Conceptual Architecture of the Void Editor

CISC 322 — A1 (Group 26: Moneyballers)

October 10, 2025

Authors

Shaun – 22QH17@queensu.ca
Maaz - 23RSMB@queensu.ca
Daryan - 22GMV4@queensu.ca
Tirthkumar - 23DWN3@queensu.ca
Shujinth - 21ss383@queensu.ca
Manu - manu.m@queensu.ca

Abstract

Void is the open-source AI powered code editor, which extends VS Code through AI powered intelligent developer assistance. Void integrates Large Language Models (LLMs) into the editing experience, enabling features such as automated refactoring, interactive code editing, and natural language code generation. This report presents the Conceptual Architecture of Void, focusing on its major subsystems, and how they interact with each other.

Void is organized in three main layers:
1) The editor and workbench interface
2) Service and orchestration layer
3) LLM integration pipeline

This report covers Void's architecture through services and actions structures. Control and data flow are handled through an internal LLM message pipeline and an apply workflow. The LLM enables concurrency as the edits are reflected in real time. The architecture supports future modifications with minimal disruption. All in all, this report on conceptual architecture shows how Void works, how its components interact, and how it can be evolved in future.

Introduction

Void Editor is an integrated development environment, which is built on VS Code and Electron, with AI assistance. Void inherits properties and interface of VS Code, while adding new AI powered capabilities. Void is a perfect example of integrating Large Language Models into an editing workflow. This functionality allows users of Void to interact with code while generating AI suggestions in projects.

This report documents the conceptual architecture of the Void editor. A high-level view of the system is provided, which is useful for developers and other stakeholders. This report highlights the system's functionality, data control, its major parts, and potential for future evolution. This is helpful for future planning, and maintenance of the system, as it simplifies all the major details. This is also crucial for explaining the system to the non-technical stakeholders as the conceptual architecture does not go in depth into technical implementation.

Void uses VS Code workbench and the model subsystems which provides the editing capability, file model management, and extension framework. Void's AI feature extends the base with pipelines for messaging LLMs, manages the settings and providers, and applies the model generated differences to the source code. Electron manages the platform level interactions.

This documentation focuses on the high level implementation and interaction. This is to simplify how the subsystems are divided, how all the different subsystems interact with each other, how data flows between components and how concurrency is supported. Focusing on these aspects, the reader will be able to gain a better understanding of how Void works as a whole.

Architectural Style of Void

Void primarily follows a layered architectural style. At a high level the system is organized into three logical layers. The User Interface Layer manages presentation to the user and allowing for user interaction through the workbench and chat panel components. The Editor Core Layer serves as the foundation of the editor and provides the text model, buffers and editing primitives. The Service Layer integrates subsystems such as the LLM Integration Service, Chat Orchestration, AI Code Application, Tooling, Configuration, and Update services. This layered structure enforces clear boundaries and separation of program concerns.

Void also shows characteristics of a repository style of architecture because the editor core acts as a central repository maintaining the current workspace state. Other subsystems read and write to this shared data model making it an important part of the architecture. Together, these two styles help the system's maintainability, consistency and evolvability.

