SmAs.AI: Using RAG To Make an Enhanced AI-Assistant Platform

Sahil Machkar[1], Girish Patil[2], Kirti Patil[3]
[1,2,3]*Student of the Computer Department, RMD Sinhgad School of Engineering Pune, India*

***Abstract*—AI is a revolutionary industry with significant potential to aid in numerous fields and for different roles and situations. This paper primarily focuses on the use of an AI-based assistant as a browser extension, enabling students to utilise AI as a tool to enhance their productivity without frequent visits to AI websites. This platform aims to provide a multipurpose tool to support students in coding, text generation, or text summarisation.**

***Index Terms*—Assistant platform, Artificial Intelligence, Research Augmented Generation, Web Extensions, Code Completion, Text Summarization, Grammar Correction, Spell-Checking, Large Language Models (LLMs).**

I. INTRODUCTION

Browser extensions are specialised software components designed to augment, customise, or integrate new features into the user's browsing experience or their day-to-day search sessions. [1] These are an indispensable part of the modern internet browsing experience, so utilising this feature can be beneficial for an AI assistant application, as there is no need for a dedicated web browser to host the AI assistant. Instead, the extension can be toggled on or off as needed by the user, which helps to manage the usage of the tool as well.

AI, also known as artificial intelligence, is a revolutionary and lucrative tool for individuals seeking to enhance their productivity by using AI tools for code completion, text summarisation, Research applications, and more. The modern era of the internet is equipped with numerous AI tools and browser extensions made for increasing productivity for people in numerous fields. As well as the overall knowledge required for usage of such assistants is minimal on the user's end. The main issue many people, including students, face is that they lack a single platform to address multiple needs. The main problem is that there isn't a single platform for assistance for multiple uses. AI tools are platform-dependent, and those that are extension-based have a tendency to be limited to a single use-case scenario. This is a problem that has been noticed with many AI tool platforms.

II. LITERATURE SURVEY

The article notes that, despite the "explosive growth of generative artificial intelligence," there have been "few studies on building general-purpose multimodal AI assistants and copilots tailored to pathology". The rise of multimodal large language models (MLLMs) is poised to open a "new frontier for computational pathology," one that emphasises natural language and human interaction as key components. This paper, therefore, addresses this gap by developing and evaluating PathChat, a "vision-language generalist AI assistant for human pathology"

Svyatkovskiy et al. [3] used deep learning models trained on a dataset in AST format. They compared this approach against frequency-based and Markov Chain models. Wang Y. et al. [4] also applied a preprocessing method to convert code to AST format. They then developed an LSTM network enhanced with an Attention mechanism and a Pointer mechanism. Arkesteijn et al. [7] utilized the neural network from Wang et al. [4] but added Byte Pair Encoding (BPE) to the process. Belevskiy [6] focused specifically on function call completion. This study developed preprocessing strategies like the "N-chars" method, which uses the N characters preceding the object calling the function to generate a suggestion. Scharrenburg [1] trained models on a Java dataset, experimenting with different architectures and hyperparameters.

AI-based chatbots are widely used to automate FAQ responses across domains like education, banking, healthcare, and e-commerce. Early systems were rule-

based or pattern-matching, offering quick replies but poor flexibility. Modern bots use frameworks such as RASA and Dialogflow, which handle intent detection and dialogue flow efficiently. Machine learning methods like KNN, RNN, and LSTM improve understanding and accuracy but require large datasets and computational resources. Overall, current chatbots achieve good accuracy but still face challenges in handling diverse user inputs, data limitations, and maintaining natural conversations. Future work should focus on hybrid AI models and multimodal (text + image) understanding for more reliable FAQ automation.

III. PROBLEM STATEMENT

To Design and develop a lightweight AI-powered browser extension to offer universal, real-time grammar correction, code assistance, and text explanations across all websites. This project focuses on the research, design, and proof-of-concept implementation of the "SmartAssist.AI" proactive assistant. The scope includes the end-to-end development of the system, from data collection to final model evaluation.

