optimizes images based on the user's device and screen size. This ensures that images load quickly without affecting quality.

By using ImageKit, the application improves performance and provides a faster and smoother user experience.

8.2 Implementation

On the frontend, images are displayed using the **<IKImage>** component provided by ImageKit or standard **<img>** tags that use the ImageKit URL.

Since the images are delivered through ImageKit's Content Delivery Network (CDN), bandwidth usage is reduced and load times are faster.

This makes the application more efficient, especially when accessed from different locations or devices.

9. SYSTEM TESTING & DEBUGGING

9.1 Backend Testing

Backend testing is an important step to ensure that the server, database, and API integrations are working correctly. In this project, API endpoints are tested using tools such as **Postman**. Postman allows developers to send GET and POST requests to the server and check whether the correct response is returned.

During testing, the **/api/chat** endpoint is checked by sending a sample user prompt in a POST request. The response from the OpenAI API is verified to ensure that the AI-generated message is returned properly. The response status codes (such as 200 for success or 500 for server error) are also monitored to confirm that the API is functioning correctly.

Common backend issues that are checked include:

- Incorrect **MONGO_URI** connection strings, which may prevent the server from connecting to MongoDB Atlas.
- Missing or incorrect **OPENAI_API_KEY**, which can stop the AI integration from working.
- Expired or insufficient **OpenAI credits**, which may result in failed API responses.
- CORS errors that block communication between frontend and backend.
- Server crashes due to unhandled errors or incorrect request formats.

Error handling mechanisms are also tested to ensure that the server returns meaningful error messages instead of crashing. Logs in the terminal are monitored to detect and fix issues quickly.

Through proper backend testing, the reliability, security, and performance of the application are improved.

9.2 Frontend Testing

Frontend testing ensures that the user interface works smoothly and provides a good user experience. The React application is tested in different browsers and screen sizes to confirm that it is fully responsive.

Special attention is given to mobile responsiveness. On smaller screens, the **sidebar should collapse or become toggleable**, so it does not take too much space. The chat container should remain scrollable, and the input box at the bottom should always stay accessible for typing new messages.

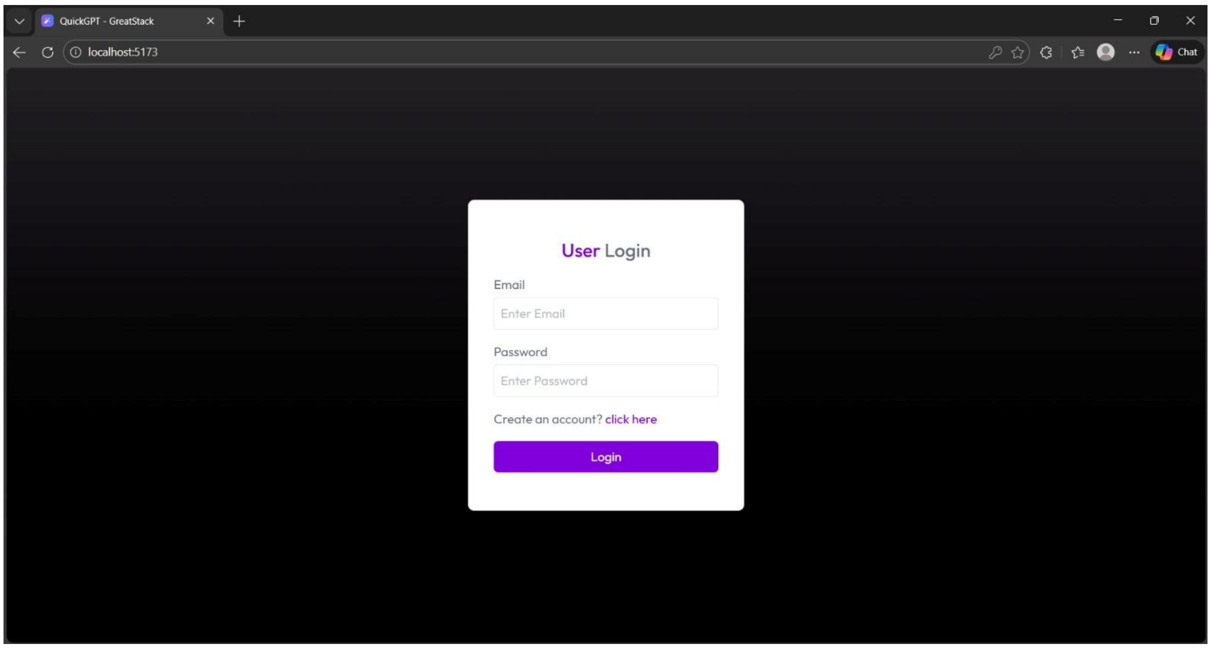
Other frontend checks include:

