CISC 322/326 - Software Architecture
Assignment #1: Report

Conceptual Architecture of Void Editor

Team Potential Guacamole
Karim Sponheimer - 21ktjs@queensu.ca
Kasia Zaski - 21kjdz@queensu.ca
Zachary Stephens - 20zs46@queensu.ca
Zurui Wang - 21zw105@queensu.ca

Abstract:

This report analyzes Void, an open-source, AI-powered Integrated Development Environment (IDE) forked from Visual Studio Code, and applies its architectural principles in the creation of our project website.Void extends VS Code's modular and service oriented architecture by incorporating artificial intelligence capabilities while keeping emphasis on privacy, customization, and community collaboration. It builds upon VS Code's microkernel and plugin-based foundations. AI models can be hosted locally or integrated through external providers using secure adapters. By retaining compatibility with VS Code's ecosystem, Void offers users a familiar yet enhanced development experience that unites the flexibility of open-source design with modern AI assistance.

The goal of the project was to study Void's conceptual architecture and apply its design principles in a practical, web based context. Our team developed a simple but structured website using Void as the primary development environment, designed to reflect the layered organization and modularity seen in complex software systems. While not functionally complex, our website was intentionally constructed to demonstrate clear separation of concerns, maintainability and scalability, key traits that underpin Void's architecture.

Each section of the site functions as an independent, self contained component that communicates through well defined boundaries, mirroring how Void separates its Renderer, Main, Extension Host separation, which forms the backbone of its performance and extensibility. The Renderer handles interface display and user interactions, the Main Process governs system-level access and inter-process communication, and the Extension Host allows modular plugin integration, allowing for powerful customization without compromising stability.

This layered structure is further extended by Void's AI integration framework, which connects to large language model (LLM) providers through Provider Adapters. These adapters facilitate secure, configurable communication between the IDE and AI systems, making sure that developers keep control over their data while benefiting from useful code suggesting, contextual edits, and natural language interactions.

Our research has found that this modular approach enables strong adaptability and scalability, even though it introduces performance overhead due to the reliance on Electron and multiple communication layers. Void's design shows how modularity and extensibility support not only system evolution but also community driven innovation. Applying these principles to our website improved clarity, maintainability, and separation of concerns, showing that IDE design concepts can translate effectively to frontend web design.

We also observed the limitations of such architectures, including potential performance constraints and the complexity of managing multiple subsystems.

The deliverables for this report include detailed architecture diagrams, use case scenarios, a data dictionary, and references. The use cases outline key workflows, demonstrating how each component interacts within the website and highlighting parallels with Void's modular design. The data dictionary provides concise definitions for essential components including LLM (Language models), IPC (Inter-process communication), and Provider Adapter, and Fast/Slow Apply modes, ensuring clarity for future developers or researchers studying AI-integrated IDE systems.

Collaboration with an AI teammate, GPT-5, further supported our work by assisting report structure, terminology alignment, and summarizing relationships between Void, VS Code, and Electron. While the AI supported planning and documentation, all technical validation, architecture design, and implementation decisions were performed by the human team, highlighting the effectiveness of AI as a guided collaborator rather than a content generator.

Beyond the immediate project outcomes, our work illustrates broader lessons for software engineering and web development. Taking design principles from a complex IDE to a smaller web project shows that modularity, separation of concerns, and extensibility are valuable in any context, no matter the project's scale. The project also highlights responsible AI use, and that with human oversight, developers can harness AI to improve efficiency and insights while preserving control, privacy, and system stability.

Overall, the project demonstrates how modular architecture, disciplined collaboration and responsible AI integration can work together to produce a robust, extensible and comprehensive software design.

Introduction & Overview:

Visual Studio Code was developed at a pivotal time in Microsoft's history, when the company hadn't found their footing in the software industry. At a time of strong competition from established giants, The company looked to redefine their role, and create a platform that could keep up with the diverse and changing needs of developers. Rather than competing directly, Microsoft set out to build a tool that was fast, adaptive and scalable.

The project began in 2011 under the direction of software engineer Eric Gamma, who initiated the development of a lightweight web-based code editor called Monaco. This early framework became the foundation for what later became VS Code. When VS Code was finally released 2015, it quickly gained popularity due to its performance, simplicity and wide range of community-built extensions. Today, it still stands as one of the most widely used code editors worldwide, and has continued to adapt to the needs of developers across the world.