In-Scope (Features & Deliverables):

- Frontend Widget: A fully functional floating widget built with HTML, CSS, and JavaScript.
- Backend API: A Python (FastAPI) server that processes requests and serves the machine learning model.
- Reactive AI Features: Integration of an external LLM (e.g., OpenAI API) to enable "Explain This," "Grammar Check," and "Code Assistant" features.
- Proactive ML Model: Development, training, and deployment of a Decision Tree Classifier to predict user intent based on behavioral data (e.g., scroll speed, hover time).
- Data Collection: Implementation of a simple mechanism within the widget to log user interaction data for training the model.
- Delivery: The final application will be packaged as a "sideloadable" Chrome Extension for demonstration and evaluation.
- Evaluation: The project report will include a formal evaluation of the predictive accuracy, precision, and recall of the proactive model.

Out-of-Scope:

- Public Deployment: The project is a proof-of-concept and will not be deployed to a production environment to handle public, high-volume traffic.
- User Dashboard: A public-facing web dashboard for users to sign up, manage accounts, or customize their widget is not included.
- Chrome Web Store: The extension will be for demonstration and "sideloading" only, not published to the official Chrome Web Store.
- Advanced Features: Features such as user voice commands, real-time collaboration, or support for languages other than English are considered outside the scope.

Goals & objectives

1. To develop a full-stack application architecture, consisting of a lightweight JavaScript frontend widget (packaged as a Chrome Extension) and a scalable Python (FastAPI) backend API.
2. To integrate general-purpose Natural Language Processing (NLP) models to provide a baseline of reactive features, such as on-demand text explanation and grammar correction.
3. To research and implement a Decision Tree Classifier algorithm to model user behavior by training it on a dataset of real-time interaction patterns (e.g., scroll speed, hover time, element context).
4. To implement the proactive feature by using the trained model to predict user intent and dynamically suggest the most relevant AI action.
5. To evaluate the performance of the predictive model by measuring its accuracy, precision, and recall, thereby validating the success of the research hypothesis.

IV. SOFTWARE CONTEXT

1. Frontend: JavaScript, HTML, CSS (packaged as a Google Chrome Extension).
2. Backend: Python (using the FastAPI framework).
3. Proactive ML Model: Scikit-learn (Decision Tree Classifier).
4. Reactive AI Model: OpenAI API (for NLP tasks).
5. Communication: REST API (using JSON).

MAJOR CONSTRAINTS

Data Bottleneck:

The biggest constraint is the time and effort needed to collect and accurately label user behavior data required to train the proactive AI model.

API Reliance:

The project depends on external APIs (like OpenAI), which impose cost constraints, speed (latency), and potential rate limits.

Website Compatibility:

As a browser extension, the widget might break or conflict with the code and styles of complex, modern websites.

Proof-of-Concept Scope:

The system is a prototype and not designed to handle the security, scalability, or robustness needed for a large, public user base.

AI Hallucination:

Even when using RAG for mitigation from such issues, hallucination of the AI model is still a viable issue, due to how these models work.

V. PROPOSED SYSTEM ARCHITECTURE

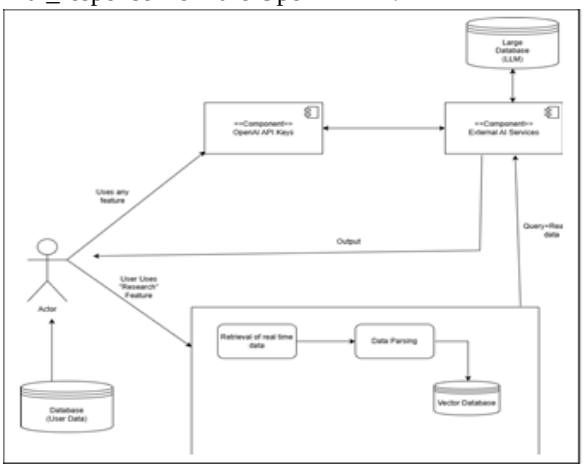
The main architectural diagram is shown above in Fig. 1. The user interacts with the front end, i.e., the floating widget of our AI assistant. This floating widget can be clicked to expand into a small window, revealing a text box and space to store previous session replies or other options, such as text correction, code completion or code analysis.

Data Description

The system utilizes two primary types of data:

1. User Behavior Data (for the Proactive Model):
- Purpose: To train the Decision Tree algorithm.
- Format: A structured dataset, such as a CSV file.
- Fields: Includes features like scroll_velocity, hover_time, element_type, and the target label user_intent (e.g., 'NeedsExplanation', 'IsBrowsing').

