NATWEST Hackathon

Report on IDE Plugin for Large Language Model (LLM) Integration

Team Details :

1. **Arpan Kumar :** arpan21020@iiitd.ac.in | LinkedIn
2. **Rajat Jaiswal :** rajat21184@iiitd.ac.in | LinkedIn

Github : https://github.com/arpan21020/LLMs-IDE-Plugin

Introduction

The "IDE Plugin for Large Language Model (LLM) Integration" project is aimed at enhancing the coding and development experience within integrated development environments (IDEs) by allowing seamless integration of LLM functionalities. With the exponential growth of artificial intelligence, LLMs like GPT-4 have shown tremendous potential in aiding developers by generating code snippets, providing explanations, auto-completing code, and more. This plugin was designed to leverage these powerful AI-driven tools directly from the IDE, making it more accessible and user-friendly for developers working across various programming languages.

The plugin supports for `vs-Code` IDE only .Through simple API integration with providers like `gpt-35-turbo-0613 , llama3 , gemini` , this plugin allows developers to input their API key and interact directly with the LLM, generating text-based outputs based on their prompts. The overall objective was to simplify developer workflows by embedding advanced AI functionalities directly into the IDE, thus enhancing productivity and innovation.

Problem Solving Approach

- **Identify the Need**

As developers, we often find ourselves switching between multiple tools—IDEs for coding, browsers for looking up documentation, and maybe even ChatGPT for language model assistance. All that switching takes time and interrupts your workflow. This is where our project comes in. To create an IDE plugin that integrates directly with Large Language Models like OpenAI's GPT, making coding smarter and faster.

- **Define Objectives**

  The primary goals of the plugin are:

  - Seamless integration of LLM functionalities within popular IDEs, specifically Visual Studio Code (VSCode).
  - Easy access to multiple language models (e.g., Gemini, Llama3, GPT-3 Turbo) through simple API integration.
  - A user-friendly interface that allows developers to input prompts and receive outputs directly in their coding environment.

Use Case for IDE Plugin for Large Language Model Integration

1. Code Generation

Developers can quickly generate boilerplate code or complete functions by providing high-level descriptions. This reduces time spent on repetitive coding tasks, allowing developers to focus on logic and design.

2. Code Explanation

The plugin can explain new or complex code in simple terms, aiding developers in understanding unfamiliar codebases or concepts, which is particularly useful for onboarding junior developers.

3. Debugging Assistance

The plugin assists in identifying bugs or suggesting fixes based on error messages or problematic code snippets. Developers can input the error message or code to receive contextual help.

4. Auto-completion and Suggestions

Leveraging LLM capabilities, the plugin provides smart code auto-completion and suggestions, improving coding efficiency. Developers can receive suggestions based on best practices and common patterns.

5. Natural Language Queries

The plugin allows developers to interact with the IDE using natural language. For instance, they can ask questions about specific libraries or frameworks, and the LLM provides relevant information, examples, or links to documentation.

6. Learning and Exploration

Serving as an educational tool, the plugin enables developers to learn about new technologies, frameworks, and coding practices through interactive queries and examples generated by the LLM.

Installation

1. Prerequisites:

Before installing the LLMs IDE Plugin, ensure that you have the following:

- Visual Studio Code (VSCode): Make sure you have VSCode installed on your system. You can download it from Visual Studio Code.
- Node.js and npm: The plugin is built using Node.js, so you need Node.js and npm (Node package manager) installed. You can download them from Node.js.

2. Clone the Repository:

Clone the plugin repository to your local machine using Git:

-> git clone https://github.com/arpan21020/LLMs-IDE-Plugin.git

-> cd <repository-directory>

3. Install Dependencies:

After cloning the repository, navigate to the plugin's directory and run the following command to install the required npm packages:

-> npm install

This will install all the dependencies listed in the `package.json` file into the `node_modules/` folder.

4. Build the Plugin:

If there are any build scripts defined in `package.json`, you may need to run them:

->npm run build

5. Open the Plugin in VSCode:

Open the project folder in VSCode:

1. Press **F5** to launch a new VSCode window with the plugin loaded in the "Extension Development Host" mode.

6. Install the Plugin Locally:

To install the plugin locally:

1. Install vsce using, command “npm install -g vsce”
2. Navigate to Your Extension's Directory
3. Package the Extension, command “vsce package”
4. Open the command palette in VSCode (`Ctrl+Shift+P` on Windows/Linux or `Cmd+Shift+P` on macOS).
5. Type `Extensions: Install from VSIX...` and select the option.
6. Browse to the directory where your extension is located and select the `.vsix` file

Configuration

The LLMs IDE Plugin requires certain configurations to connect to various LLM APIs and customize how the models are used within the VSCode environment.