Built upon this foundation, Void is a modern reimaging of VS Code architecture with the integration of artificial intelligence. It is an AI-powered code editor forked from VS code, that allows developers to use the power of large language models while maintaining complete control over their data. Void prioritizes privacy, customization and community involvement without sacrificing the AI assistance that so many developers rely on. Its open architecture allows

developers to view, modify and contribute to the system's prompts and features. This open and collaborative approach combines the technical robustness of VS Code with the growing model of AI-assisted programming.

Void also stands out for its seamless compatibility with the VS Code ecosystem. Users can seamlessly transfer all their existing configurations on VS Code, such as their settings, themes, keybinds in one click. In addition, Void offers more customization through access to community-made features beyond what closed-source alternatives offers, and local model hosting. This makes it not only practical, but also an appealing choice to developers that are looking for greater control over their data and tools. Because of these qualities, our team chose to work with Void, as it aligns with the objectives of the course. These objectives include understanding the conceptual architecture of complex software systems and exploring how design decisions support scalability, collaboration and innovation. Void's nature provides a rich case study for how emerging technologies reshape software development practices.

Our group used Void editor as the primary development environment to create a simple, modular website for our project. The goal of the project was to apply architectural and design principles within a web-based context. The site consists of clearly defined sections, where each functions as an independent yet unified component. The site was designed to be modular and maintainable, with each section acting as a contained component that reflects Void's own layered approach. For doing so, our implementation demonstrates how principles from IDE architecture, such as separation of concerns and extensibility, can be applied even in a web context. This design means that the site stays adaptable for future updates, while maintaining a structured, browser friendly site that communicates both technical and architectural insights.

The architecture of an Integrated Development Environment (IDE) plays a critical role in determining how efficiently developers can build, test, and refine software. A well structured IDE architecture, such as Void's, separates the user interface, core logic, and system-level operations. This design improves maintainability, scalability and performance, enabling the seamless additions of new features. An example being AI-assisted or real-time collaborations without disrupting existing systems. Our website applies these same principles by dividing the layout into clearly defined sections, where each one operates as a self contained component. For example, the navigation buttons in the header interact with the page's content sections without requiring changes to the underlying layout or other sections, demonstrating separation of concerns in a web context. Even though the website itself does not implement functional logic, this structure allows readers to interact with content in an organized, predictable way, mirroring the separation of concerns found in modern IDEs. IDE architecture matters because it shapes not only how software is written, but how smoothly innovation, usability, and collaboration can occur within a system.

The report is structured to provide a clear overview of our team, Potential Guacamole website and its development using Void. The report will begin with an Architecture Overview, serving as the core of the report, presenting the conceptual structure of our website, including its main components. This contains the header, content sections and other major parts. The following section, External Interfaces and Communication, goes over how users interact with the website through buttons and other interactive elements, highlighting the communication between different components. The next section is the Use Cases, which illustrates key interactions with

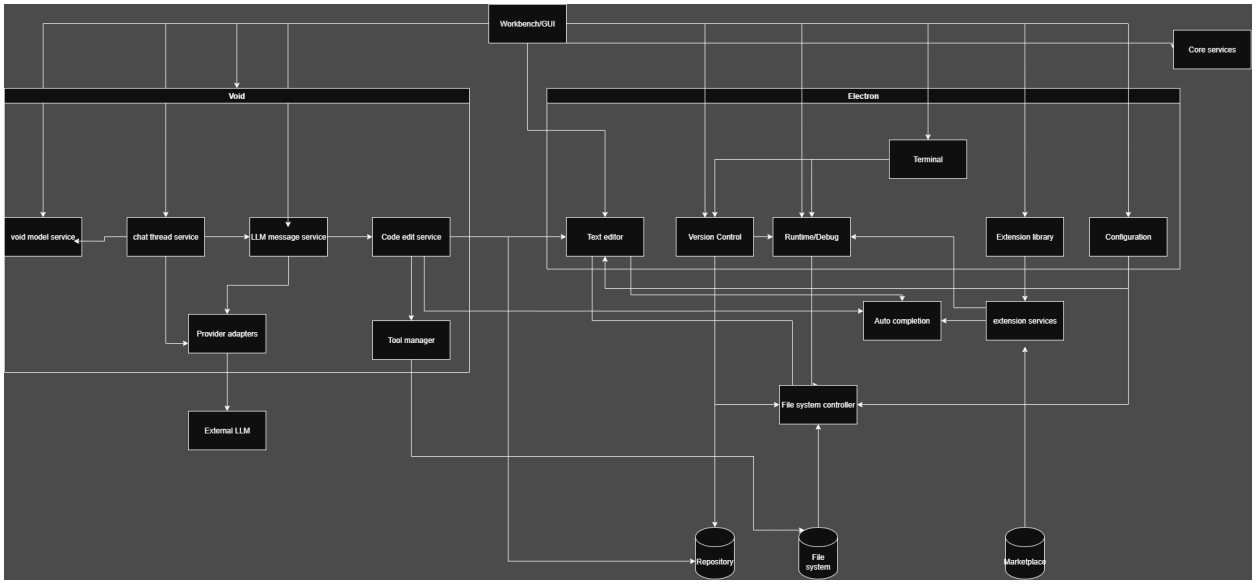
the system, including workflows such as futile editing and AI-assisted code changes. Sequence diagrams accompany these examples to show control and data flow, demonstrating how the website separates interface, core logic, and system access while using modern AI-assisted functionality. The Data dictionary, provides a reference for the website's key elements and properties, such as LLMs (Large Language Models), IPC(Inter-Process Communication), DiffZone, Provider Adapter, and the Fast Apply and Slow Apply modes for AI edits. The final sections summarize the key findings from the project, reflecting on the benefits and challenges of using Void for AI-assisted development. It highlights insights into system design, team collaboration, and software architecture. This organization allows readers to first understand the system's structure, then its interactions, followed by the concrete examples of user workflows, and finally a reference for key terms and components. This offers a comprehensive perspective suitable for those seeking to understand our website.