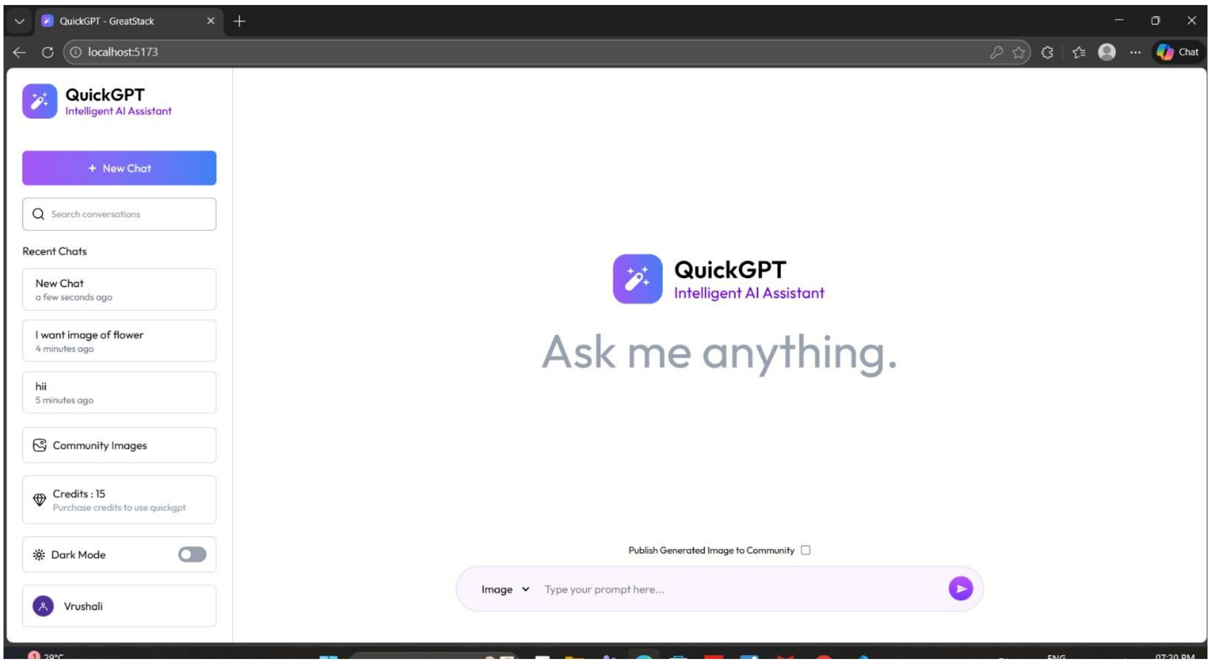
- Verifying that messages appear correctly in the chat interface.
- Ensuring user messages are aligned to the right and AI messages to the left.
- Confirming that the typing effect works smoothly without freezing.
- Checking that loading indicators appear while waiting for AI responses.
- Testing button functionality such as “New Chat” and message submission.
- Ensuring there are no UI glitches or broken layouts.

The application is also tested for performance to ensure fast response times and smooth rendering. Console errors in the browser’s developer tools are monitored and resolved.

By performing thorough frontend testing, the application becomes more user-friendly, stable, and ready for real-world usage.

PUC (Proof Of Concept) :