1. API Key Setup:

- The plugin requires API keys to connect to different language models such as **Gemini**, **Llama3**, and **GPT-3 Turbo**.
- When the plugin is launched, it will display a field to input the API key for the selected model. Users must:
    1. **Enter the API key** into the provided input field.
    2. Click **Save API Key** to store the API key securely within the VSCode global state.

The API key for each model can be updated at any time by selecting the model and entering a new API key.

2. Model Selection:

- The plugin supports multiple LLMs, including:
    - **gemini-1.5-flash**
    - **llama3-8b-8192**
    - **gpt-35-turbo-0613**
- Users can select a model from the dropdown menu in the UI. The corresponding API key for the selected model will be retrieved from the global state or prompted if not saved.
- The user can then enter a prompt, and the plugin will send the prompt to the selected model for text generation.

3. Customizing Plugin Behavior:

- The plugin includes several predefined models in the `conversationalLLMs` array within the `extension.js` file:

```
const conversationalLLMs = [
  { name: 'gemini-1.5-flash', func: textGenTextOnlyPromptStreaming },
  { name: 'llama3-8b-8192', func: getGroqChatCompletion },
  { name: 'gpt-35-turbo-0613', func: callAzureOpenAI }
];
```

- To add new models, users can modify this array by adding new entries with the model name and its corresponding API function.

If additional configuration or models are required, users can extend this array with new models and their associated API methods.

4. Webview Content Customization:

- The webview content displayed to the user can be customized by editing the HTML template inside the `getWebviewContent()` function in `extension.js`.

- For instance:
  - Custom styles can be applied by modifying the linked **CSS** file in the `media/` folder.
  - Additional input fields or UI elements can be added to further extend the plugin's capabilities.

5. ESLint Configuration:

- The project utilizes **ESLint** to enforce coding standards and improve code quality. The configuration file `eslint.config.mjs` supports **CommonJS**, **Node.js**, and **Mocha** environments, with the ECMA version set to **2022**. Key rules include warnings for unused variables, reassignment of constants, and unreachable code, allowing developers to maintain clean and efficient code.

6. JSConfig Customization:

- The project includes a `jsconfig.json` file, which configures JavaScript file handling in VSCode. It specifies **Node16** as the module system and **ES2022** as the target ECMAScript version, enabling modern features and type-checking. The `checkJs` option is enabled for error detection, while `node_modules` are excluded from type-checking.

Usage of Plugin

Launching the Plugin:

- Open Visual Studio Code (VSCode) and navigate to the Command Palette by pressing `Ctrl+Shift+P` (Windows/Linux) or `Cmd+Shift+P` (Mac).
- Type and select the command **"LLMs IDE Plugin: Open"** to launch the extension interface.

Selecting a Language Model:

- In the extension interface, a dropdown menu allows users to select from available LLMs, including **Gemini**, **Llama 3**, and **GPT-3 Turbo**.
- Choose a model to be used for text generation.

Configuring the API Key:

- Enter your API key for the selected model in the **API Key** input field. Click on the **"Save API Key"** button to store the key securely.

Entering the Prompt:

- Type your desired prompt in the **Prompt** input field. This prompt serves as the input for the selected language model.

Generating Text:

- Click the **“Generate Text”** button to initiate the text generation process. A loading indicator will appear during the operation.
- The generated text will be displayed in the **Generated Text** area upon completion.

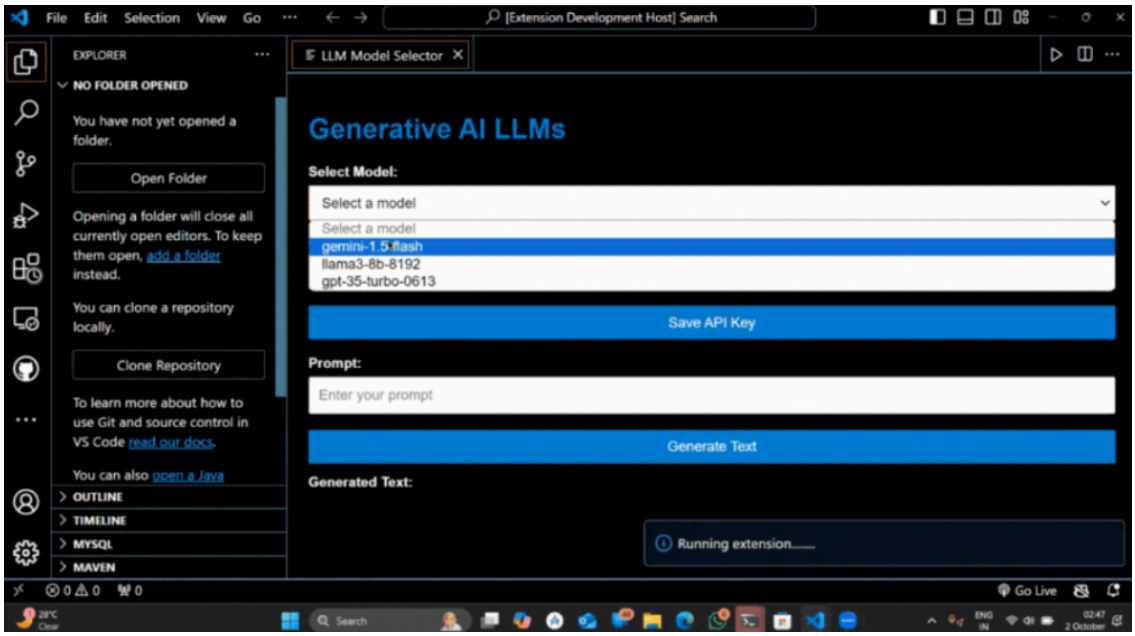
Error Handling:

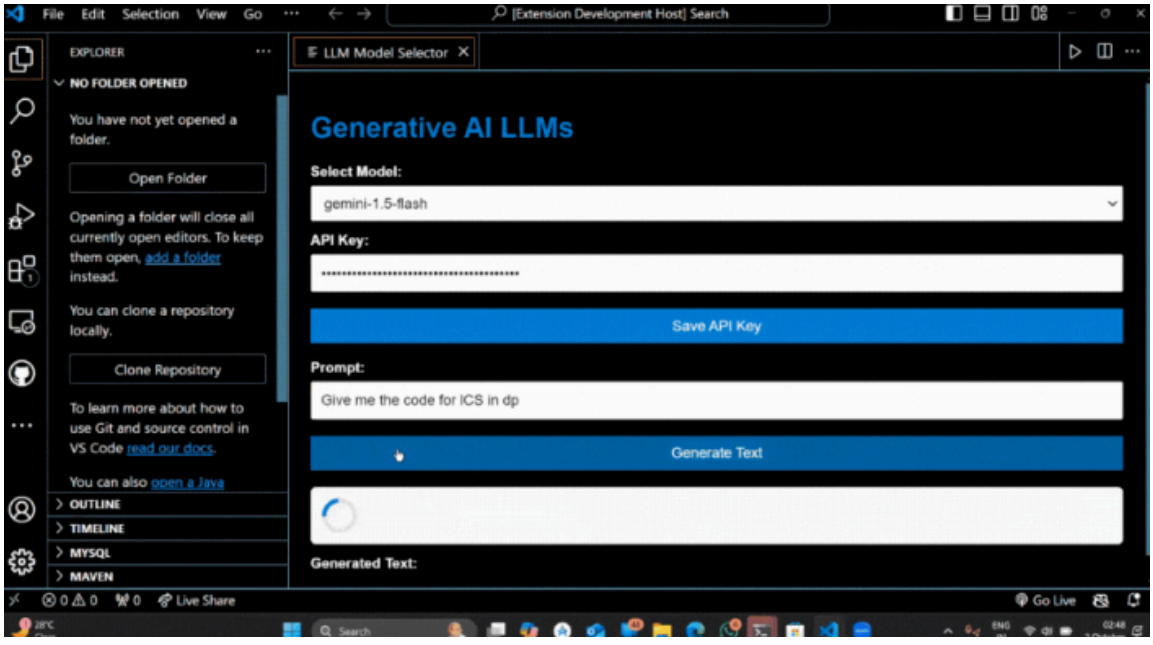
- If the API key is missing or if there are any errors during text generation, an error message will be shown to inform the user.