Void uses a layered, service oriented architecture after Visual Studio Code's microkernel design. The system separates the renderer, core, and main process, which allows clear boundaries between interface rendering, logic handling, and system-level operations. This modular design is strengthened by its plugin based extension model, where independent services interact through APIs and inter-process communications. The structure supports maintainability and scalability, enabling developers to extend Void's capabilities with the least amount of coupling between components. By mirroring this organization, where each section summarizes a self contained functional area, reflecting the same principles of separation of concerns and modular design that Void itself employs. These design choices show how modern IDE architectures can combine traditional extensibility with the latest AI collaboration, yielding a robust, and privacy oriented environment for developer change.

Architecture:

1. Overall Structure

Void is built on VScode so inherits much of its architecture. VScode has a layered style with Electron being the layer for user interaction, the core services layer being responsible for managing the application such as updates and memory allocation. Void adds a layer, in this diagram named Void, that connects to the Electron layer and handles all the AI integration for the system. This structure allows for relatively easy addition to VScode as there is no need to restructure VScode while adding the new features.



Subsystems

The Workbench UI subsystem is what controls the window and hosts all the various panels. It handles launching, resizing, and closing of the application. It also handles capturing input from the user and sending that input to other subsystems.

The Core Services subsystem handles the background running of the application. Things such as launching, exiting, allocating resources and updates. This is not apparent to the user, but it works as backend support for the function of the application.

The Terminal subsystem is responsible for querying the system terminal and redirecting the output into the terminal panel of VScode or Void. This allows for testing and running code without leaving the app. It is able to have multiple terminals active of various types, like git bash. It also hosts the console output, which requires it to connect with the runtime editor, and its connection with version control ensures it runs only the most up-to-date version of whatever code it is running.

The Text editor is where the code is stored and edited while it is being written or updated. It highlights keywords in the code with different keywords and colours changing with the code language, which is either chosen by the user or auto-detected. It connects to autocomplete in order to allow for it to work. It also connects with the file system manager, to allow saving and opening files into the editor. Chat edit service connects to the text editor allowing the AI systems to edit or add code directly into the file that's being worked on.

Version control: VScode provides integration for Git and Github to allow for version control and collaboration across multiple people. Void inherits that integration. The version control subsystem allows for cloning repositories directly from Github, as well as the push, pull, fetch, and commit from Git. This allows for easy collaboration with others over Github without leaving the app, as well as easy opening of files from Github.

Runtime/Debug: Works with extensions to run, compile and debug files. It has limited language support natively, but that support can be expanded through extensions. In addition to the standard run and debug commands it allows for custom running configurations through use of a .json file. Often outputs to the Terminal subsystem, either to console or to system terminal.

Extension library: Handles the download of extensions as well as the connection to the extension marketplace. Through that connection it allows users to search for, browse, download, update, and uninstall extensions.

Extension Services: The subsystem provides a way to use extensions to affect the application. Common extensions are different programming languages and different themes than are provided in the base application. The effect of this subsystem is that VScode and Void become much more flexible and powerful IDEs.

Configuration: The configuration subsystem is in charge of opening and closing the panels that other subsystems use for user interaction. It also is responsible for updating settings for all the other subsystems and in managing the use of multiple tabs.

