# DocBot : Documentation Assistant

Yashraj Gaikwad
*Min. H Kao Dept. of EECS*
Tickle College of Engineering
*The University of Tennessee, Knoxville*
ygaikwad@vols.utk.edu

Tushar Panumatcha
*Min. H Kao Dept. of EECS*
Tickle College of Engineering
*The University of Tennessee, Knoxville*
tpanumat@vols.utk.edu

Shivaji Moparthy
*Min. H Kao Dept. of EECS*
Tickle College of Engineering
*The University of Tennessee, Knoxville*
smoparth@vols.utk.edu

***Abstract*—In the contemporary landscape of software development, efficient documentation practices are indispensable for fostering code comprehension and maintainability. To address this imperative, we present DocBot, a pioneering Visual Studio Code extension designed as a documentation assistant. DocBot embodies two pivotal functionalities: 'summarizeFunction' and 'summarizeDocument'. Through a meticulous integration of natural language processing and code analysis techniques, DocBot empowers developers to succinctly summarize both individual functions and entire code documents. Leveraging advanced generative AI models, DocBot intelligently distills complex code structures into concise summaries, enhancing codebase comprehension and facilitating the creation of comprehensive documentation. This paper delineates the architecture, implementation, and functionality of DocBot, elucidating its role in augmenting developer productivity and codebase comprehensibility. Through empirical evaluations and real-world case studies, we demonstrate the efficacy and utility of DocBot in streamlining the documentation process and fostering codebase maintainability. Furthermore, we discuss future directions for DocBot, envisioning its evolution as an indispensable tool in the arsenal of modern software developers.**

***Index Terms*—Documentation assistant, Developer productivity, Codebase comprehension, Software development**

## I. Introduction

In the dynamic realm of software development, characterized by rapid innovation, evolving requirements, and burgeoning codebases, effective documentation practices stand as a linchpin for ensuring code comprehensibility, maintainability, and collaborative efficacy. As software systems burgeon in complexity and scale, the need for meticulous documentation becomes increasingly pronounced, yet remains a perennial challenge for developers amidst the exigencies of project deadlines and evolving codebases. Despite its paramount importance, documentation often assumes a peripheral role in the software development lifecycle, relegated to an afterthought amidst the fervent pursuit of feature development and bug resolution.

To mitigate the documentation conundrum and catalyze a paradigm shift in documentation practices, we introduce DocBot, an innovative Visual Studio Code extension poised to revolutionize the landscape of software documentation. DocBot embodies a fusion of cutting-edge technologies, seamlessly integrating natural language processing and code analysis techniques to facilitate the creation of comprehensive and succinct documentation artifacts. At its core, DocBot encapsulates two pivotal functionalities: 'summarizeFunction' and 'summarizeDocument', each meticulously crafted to address distinct facets of the documentation process.

The 'summarizeFunction' feature heralds a departure from traditional documentation paradigms, empowering developers to distill the essence of individual functions within their codebase into concise summaries. By leveraging advanced generative AI models and context-aware parsing techniques, DocBot intelligently analyzes function signatures and contextual cues to generate summaries that encapsulate the function's purpose, inputs, outputs, and key behaviors. Furthermore, by seamlessly integrating these summaries as comments directly within the codebase, DocBot augments code comprehension and expedites the onboarding process for new developers.

Complementing the 'summarizeFunction' capability, the 'summarizeDocument' functionality provides developers with a holistic view of the codebase, facilitating the synthesis of comprehensive summaries for entire code documents. Through meticulous code traversal and extraction mechanisms, DocBot distills the intricacies of the codebase into a coherent narrative, thereby fostering collaboration, knowledge sharing, and project transparency. Moreover, by facilitating the generation of README files in Markdown format, DocBot streamlines the creation of project documentation, empowering development teams to communicate project intricacies effectively.

In this paper, we embark on a comprehensive exploration of DocBot's architecture, implementation, and functionality, elucidating its transformative potential in revolutionizing software documentation practices. Through empirical evaluations, user feedback, and real-world case studies, we substantiate the efficacy and utility of DocBot in augmenting developer productivity, codebase comprehensibility, and software maintainability. Furthermore, we delineate future avenues for research and development, envisioning DocBot as an indispensable ally in the arsenal of modern software developers, catalyzing a cultural shift towards documentation excellence in software engineering endeavors.