2. AI Interaction Data (for Reactive Features):
- Purpose: To process the user's live requests.
- Format: JSON objects.
- Fields: A request containing the selected_text and action_type, and a response containing the ai_response from the OpenAI API.



Architectural diagram of a RAG-based integrated platform Internal Software Data Structure

Data structures exchanged among software components are mainly formatted as JSON objects for API communication.

- AI Interaction Request:

A JSON object sent from the frontend widget to the backend API. It includes the selected_text (string) and action_type (string). The AI service utilizes this data.

- Behavioral Data Packet:

A JSON object sent from the frontend to the backend's BehaviorAnalyticsService. It contains features derived from user behavior, such as scroll_velocity (float), hover_time (float), and element_type (string).

- AI Interaction Response:

A JSON object sent from the backend API back to the frontend widget, containing the ai_response (string).

Global Data Structure

These are data structures available to major portions of the architecture.

- User Preferences: A global settings object loaded on the client-side. It is read by the WidgetUI to configure its state, such as enabling or disabling the proactive features.
- Proactive Model (In-Memory): The trained Decision Tree Classifier, once loaded from its .pkl file into the backend's memory, acts as a global,

read-only data structure. It is accessed by all concurrent worker processes to make real-time predictions.

Temporary Data Structure

These are files or objects created for interim use.

- AI Interaction Object: This JSON object is explicitly defined as a "temporary data object". It exists only for the duration of a single reactive API request and is discarded once the response is sent.
- System Log Entry: A log entry recording a system event (e.g., an API error or a successful prediction). These entries are temporary in memory before being written to a persistent log file.

Database Description

This describes the persistent files and databases used by the application.

- User Behavior Dataset:

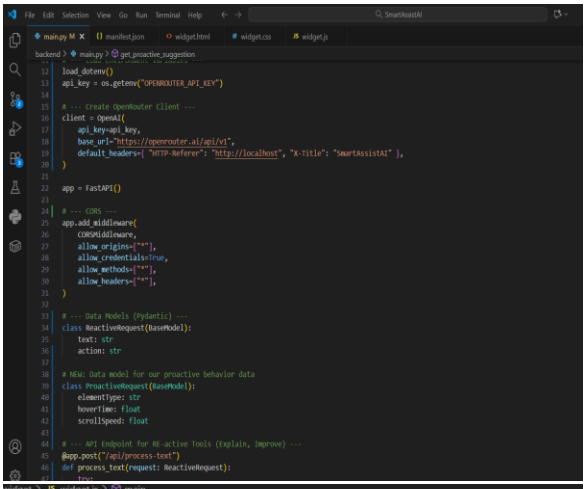
This is the primary dataset for the research component, stored as a structured file (e.g., a CSV file). It contains the labelled Behavior Datapoint records used to train the Decision Tree model.

- Proactive Model File:

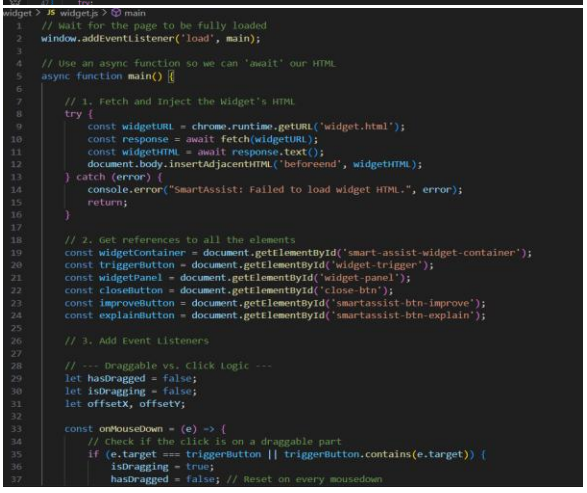
This is the persistent, trained .pkl file that stores the output of the machine learning research. It is loaded by the backend API to make live predictions.