Extension services: The Visual Studio provides a bunch of APIs for extension development, allows extensions to do many things, for instance, language model support, add features, add commands, etc. In addition, Visual Studio also uses the process sandbox to isolate different extensions host and rendering process for balance between performance and security. The extension services were designed as implicit invocation so extensions can adapt to different scenarios and reduce performance cost.

File system manager: Void inherits the abstraction ability of the file system from Visual Studio. To improve the compatibility of Visual Studio in different operating systems or environments, all file-related operations will use a group of API provided by Visual Studio and it has different implementations for different schemes, like local storage, remote file system or cloud storage.

Autocomplete: This subsystem is inherited from VScode but was reconfigured to work in Void. Autocomplete uses either intellisence or the configured AI provider in order to provide suggestions for what the next part of the code should be. In addition to this feature, it also allows users to scroll through all possible keywords that start with the substring that the user input. This allows the user to see all the functions in a package and see the possible formats for a specific function.

Code edit service: Interfaces with the Text Editor and Autocomplete to provide AI created code when requested by the User. Functions either based on a command in the Text Editor or by providing possible next steps to the Autocomplete. In both situations it takes the context of the code currently in the file and pairs it with a goal and sends it to the LLM Messenger Service to be sent to the LLM.

Tool manager: Manages the installation and usage of LLMs. Sets which task is sent to which connected LLM to better process requests.

LLM message service: Creates prompts for the connected LLM based on the goal and code context sent by the Code Edit Service. The prompts must be tailored to each version of each AI in order to provide the optimal response.

Chat thread service: Manages the usage of multiple LLMs at once, ensuring the prompts go to the correct destination and that memory is properly allocated between the different AI.

Void model service: Contains and manages information about the LLM that are connected to Void. Makes that information available to other subsystems to allow usage of the models.

Provider adapter: Serves to connect to external LLMs and send and receive information in whatever format they require. It then reconfigures the response from the LLM into a format that the rest of the system can understand.

Architecture Style

The architecture style of Void is primarily layered. This allows for easy upgradability and for more control over dataflow between layers, which is important due to Void being made as a way to use AI to code while protecting user privacy. In addition the layered style also allows for a fairly easy to understand design, an important requirement when working on open source software so that all developers understand the design.

In addition to the layered style, there is a clear implied invocation element to the design. Particularly around how extensions are handled. This element likely includes the Text Editor, the Autocomplete, and the Extension Service with in use extensions acting as plug-ins to the main process of the Text Editor and Autocomplete.

External Interfaces:

- **GUI**: Editor panes, chat, approval prompts.

  The user interface of Void Editor primarily inherits from the frontend shell of Visual Studio Code + Electron, where most panels are just exactly the same. We have the menu bar (in Windows), sidebar (shows files, search panel, extensions, etc.), status bar, text editor and some other optional panels as well.

  The major difference comes up in the `Chat` panel where most features and functionality of Void Editor are integrated into that panel. First, the chat function is quite similar, both void editor or copilot chat can use different models, but void editor chat panel provides you some code-based suggestion buttons while copilot chat is only composed from the chat history section and a text box input. And, there are also some other features like inline quick edit, visualized diff/checkpoint integrated into the editor area. The checkpoint also enables the ability to rollback from AI changes.

- **APIs**: Extension APIs, LLM provider APIs (REST/streaming).

The extension APIs of Visual Studio Code are exposed within the namespace of vscode, the functionality can be categorized as following list:

- Authentication
- Chat
- Commands
- Editor/Document
- Environment
- Localization
- Source Control
- Tasks/Tests/Debug
- Window
- Workspace

Both VSCode and Void provide the ability to use a Large Language Model as assistant, but the implementation could be a little bit different, where Void absolutely provides more features, functionality, and freedom. In this case, Void allows you to add your own LLM provider, even the model deployed locally. Additionally, Void promised that the data was sent directly to the cloud or local LLM endpoint and no data will be reserved.