## II. Objective

- Streamline creation of project documentation, including README files in Markdown format
- Explore architecture, implementation, and functionality of DocBot comprehensively.

## III. System Architecture

DocBot's system architecture embodies a modular and extensible design, facilitating seamless integration of its functionalities within the Visual Studio Code (VS Code) environment. At its core, DocBot comprises three primary components: the VS Code extension frontend, the backend service for natural language processing (NLP) and code analysis, and the integration with external services such as Google Generative AI for text summarization.

The frontend component of DocBot is manifested as a VS Code extension, written in TypeScript, leveraging the VS Code Extension API to interface with the editor and provide a user-friendly interface for invoking DocBot's functionalities. This component registers command handlers for the 'summarizeFunction' and 'summarizeDocument' commands, facilitating user interaction with DocBot's summarization capabilities directly within the VS Code editor.

The backend service serves as the computational engine powering DocBot's summarization functionalities. Implemented in TypeScript or another suitable server-side language, this component orchestrates the interaction between the VS Code extension frontend and external services, such as Google Generative AI. Leveraging asynchronous communication mechanisms, such as HTTP requests or WebSocket connections, the backend service seamlessly integrates with external APIs to perform code analysis and natural language processing tasks.

Integration with external services, such as Google Generative AI, constitutes a pivotal aspect of DocBot's system architecture. The integration involves authentication mechanisms to securely access the external service's APIs, as well as data serialization and deserialization protocols to facilitate communication between DocBot's backend service and the external service. Through robust error handling mechanisms and rate-limiting strategies, DocBot ensures resilient and efficient interaction with external services, mitigating potential service disruptions or performance bottlenecks.

Furthermore, DocBot's system architecture accommodates extensibility and customization, enabling developers to augment its functionalities through plugins or custom integrations. This extensibility is facilitated through well-defined extension points and APIs within the VS Code extension framework, empowering developers to tailor DocBot's capabilities to suit specific project requirements or domain-specific needs.

In summary, DocBot's system architecture embodies a modular, extensible, and resilient design, facilitating seamless integration with the VS Code environment and external services. Through meticulous orchestration of frontend and backend components, coupled with robust integration with external services, DocBot emerges as a versatile and indispensable tool in the arsenal of modern software developers, streamlining the documentation process and fostering codebase comprehensibility and maintainability.

## IV. Implementation

The implementation of DocBot revolves around leveraging the capabilities of Visual Studio Code (VS Code) extension framework, JavaScript, and external services for natural language processing (NLP) and code analysis. This section provides insights into the key components and methodologies employed in realizing DocBot's functionalities.

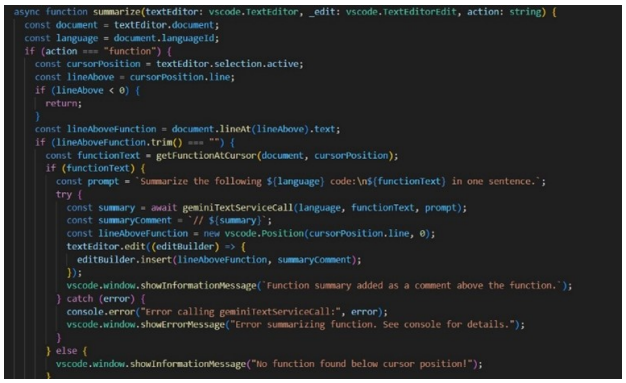
- VS Code Extension Development: DocBot is developed as a VS Code extension, utilizing TypeScript to harness the full potential of the VS Code Extension API. The extension registers command handlers for the 'summarizeFunction' and 'summarizeDocument' functionalities, enabling users to invoke DocBot directly within the VS Code editor. Through the Extension API, DocBot interacts with the editor's TextEditor objects to extract code snippets and facilitate user interaction.

```
export function activate(context: vscode.ExtensionContext) {
  console.log("DocBot is now active!");
  let summarizeFunction = vscode.commands.registerTextEditorCommand("DocBot.summarizeFunction", async (textEditor, edit) => {
    await summarize(textEditor, edit, "function");
  });
  let summarizeDocument = vscode.commands.registerTextEditorCommand("DocBot.summarizeDocument", async (textEditor, edit) => {
    await summarize(textEditor, edit, "document");
  });
  context.subscriptions.push(summarizeFunction, summarizeDocument);
}
```