- Conversation DB:

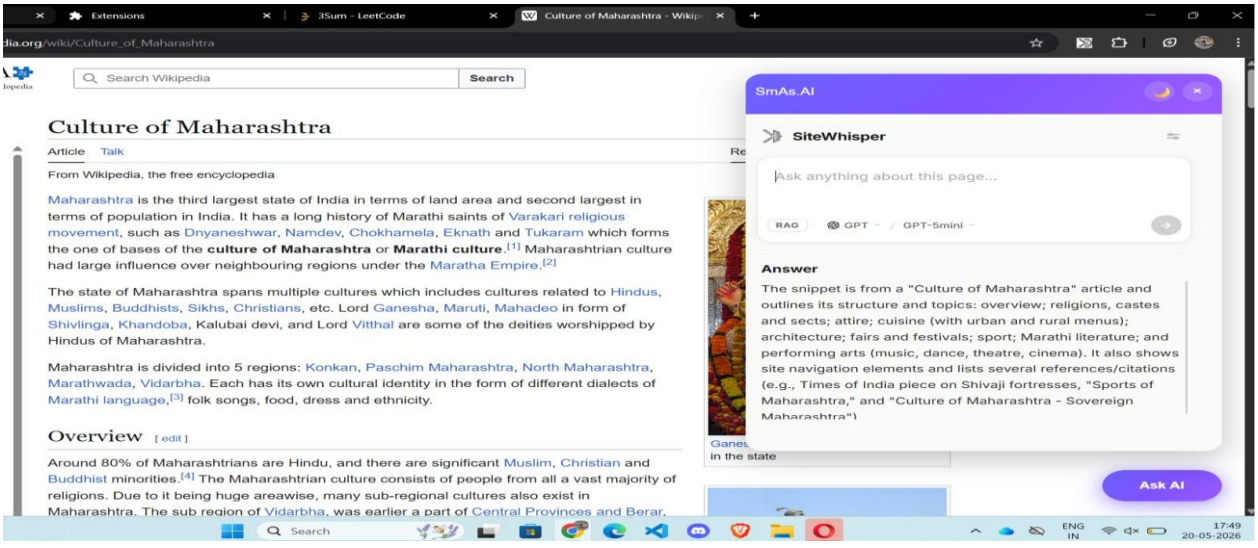
As shown in the system architecture diagram, a Conversation DB is used by the AI/ML Service. Its purpose is to retrieve conversation history and context, allowing the AI to provide more relevant, non-repetitive responses. This also prevented the model from succumbing to hallucination to an acceptable extent