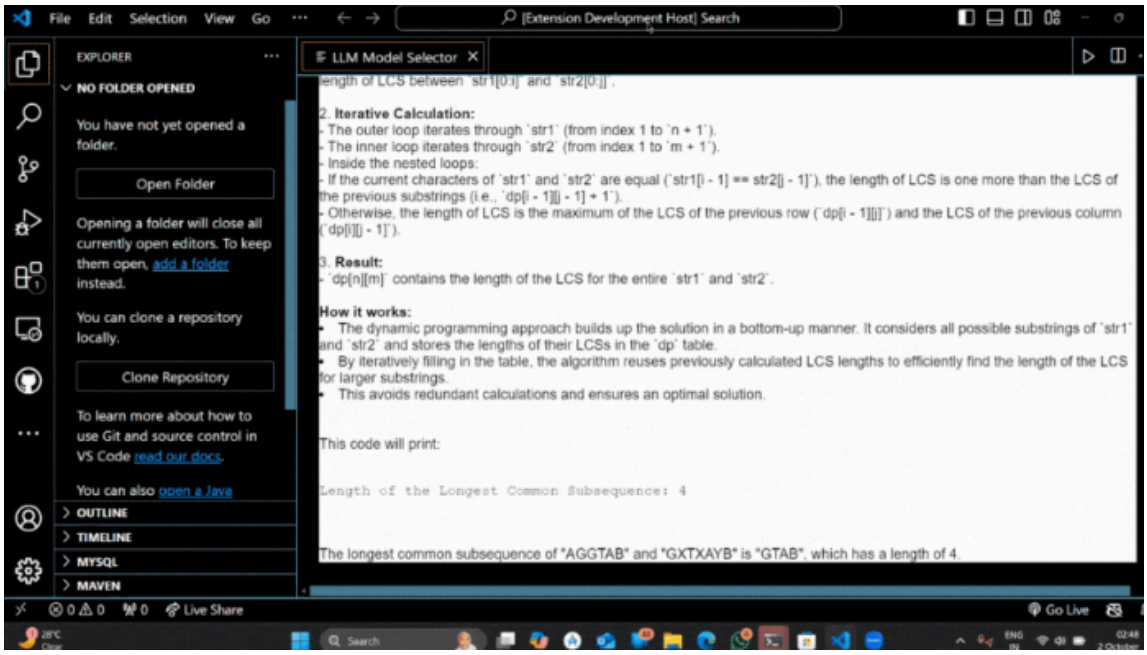
Usage Recommendations:

- Ensure that the API keys used are valid and that any model-specific limitations are acknowledged for optimal performance.

Demo Video


File Edit Selection View Go
[Extension Development Host] Search
EXPLORER
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
Opening a folder will close all currently open editors. To keep them open, add a folder instead.
You can clone a repository locally.
Clone Repository
To learn more about how to use Git and source control in VS Code read our docs.
You can also open a Java
OUTLINE
TIMELINE
MYSQL
MAVEN
LLM Model Selector
Generative AI LLMs
Select Model:
Select a model
Select a model
gemini-1.5-flash
llama3-8b-8192
gpt-35-turbo-0613
Save API Key
Prompt:
Enter your prompt
Generate Text
Generated Text:
Running extension.....
Go Live


File Edit Selection View Go
[Extension Development Host] Search
EXPLORER
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
Opening a folder will close all currently open editors. To keep them open, add a folder instead.
You can clone a repository locally.
Clone Repository
To learn more about how to use Git and source control in VS Code read our docs.
You can also open a Java
OUTLINE
TIMELINE
MYSQL
MAVEN
LLM Model Selector
Generative AI LLMs
Select Model:
gemini-1.5-flash
API Key:
Save API Key
Prompt:
Give me the code for ICS in dp
Generate Text
Generated Text:
Live Share
Go Live


File Edit Selection View Go
[Extension Development Host] Search
EXPLORER
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
Opening a folder will close all currently open editors. To keep them open, add a folder instead.
You can clone a repository locally.
Clone Repository
To learn more about how to use Git and source control in VS Code read our docs.
You can also open a Java
OUTLINE
TIMELINE
MYSQL
MAVEN
LLM Model Selector
length of LCS between `str1[0:i]` and `str2[0:j]`.
2. Iterative Calculation:
- The outer loop iterates through `str1` (from index 1 to `n + 1`).
- The inner loop iterates through `str2` (from index 1 to `m + 1`).
- Inside the nested loops:
- If the current characters of `str1` and `str2` are equal (`str1[i - 1] == str2[j - 1]`), the length of LCS is one more than the LCS of the previous substrings (i.e., `dp[i - 1][j - 1] + 1`).
- Otherwise, the length of LCS is the maximum of the LCS of the previous row (`dp[i - 1][j]`) and the LCS of the previous column (`dp[i][j - 1]`).
3. Result:
- `dp[n][m]` contains the length of the LCS for the entire `str1` and `str2`.
How it works:
• The dynamic programming approach builds up the solution in a bottom-up manner. It considers all possible substrings of `str1` and `str2` and stores the lengths of their LCSs in the `dp` table.
• By iteratively filling in the table, the algorithm reuses previously calculated LCS lengths to efficiently find the length of the LCS for larger substrings.
• This avoids redundant calculations and ensures an optimal solution.
This code will print:
Length of the Longest Common Subsequence: 4
The longest common subsequence of "AGGTAB" and "GXTXAYB" is "GTAB", which has a length of 4.
Live Share
Go Live
Search