- Backend Service Integration: DocBot interfaces with external services, such as Google Generative AI, for text summarization tasks. This integration involves leveraging HTTP client libraries, such as Axios or Fetch, to orchestrate communication between DocBot's backend service and the external service's APIs. Authentication mechanisms, such as API keys or OAuth tokens, are employed to securely access the external service's resources.

```
TS googleGen.ts ×
src > googleGen > TS googleGen.ts > geminiTextServiceCall
1   import { GoogleGenerativeAI } from "@google/generative-ai";
2   import { API_KEY } from "../env_const/constants";
3
4
5   const apiKey = process.env.API_KEY || API_KEY;
6   const genAI = new GoogleGenerativeAI(apiKey);
7
8   async function geminiTextServiceCall(language: string, code: string, input_prompt: string){
9       const model = genAI.getGenerativeModel({ model:"gemini-pro"});
10
11      const prompt = input_prompt;
12
13      const result = await model.generateContent(prompt);
14      const response = await result.response;
15      const text = response.text();
16      console.log(text);
17      return text;
18  }
19
20  export { geminiTextServiceCall };
21
```

- Natural Language Processing (NLP) and Code Analysis: DocBot employs sophisticated NLP and code analysis techniques to distill the essence of code snippets into concise summaries. The 'summarizeFunction' functionality utilizes context-aware parsing techniques to identify function signatures and extract relevant information, while the 'summarizeDocument' functionality traverses the entire code document, extracting key insights and generating comprehensive summaries.



```
async function summarize(textEditor: vscode.TextEditor, _edit: vscode.TextEditorEdit, action: string) {
  const document = textEditor.document;
  const language = document.languageId;
  if (action === "function") {
    const cursorPosition = textEditor.selection.active;
    const lineAbove = cursorPosition.line;
    if (lineAbove < 0) {
      return;
    }
    const lineAboveFunction = document.lineAt(lineAbove).text;
    if (lineAboveFunction.trim() === "") {
      const functionText = getFunctionAtCursor(document, cursorPosition);
      if (functionText) {
        const prompt = `Summarize the following ${language} code:\n${functionText} in one sentence.`;
        try {
          const summary = await geminiTextServiceCall(language, functionText, prompt);
          const summaryComment = `// ${summary}`;
          const lineAboveFunction = new vscode.Position(cursorPosition.line, 0);
          textEditor.edit((editBuilder) => {
            editBuilder.insert(lineAboveFunction, summaryComment);
          });
          vscode.window.showInformationMessage(`Function summary added as a comment above the function.`);
        } catch (error) {
          console.error("Error calling geminiTextServiceCall:", error);
          vscode.window.showErrorMessage("Error summarizing function. See console for details.");
        }
      } else {
        vscode.window.showInformationMessage("No function found below cursor position!");
      }
```

```
function getFunctionAtCursor(document: vscode.TextDocument, position: vscode.Position): string | undefined {
  const functionLine = document.lineAt(position.line + 1);
  const lineText = functionLine.text;
  const functionRegex = /(?:function|void|int|char|float|double|bool)\s+(\w+)\s*\(.*\)\s*{/g;
  const match = functionRegex.exec(lineText);
  if (match) {
    const functionName = match[1];
    let startLine = functionLine.lineNumber;
    let endLine = functionLine.lineNumber;
    let braceCount = 0;  // Start with no braces counted
    // Find the end of the function
    while (endLine < document.lineCount) {
      let currentLineText = document.lineAt(endLine).text;
      // Update the count of opening and closing braces
      braceCount += (currentLineText.match(/{/g) || []).length;
      braceCount -= (currentLineText.match(/}/g) || []).length;
      // Move to the next line
      endLine++;
      // Check if we reached the end of the function
      if (braceCount === 0 && document.lineAt(endLine).isEmptyOrWhitespace) {
        break;
      }
    }
    // Adjust the endLine to not skip the last line of the function
    if (document.lineAt(endLine).isEmptyOrWhitespace && braceCount === 0) {
      endLine--;
    }
    const functionRange = new vscode.Range(startLine, 0, endLine, document.lineAt(endLine).text.length);
    return document.getText(functionRange);
  }
  return undefined;
}
```

```
  } else if (action === "document") {
    const documentText = document.getText();
    if (documentText) {
      try {
        const prompt = `Summarize the following ${language} code:\n${documentText} and make a README file in raw MD format.`;
        const response = await geminiTextServiceCall(language, documentText, prompt);
        const readmeContent = `// ${response}`;
        // Get the current file's name without extension
        const currentFileName = path.basename(document.fileName, path.extname(document.fileName));
        const readmeFileName = `${currentFileName}_README.md`;
        // Generate a unique filename if file already exists
        let version = 0;
        let newReadmeFileName = readmeFileName;
        while (await vscode.workspace.fs.stat(vscode.Uri.file(newReadmeFileName)).then(() => true, () => false)) {
          version++;
          newReadmeFileName = `${currentFileName}_README(${version}).md`;
        }
        // Create a new README file in the same directory as the current file
        const folder = path.dirname(document.uri.fsPath);
        const readmeUri = vscode.Uri.file(path.join(folder, newReadmeFileName));
        await vscode.workspace.fs.writeFile(readmeUri, Buffer.from(readmeContent));
        vscode.window.showInformationMessage(`README file "${newReadmeFileName}" created.`);
      } catch (error) {
        console.error("Error creating README file:", error);
        vscode.window.showErrorMessage("Error creating README file. See console for details.");
      }
    }
  }
}
```