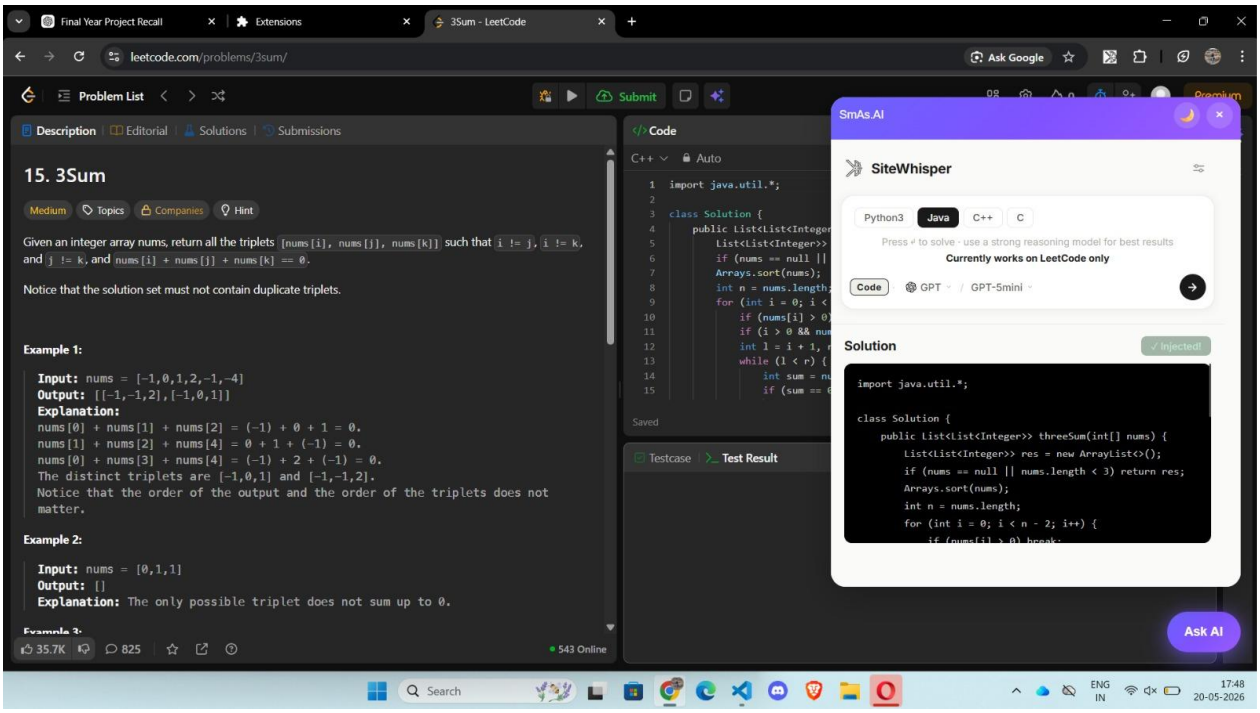


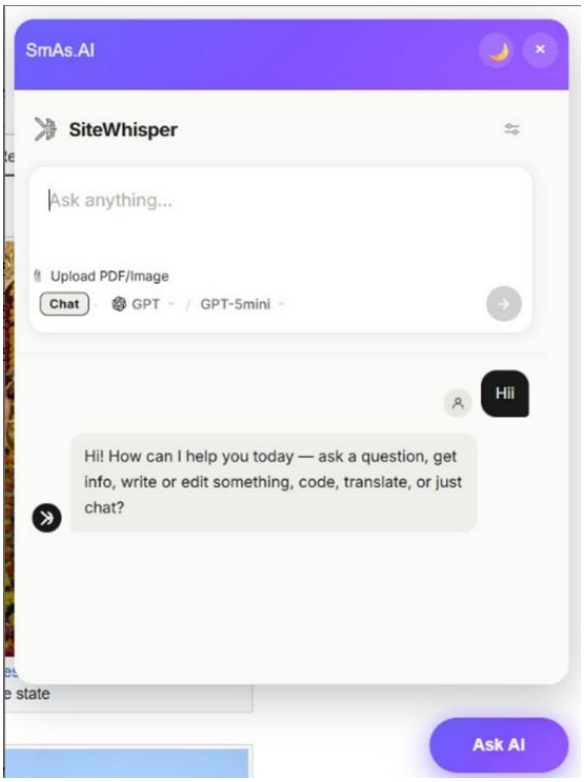
```python
load_dotenv()
api_key = os.getenv("OPENROUTER_API_KEY")

# --- Create OpenRouter Client ---
client = OpenAI(
    api_key=api_key,
    base_url="https://openrouter.ai/api/v1",
    default_headers={ "HTTP-Referer": "http://localhost", "X-Title": "SmartAssistAI" },
)

app = FastAPI()

# --- CORS ---
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# --- Data Models (Pydantic) ---
class ReactiveRequest(BaseModel):
    text: str
    action: str

# NEW: Data model for our proactive behavior data
class ProactiveRequest(BaseModel):
    elementType: str
    hoverTime: float
    scrollSpeed: float

# --- API Endpoint for RE-active Tools (Explain, Improve) ---
@app.post("/api/process-text")
def process_text(request: ReactiveRequest):
```



```javascript
// Wait for the page to be fully loaded
window.addEventListener('load', main);

// Use an async function so we can 'await' our HTML
async function main() {

    // 1. Fetch and Inject the Widget's HTML
    try {
        const widgetURL = chrome.runtime.getURL('widget.html');
        const response = await fetch(widgetURL);
        const widgetHTML = await response.text();
        document.body.insertAdjacentHTML('beforeend', widgetHTML);
    } catch (error) {
        console.error("SmartAssist: Failed to load widget HTML.", error);
        return;
    }

    // 2. Get references to all the elements
    const widgetContainer = document.getElementById('smart-assist-widget-container');
    const triggerButton = document.getElementById('widget-trigger');
    const widgetPanel = document.getElementById('widget-panel');
    const closeButton = document.getElementById('close-btn');
    const improveButton = document.getElementById('smartassist-btn-improve');
    const explainButton = document.getElementById('smartassist-btn-explain');

    // 3. Add Event Listeners

    // --- Draggable vs. Click Logic ---
    let hasDragged = false;
    let isDragging = false;
    let offsetX, offsetY;

    const onMouseDown = (e) => {
        // Check if the click is on a draggable part
        if (e.target === triggerButton || triggerButton.contains(e.target)) {
            isDragging = true;
            hasDragged = false; // Reset on every mousedown
```