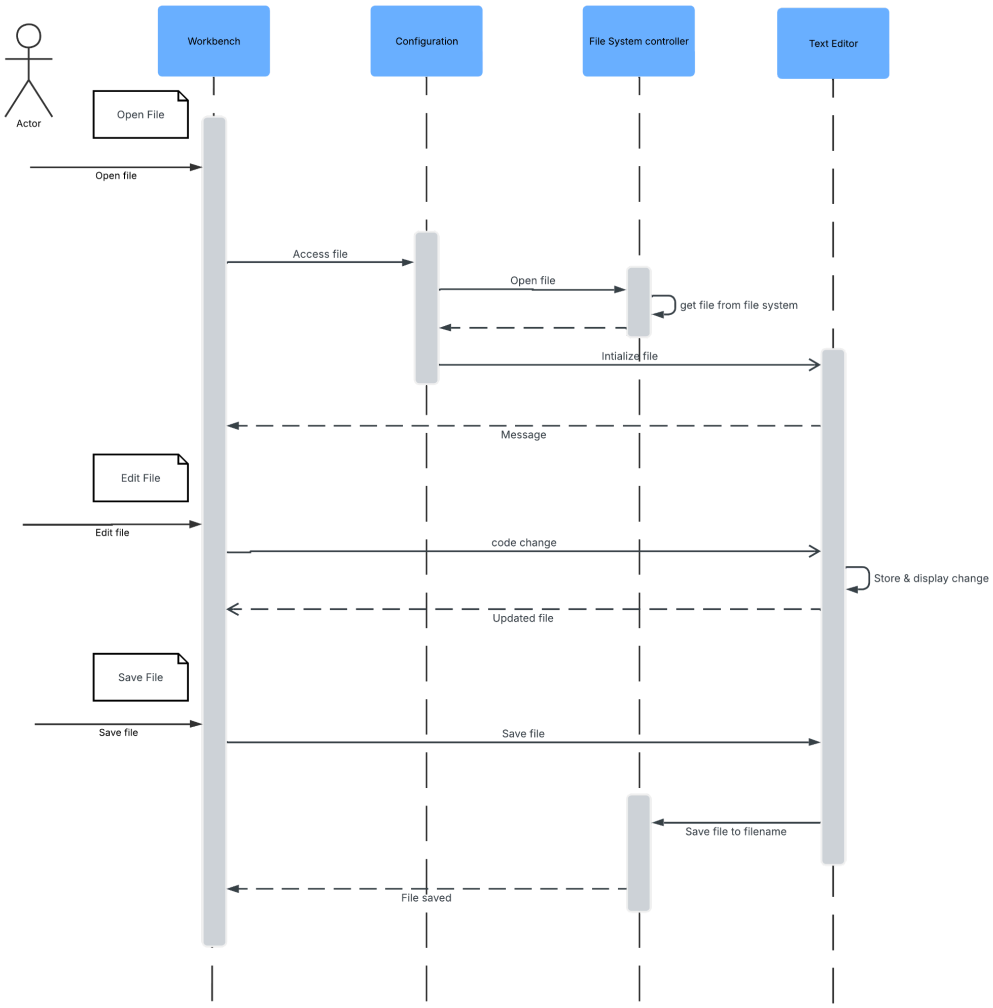
- **OS Services**: File System, Clipboard, Process exec.
  - File System: The vscode provides an abstraction to either local file system or remote. The vscode use URI + workspace for a unified access and access to different kind of source is provided by different FileSystemProvider, therefore, SSH/WSL/Cloud/Containers/etc. can be read or write through the same API.
  - Terminal: vscode can create a terminal which redirects the standard input/output to integrated panel for userspace execution.
  - Clipboard, Network, etc.: Mostly provided by the OS.

Use Cases:

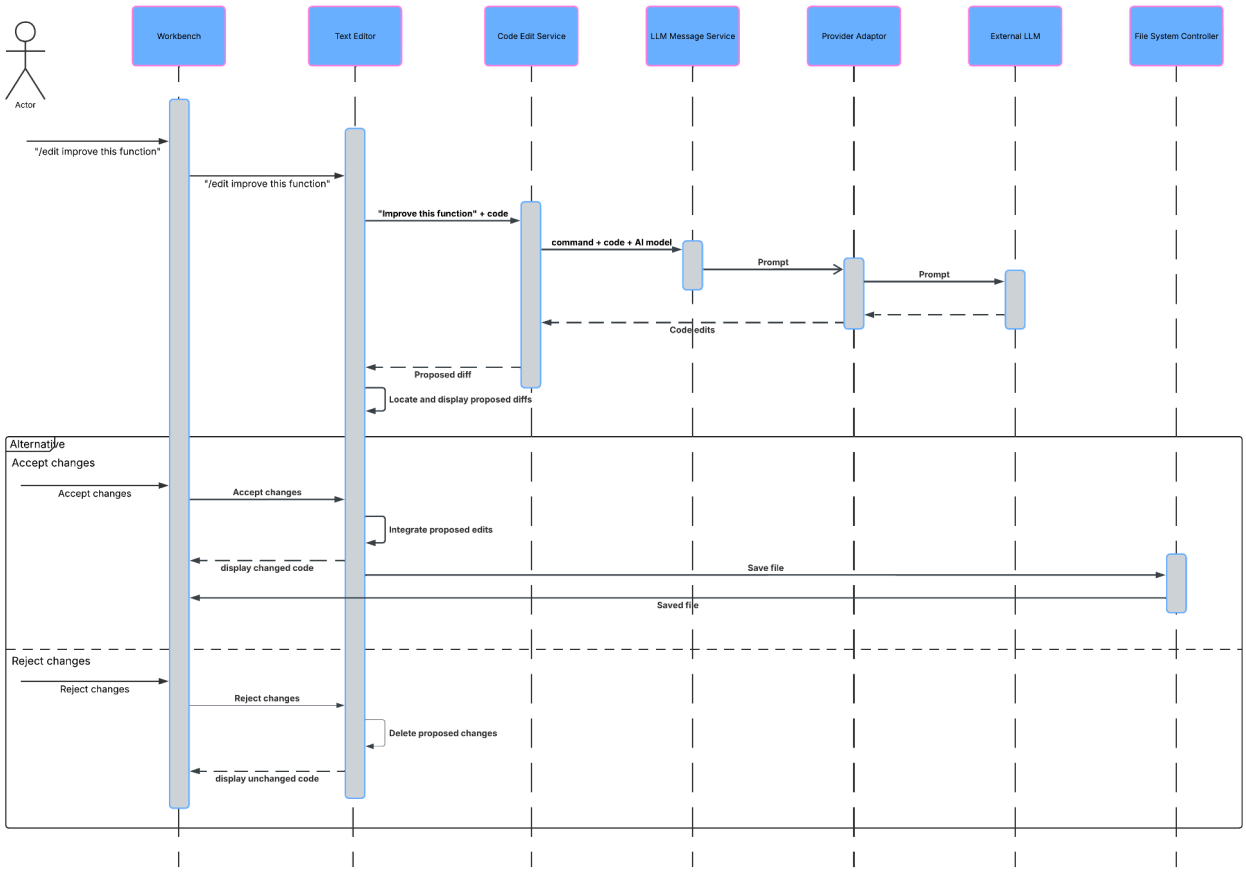
The first use case explains the process of opening, editing, and saving a file. When the user selects a file from the workspace, the Workbench sends a request through the Configuration subsystem to the file system controller to access that file. The File System Controller gets the file from the computer's file system and sends it back to the Configuration subsystem. The Configuration subsystem reads the file and sends the contents to the Text Editor, which updates the text buffer and completes the operation. The file then appears in the editor, and the user can make changes in real time.

When the user edits the file, the Text Editor maintains a buffer of unsaved changes that also supports undo and redo actions. Once the user chooses to save, the Workbench tells the Text Editor to send the updated data to the File System Controller. The File System Controller then writes the changes to disk and sends a success message back. The Workbench updates the

interface to show that the file is saved. This interaction shows how Void separates user interface, logic, and system access through a clear layered structure that it inherits from VS Code and extends in its own design.



The second use case covers Void's AI editing feature using the chat command called /edit. When the user types a message such as "/edit improve this function," the Workbench captures the request and sends it to the Text Editor. The Text Editor then sends the request and the current code to the Code Edit Service, which collects the code context and the command and decides which AI to use. The Code Edit Service sends this information to the LLM Message Service, which combines the code context and command into a prompt for the AI. The prompt is then routed to the Provider Adapter connected to external language models like OpenAI or Anthropic. The Provider Adaptor communicates with the chosen model over an API and receives a stream of suggested code edits. These are passed back to the Code Edit Service and sent to the Text Editor as a proposed diff. The Text Editor shows the new code alongside the old version so the user can review or reject it. If accepted, the Text Editor updates the text buffer with the approved edits and triggers a save operation. If the user cancels, the buffer stays unchanged.



Together, these two use cases show how Void blends traditional IDE features with modern AI assistance. The first example demonstrates its reliable file management and structured layering between Renderer, Core, and Main. The second highlights its extensible architecture that allows live integration with AI providers while keeping user control and privacy. Both illustrate how Void's design builds on the strengths of Visual Studio Code and adapts them for an AI-centered development environment.

Data Dictionary:

- **LLM**: Large Language Model. A class of language models composed of artificial neural networks with a large number of parameters are trained on a large amount of unlabeled text using self-supervised learning or semi-supervised learning.
- **IDE:** Integrated development environment. An application software that assists software developers in developing software. An IDE typically includes a programming language editor, an automatic build tool, and usually a debugger as well; some also incorporate a compiler/interpreter.
- **Terminal:** The terminal is a shell that allows users to input and receive output from the operating system; it's a bridge between the user and the operating system kernel. The

terminal used by Void is typically a pseudo-teletypewriter for simulating traditional teletypewriters.
- **IPC**: Inter-Process Communication between processes. Rely on the operating system cooperation, in Void we use IPC to communicate in different subprocesses like the renderer and host process.
- **DiffZone**: Editor decoration for previewing edits. It shows the preview of the changes made by LLM models that will be applied to the current code.
- **Provider Adapter**: Main-process module bridging Void to external LLM API.