QuickGPT - GreatStack

localhost:5173

QuickGPT
Intelligent AI Assistant

+ New Chat

Search conversations

Recent Chats

New Chat
a few seconds ago

I want image of flower
4 minutes ago

hii
5 minutes ago

Community Images

Credits : 15
Purchase credits to use quickgpt

Dark Mode

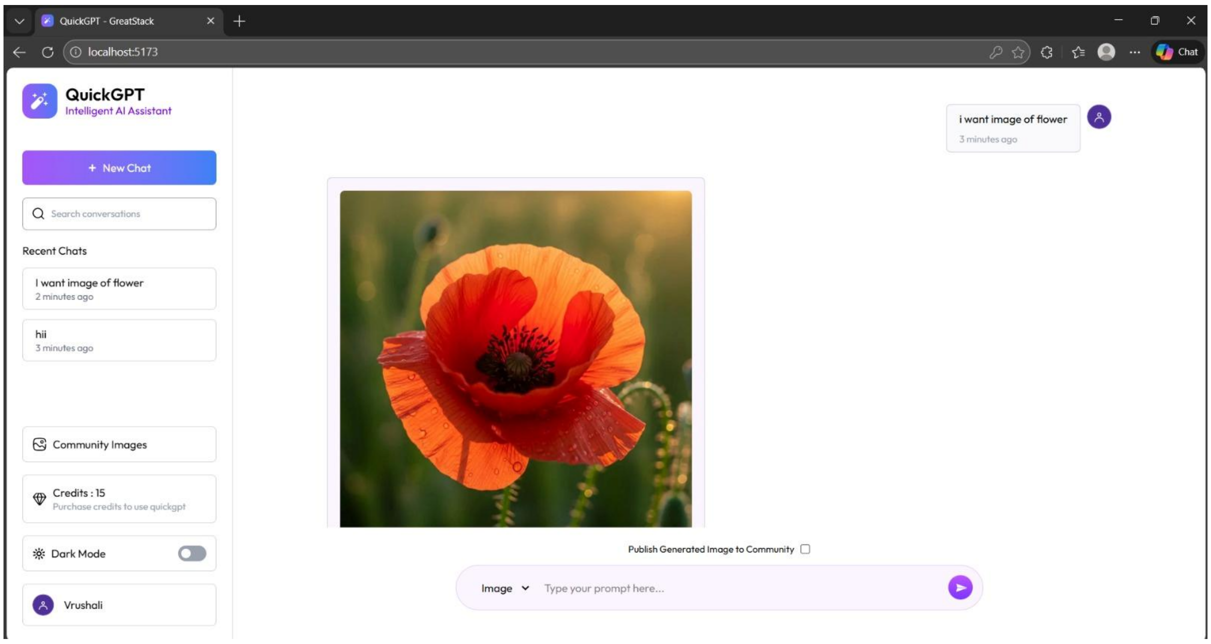
Vrushali

QuickGPT
Intelligent AI Assistant

Ask me anything.

Publish Generated Image to Community

Image — Type your prompt here...



QuickGPT - GreatStack

localhost:5173

QuickGPT
Intelligent AI Assistant

+ New Chat

Search conversations

Recent Chats

I want image of flower
2 minutes ago

hii
3 minutes ago

Community Images

Credits : 15
Purchase credits to use quickgpt

Dark Mode

Vrushali

i want image of flower
3 minutes ago

Publish Generated Image to Community

Image — Type your prompt here...

10. CONCLUSION & DEPLOYMENT STRATEGY

10.1 Project Achievement

This project successfully demonstrates the ability to design and develop a complete AI-powered web application using modern web technologies. By using the MERN stack along with the OpenAI API, the system integrates frontend design, backend logic, database management, and artificial intelligence into one working platform.

The application is capable of receiving user input, securely processing it through the backend, communicating with the GPT model, and displaying intelligent responses in real time. It also includes structured database storage, organized project architecture, responsive UI design, and image optimization through ImageKit.

Through this project, important concepts such as API integration, environment variable security, state management, component-based architecture, and cloud database connectivity were successfully implemented. Overall, the project proves that a scalable and intelligent conversational system can be built using standard web development tools.

10.2 Deployment Recommendations

For deploying the application in a production environment, proper hosting platforms should be used for both frontend and backend.

The **frontend** can be deployed on platforms like **Vercel** or **Netlify**. These platforms are optimized for React applications and provide fast global content delivery. They also support automatic deployment through GitHub integration.

The **backend** can be deployed on services such as **Render** or **Railway**, which support Node.js applications. These platforms allow environment variables to be securely configured and ensure that the server runs continuously.

Additionally, **MongoDB Atlas** should be properly configured for production. The IP whitelist settings must allow connections from the backend hosting platform. Environment variables such as MONGO_URI and OPENAI_API_KEY must be securely added in the hosting platform's settings.

Proper deployment ensures that the application is accessible online, secure, scalable, and ready for real-world users.

10.3 Future Scope

Although the project is fully functional, there are several improvements that can be added in future versions to enhance functionality and user experience.

One major improvement is **User Authentication using JWT (JSON Web Tokens)**. This would allow users to securely register, log in, and maintain sessions. It would improve security and provide personalized chat experiences.

Another enhancement is **Chat History Persistence and Management**. While basic chat storage is implemented, future versions can include advanced features such as renaming chats, deleting conversations, organizing chats into folders, and searching previous messages.

The application can also be improved by adding **Voice Input functionality using the Web Speech API**. This would allow users to speak instead of typing, making the application more interactive and accessible.

Additional future features may include:

- Dark/Light theme toggle
- Streaming responses using real-time API handling
- Payment integration for credit-based usage
- Multi-language support
- AI image generation integration

In conclusion, this project lays a strong foundation for building advanced AI-based web applications, and with further enhancements, it can evolve into a production-level intelligent assistant platform.