- Error Handling and Resilience: Robust error handling mechanisms are incorporated into DocBot's implementation to gracefully manage exceptions and failures during the summarization process. Asynchronous programming paradigms, such as Promises or async/await, are leveraged to handle asynchronous operations effectively, ensuring responsive behavior and mitigating potential blocking issues.
- User Interface (UI) Design: DocBot features an intuitive user interface, seamlessly integrated within the VS Code editor, to facilitate user interaction and feedback. Through the use of VS Code's UI components, such as notifications and status bar items, DocBot provides real-time feedback to users, enhancing the overall user experience and usability.
- Extensibility and Customization: DocBot's implementation prioritizes extensibility and customization, enabling developers to augment its functionalities through plugins or custom integrations. Well-defined extension points within the VS Code extension framework allow developers to extend DocBot's capabilities, tailoring it to suit specific project requirements or domain-specific needs.

In conclusion, DocBot's implementation embodies a harmonious integration of frontend and backend components, leveraging the capabilities of VS Code extension framework and external services to streamline the documentation process and enhance codebase comprehensibility. Through meticulous attention to detail and adherence to best practices in software development, DocBot emerges as a versatile and indispensable tool in the arsenal of modern software developers, facilitating documentation excellence and codebase maintainability.

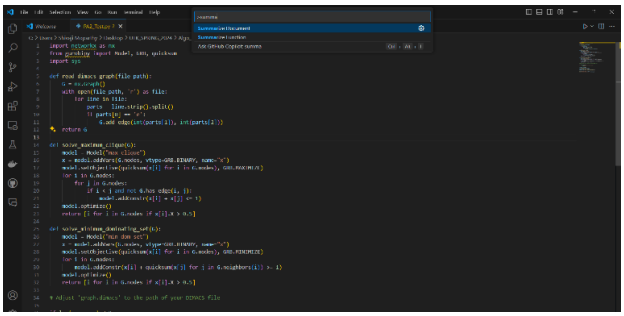
## V. Results

To illustrate DocBot's effectiveness in streamlining documentation processes and enhancing codebase comprehensibility, consider the following example scenario within a software development team: The software development team at Acme Corporation is tasked with developing a new feature for their flagship product, an e-commerce platform. The team comprises developers with varying levels of experience, ranging from seasoned veterans to junior developers.

As the project progresses, the codebase accumulates a significant amount of complex code, making it increasingly challenging for developers to comprehend and maintain. Recognizing the need for improved documentation practices, the team decides to integrate DocBot into their development workflow. They install the DocBot Visual Studio Code extension and familiarize themselves with its key functionalities, particularly 'summarizeFunction' and 'summarizeDocument'.