- **Fast Apply**: Patch insertion mode for AI edits.

- **Slow Apply**: Full-file rewrite mode.

Conclusions:

Overall, our conceptual architecture for Void explains how its key subsystems work together to deliver an AI-assisted IDE while keeping responsibilities cleanly separated. The Workbench handles user interaction and routes commands. The Configuration subsystem coordinates panel state, settings, and multi-tab behavior while mediating requests between the UI and the rest of the system. The File System Controller performs reads and writes, which keeps operating system access out of the UI path. The Text Editor maintains the active document and the in-memory buffer that supports editing, undo, and redo. Together, these parts reflect a layered approach that improves maintainability and testability.

Void also uses an implicit invocation style for extensibility and AI features. The Code Edit Service reacts to edit intents such as /edit, gathers context from the Text Editor, and hands off to the LLM Message Service to construct prompts. The Tool Manager selects a model based on task and policy, and the Provider Adapter communicates with the external or local LLM, then normalizes the response back into a proposed diff. These interactions are event-driven rather than tightly coupled, which allows streaming responses, inline previews, and safe approval flows inside the Text Editor.

This combination of layering and implicit invocation supports the project goals of extensibility, privacy, and evolvability. Upgrades to the Provider Adapter or Tool Manager do not require changes to the Workbench or Text Editor. The use cases confirm that routine editing flows and AI-assisted editing both follow clear paths. The open, plugin-friendly model remains a strength, though Electron-based I/O and inter-process messaging can introduce overhead. Future work should focus on caching, prompt construction efficiency in the LLM Message Service, and reducing latency in the File System Controller and Provider Adapter pipelines. Altogether, the architecture shows how Void evolves the VS Code base into an AI-native editor without sacrificing control or clarity.

Lessons Learned:

This project taught us how architectural choices show up in real workflows. Using the same terms across the report kept us aligned. Talking about the Workbench, Configuration subsystem, File System Controller, and Text Editor made it easier to reason about control and data flow in the open-edit-save use case. Likewise, naming the Code Edit Service, LLM Message Service, Tool Manager, and Provider Adapter helped us map the AI path from a /edit request to a diff the user can accept.

We also learned the value of diagrams that match our terminology. Drawing component and sequence diagrams with these subsystem names made the implicit invocation style visible. Events between the Text Editor, Code Edit Service, and Provider Adapter were easier to understand once we labeled them consistently. It also clarified what Void inherits from VS Code versus what Void adds for AI, such as the Tool Manager and LLM Message Service.

Teamwork mattered as much as the design. When a member dropped the course near the end, we were able to reassign sections because our tasks and subsystem boundaries were already clear. Shared outlines, version control, and frequent check-ins kept the Workbench and Text Editor sections consistent with the AI services write-up.

Finally, collaborating with an AI teammate helped with structure but required verification. The AI was useful for sketching the outline of the Code Edit Service and Provider Adapter flow, but we still confirmed details against sources and our own diagrams. The main takeaway is that consistent naming, clear diagrams, and steady collaboration make it easier to understand and communicate a layered, event-driven architecture like Void's.

AI Collaboration Report:

AI Member Profile and Selection Process

Our group chose ChatGPT (GPT-5, October 2025 edition) as our AI teammate for this project. We wanted a partner that could help us organize our architecture report, summarize source material, and ensure that our structure followed the course's expectations. Before choosing ChatGPT, we reviewed other tools, but its reliability, conversational format, and support for markdown and code made it ideal for technical writing.

We treated GPT-5 as an assisting team member rather than as a content generator. The main reason we picked it was to help us get unstuck during planning. We often had many ideas but struggled to fit them into a professional report structure. The AI gave us a starting framework and allowed us to refine it collaboratively.

Tasks Assigned to the AI Teammate

The AI was primarily used to create an initial outline for the final architecture report on Void. We asked it to include section titles, estimated lengths, and notes on what each section should contain. To guide it, we pasted the official project guidelines PDF directly into our prompt. That ensured the AI followed all required deliverables such as the abstract, use cases, data dictionary, and AI collaboration section itself.

We also gave it a custom example prompt that represented our team's real planning document. It included our group name, our member list, and specific responsibilities. We also included sample research links to Void, Visual Studio Code, and Electron architecture sources.