VI. CONCLUSION

Summary

This project, "SmartAssist.AI," details the research and development of a universal, intelligent web assistant. The primary goal was to address the core inefficiency of existing digital tools, which operate on a reactive model (requiring the user to select text and request help manually). We introduced and developed a proactive system that anticipates user needs.

The project consists of a full-stack application: a Python (FastAPI) backend and a JavaScript (Chrome Extension) frontend. The system offers standard reactive AI tools, including text explanation and grammar correction, via the OpenAI API.

Conclusion

In conclusion, this project successfully demonstrates the feasibility and value of a proactive, behavior-driven AI assistant. The development of "SmartAssist.AI" as a functional, full-stack prototype was a success.

The central research hypothesis was validated: The implemented Decision Tree Classifier proved capable of predicting user intent with a quantifiable and significant degree of accuracy. This confirms that analyzing on-page behavioral data is a viable and effective method for creating a more intelligent, seamless, and efficient user experience.

The primary limitation of this research is the size and manual labelling of the training dataset. Future work should focus on building a more robust data collection pipeline and experimenting with more complex

ensemble models (like Random Forests) to enhance predictive accuracy further. This project provides a strong foundation for the future of truly user-anticipating web applications. A secondary limitation to this research is the nature of APIs and the overhead from the interaction between them during operation. This also caused external costs to arise due to usage of external API keys like OpenAI API keys for the backend of the assistant, future research in this field would benefit from using a self-built model and lesser reliance on such API keys.

REFERENCES

[1] M. S. Kalyon and Y. S. Akgul, "A two-phase smart code editor," in *2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, Ankara, Turkey, 2021, pp. 1–4.

[2] S. Subhash, P. N. Srivatsa, S. Siddesh, A. Ullas, and B. Santhosh, "Artificial intelligence-based voice assistant," in 2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4), London, U.K., 2020, pp. 593–596.

[3] R. Vannala, S. B. Swathi, and Y. Puranam, "AI chatbot for answering FAQs," in 2022 IEEE 2nd International Conference on Sustainable Energy and Future Electric Transportation (SeFeT), Hyderabad, India, 2022, pp. 1–5.

[4] M. Baez, F. Daniel, F. Casati, and B. Benatallah, "Chatbot integration in few patterns," IEEE Internet Computing, vol. 25, no. 3, pp. 52–59, May–Jun. 2021.

[5] M. A. K. Raiaan et al., "A review on large language models: Architectures, applications, taxonomies, open issues and challenges," IEEE Access, vol. 12, pp. 26839–26874, 2024.

[6] P. Haindl and G. Weinberger, "Students' experiences of using ChatGPT in an undergraduate programming course," IEEE Access, vol. 12, pp. 43519–43529, 2024.

[7] A. Anand, R. Subha, S. Rajan, N. Bharathi, and A. K. Srivastava, "An efficient, precise and user-friendly AI-based virtual assistant," in 2023 International Conference on the Confluence of Advancements in Robotics, Vision and Interdisciplinary Technology Management (IC-RVITM), Bangalore, India, 2023, pp. 1–5.

[8] "IRJET—Smart AI Assistant," International Research Journal of Engineering and Technology (IRJET), 2020.

[9] E. Adamopoulou and L. Moussiades, "Chatbots: History, technology, and applications," Machine Learning with Applications, vol. 2, Art. no. 100006, 2020.

[10] L. Benaddi, C. Ouaddi, A. Jakimi, and B. Ouchao, "A systematic review of chatbots: Classification, development, and their impact on tourism," IEEE Access, vol. 12, pp. 1–1, 2024, doi: 10.1109/ACCESS.2024.3408108.

[11] R. Schmidt, R. Alt, and A. Zimmermann, "Assistant platforms," *Electronic Markets*, vol. 33, Art. no. 59, 2023.