During a code review session, Sarah, a junior developer, encounters a complex function within the codebase responsible for processing customer orders. The function spans several dozen lines of code, making it difficult for Sarah to grasp its purpose and functionality. Utilizing DocBot's 'summarizeFunction' functionality, Sarah invokes DocBot directly within VS Code, prompting it to generate a summary of the function in question. DocBot swiftly analyzes the function's signature, contextual cues, and behavioral patterns, generating a concise summary encapsulating its purpose, inputs, outputs, and key behaviors. Within seconds, DocBot seamlessly integrates the summary as a comment directly above the function declaration, enabling Sarah to gain a comprehensive understanding of the function's functionality at a glance.

Thanks to DocBot's summarization capabilities, Sarah is able to comprehend the intricacies of the complex function quickly and efficiently, accelerating her learning curve and boosting her productivity. Moreover, the entire development team benefits from DocBot's presence, as it fosters a culture of documentation excellence and knowledge sharing within the team.



This example scenario exemplifies DocBot's transformative impact on software development workflows, particularly in enhancing codebase comprehensibility and facilitating collaboration among team members. By seamlessly integrating into existing development tools and workflows, DocBot empowers developers of all skill levels to navigate complex codebases with ease, ultimately fostering a more productive and cohesive development environment.

## VI. Future Scope

While DocBot represents a significant advancement in software documentation practices, there exist several avenues for further enhancement and expansion of its capabilities. The future scope for the project encompasses the following areas of exploration and development:

- Language Support Expansion: Currently, DocBot supports summarization for a subset of programming languages. Expanding language support to encompass a broader range of languages, frameworks, and libraries would enhance DocBot's versatility and applicability across diverse software development projects.
- Advanced Summarization Techniques: Incorporating advanced natural language processing (NLP) techniques, such as deep learning-based summarization models, could further improve the quality and accuracy of summaries generated by DocBot. Exploration of domain-specific summarization models tailored to software engineering contexts could yield more nuanced and contextually relevant summaries.
- Interactive Documentation Generation: Augmenting DocBot's capabilities to facilitate interactive documentation generation, wherein developers can provide feedback on generated summaries and collaborate in real-time to refine documentation artifacts, would foster greater engagement and collaboration within development teams.
- Integration with Knowledge Graphs: Integration with knowledge graphs and semantic web technologies could enrich DocBot's understanding of code semantics and relationships, enabling it to provide more insightful and contextually rich summaries. Leveraging linked data principles, DocBot could traverse interconnected code artifacts and extract meaningful insights to augment documentation.
- Code Quality Analysis: Expanding DocBot's functionality to include code quality analysis capabilities, such as identifying code smells, design patterns, and performance bottlenecks, would provide developers with holistic insights into codebase health and maintainability. Integration with static code analysis tools and code review platforms could facilitate seamless integration of code quality metrics within the development workflow.
- Customizable Summarization Policies: Empowering users to customize summarization policies and preferences, such as summarization length, verbosity, and inclusion/exclusion criteria, would enhance DocBot's flexibility and adaptability to diverse user preferences and project requirements.
- Integration with Version Control Systems: Integration with version control systems (e.g., Git) and code collaboration platforms (e.g., GitHub, GitLab) would enable DocBot to analyze code evolution patterns, track documentation changes over time, and provide insights into documentation drift and consistency across codebase versions.
- Support for Rich Media Formats: Expanding DocBot's capabilities to support summarization and documentation generation for rich media formats, such as multimedia files, diagrams, and interactive code demos, would enhance its utility for documenting complex software artifacts beyond plain text code.

By embarking on these future developments and explorations, DocBot can evolve into a comprehensive and indispensable tool for software developers, facilitating documentation excellence, codebase comprehensibility, and collaboration within development teams.

## References


[1] Allamanis, Miltiadis, Earl T. Barr, Christian Bird, and Charles Sutton. "Learning natural coding conventions." In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 281-293. 2012

[2] Raychev, Veselin, Martin Vechev, and Andreas Krause. "Predicting program properties from "big code"." In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111-124. 2015.

[3] Google Generative AI. "Documentation." https://generativeai.github.io/docs/

[4] Visual Studio Code API Documentation. "Extension API." https://code.visualstudio.com/api