From this context, the AI generated a detailed outline that aligned perfectly with the course's expectations. It suggested section lengths (like two-thirds of a page for the abstract and four to six pages for architecture) and summarized what should go in each part. This became our working template for the report.

Later, we used GPT-5 to help us connect the architectural relationships between Void, VS Code, and Electron. We asked it to explain how the Renderer, Main, and Extension Host processes communicate, and how Electron acts as the base runtime. This helped us describe how Void inherits VS Code's structure but extends it through custom Large Language Model (LLM) provider adapters and a privacy-first plugin system. This step clarified the overall architecture for our diagrams and section explanations.

Interaction Protocol and Prompting Strategy

We agreed that only one team member (Karim) would interact directly with GPT-5. This kept the communication consistent and avoided overlapping prompts. Each prompt was reviewed by the group before being sent.

Our key prompt looked something like this:

> "Using the official CISC 322 guidelines (attached below) and the following team example outline, generate a detailed report structure for a software architecture analysis of the open-source project Void, a fork of VS Code. Include approximate page lengths and summaries for each section."

By including both the guidelines and our example outline, GPT-5 produced outputs that directly matched assignment requirements. This approach also prevented the AI from going off topic or giving answers unrelated to the course structure.

After receiving the AI's outline, we reviewed it line by line to confirm that it matched the course rubric. We also added our own notes and replaced any vague sections with more accurate content based on our research.

Validation and Quality Control

To maintain academic integrity, all AI-generated material was validated by the team. The outline served only as a starting point, not as final text.

We verified that:

- The structure matched every section from the course guidelines.
- All terminology was consistent with what was covered in lectures (for example, we confirmed the definitions of “layered architecture” and “microkernel plugin style” ourselves).
- Research links included by the AI were checked for accuracy and replaced with verified sources when needed.

Quantitative Contribution to Final Deliverable

We estimate the AI contributed roughly 15-25% percent to the overall report creation process, mostly during the early planning and organizational stages.

Area	AI Contribution	Human Contribution
Report outline & section structure	70%	30%
Connecting Void to VS Code & Electron	30%	70%
Writing, editing, and diagrams	0%	100%

Reflection on Human, AI Team Dynamics

Working with GPT-5 showed us how an AI teammate can save time by handling mechanical planning work. It gave our team a clear direction before we started writing, which helped reduce confusion and overlap. The AI was especially useful for structuring large, complex topics like the relationship between Void, VS Code, and Electron.

However, we also realized that AI output cannot replace real understanding. Some early outlines repeated content or made small factual errors about the Electron IPC system, which we caught during our review. The AI was strong at organization and summarization, but weak at technical accuracy unless we supplied examples or context ourselves.

In a future project, the best use of an AI teammate would be as a collaborative research assistant. It could help compare multiple open-source systems, generate architecture diagrams based on verified data, or summarize developer documentation before team meetings. The key lesson was that AI helps most when guided with rich, specific context, such as course guidelines or sample outlines, and when humans remain in full control of the technical content.

Overall, this collaboration saved time, improved our organization, and taught us how to manage an AI teammate responsibly within a structured academic workflow.

References:

[1] “Understanding Visual Studio Code Architecture.” (n.d.). *Medium.* Retrieved from https://franz-ajit.medium.com/understanding-visual-studio-code-architecture-5fc411fca07 Medium

[2] “VSCode · Delft Students on Software Architecture – delftswa.” (n.d.). *Delft SWA / DESOSA.* Retrieved from https://delftswa.gitbooks.io/desosa-2017/content/vscode/chapter.html DelftSwa

[3] “VSCode – From Vision to Architecture.” (2021). *DESOSA Project.* Retrieved from https://2021.desosa.nl/projects/vscode/posts/essay2/ DESOSA

[4] “Void Editor – Architecture Overview.” (n.d.). *DeepWiki.* Retrieved from https://deepwiki.com/voideditor/void/1.1-getting-started#architecture-ov8631-AE22erview

[5] “Visual Studio Code – Wiki.” (n.d.). *GitHub.* Retrieved from https://github.com/microsoft/vscode/wiki/

[6] “Void – GitHub Repository.” (n.d.). *GitHub.* Retrieved from https://github.com/voideditor/void/

[7] “Electron: Things to Watch Out for Before You Dive In.” (n.d.). *Medium.* Retrieved from https://medium.com/@vishaldwivedi13/electron-things-to-watch-out-for-before-you-dive-in-e1c23f77f38f medium.com

[8] “Void IDE: The Comprehensive Guide to the Open-Source Cursor Alternative.” (n.d.). *Medium.* Retrieved from https://medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235