Subcomponents

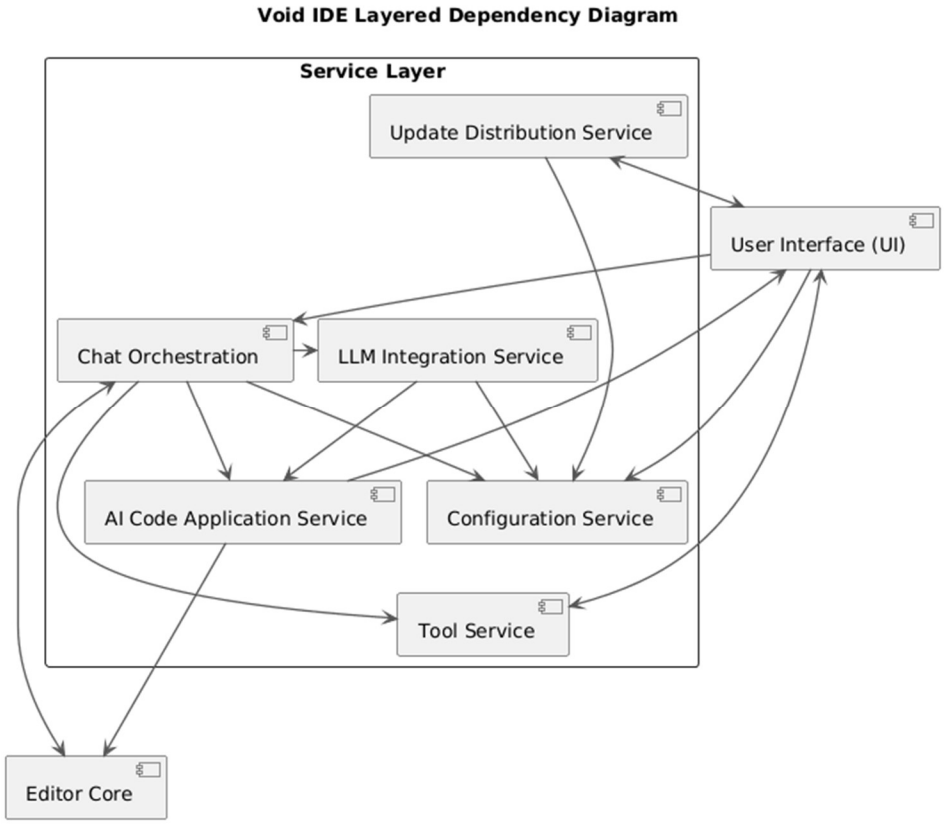

Figure 1: Void Dependency Diagram

1. User Interface (UI)

The User Interface (UI) is the presentational and interactive layer of the system. This subcomponent displays essential IDE components such as the source code editor, debugging panels, file explorer and output consoles for errors or build information. The UI also includes views for AI related features such as the chat panel for interacting with the integrated large language model (LLM).

The UI is the main point of interaction between the user and all other system components. It communicates with the Configuration Service by applying user preferences such as keybindings and themes as well as the Tool Service subsystem allowing the user to use code formatting or test running tools. The user can also interact with the Chat Orchestration system through the chat panel. The UI is crucial to the program's overall usability and responsiveness because every user command originates through the UI.

2. Editor Core

The Editor Core is the foundation of the IDE, it provides core editing capabilities and maintaining of the workspace version. It manages syntax highlighting, code folding, search, undo/redo operations and buffer management. The core also provides a set of APIs allowing other subsystems and extensions to safely read, modify and update file content through standard editor operations.

It interacts closely with the AI Code Application Service, which applies verified AI-generated edits directly to its text buffers. It also provides text snapshots and contextual information—such as the currently selected code block or file contents—to the Chat Orchestration subsystem allowing the AI to respond accurately to the user's context. For these reasons, the Editor Core is one of the most critical and performance-sensitive components in the system.

3. LLM Integration Service

The LLM integration service is a gateway between Void and the chosen external LLM either accessed locally or through the cloud. This subsystem manages model configuration, authentication and communication protocols ensuring secure and reliable interaction with external AI systems. This service handles provider configuration and authentication and acts as the mediator of obtaining data from the used LLM.

The LLM integration service is invoked by the Chat Orchestration subsystem, which sends user prompts and project context to the model for completion. The service then processes and formats the model's output. When edits are produced, the service transmits them to the AI Code Application Service for validation before integration.

4. Chat Orchestration

The Chat Orchestration subsystem coordinates communication between the user and the LLM. It manages chat sessions, message history and conversational context to determine how to request code or text completions from the AI model. This subsystem works in a similar fashion to a helper function because it turns user requests into precise AI requests.

The Chat Orchestration subsystem retrieves relevant code and selections from the Editor Core, packages it into a structured prompt and sends it to the LLM Integration Service. Once the model produces the suggested edits or responses, Chat Orchestration forwards these results to the AI Code Application Service for validation and safe application into the workspace. In “Agent Mode” the subsystem can also interact with the Tool Service in performing tasks such as searching within project files or running tests automatically. By maintaining conversational context and controlling AI interactions, Chat Orchestration ensures coherent communication between the user and the AI system.

5. AI Code Application Service

The AI Code Application Services acts as a guard against the LLM’s rare implementation of bad code into the user’s source code. It is also responsible for validating, visualizing and applying AI-generated edits in a controlled manner. This service retrieves proposed code from the AI and compares it against the existing code. The subsystem also detects syntax errors, conflicting edits or incomplete transformations before they are applied to the source code.

This subsystem receives suggested code changes from the Chat Orchestration and LLM Integration model subsystems. Once validated, the AI Code Application Service applies the approved modifications directory to the Editor Core. Visual diff views are also shown on the UI component allowing users to preview changes before actual integration of AI generated code. In summary, this subsystem prevents the introduction of errors and supports collaboration between the user and the AI model.

6. Tool Service

The Tool Service manages extensions, developer tools and command-line utilities enhancing IDE functionality. It supports language-specific formatters, test runners, and remote development tools, integrating them through a consistent API. This subsystem is a bridge between the core editor and external utilities enabling automation and customization of Void.

The Tool Service can be activated by the UI through user interaction or through Chat Orchestration when the “Agent Mode” option is enabled. For example, the AI may request a code search or initiate a code formatter run. Once a tool completes execution, the results are returned to the UI for display information and for future use. Since the Tool Service operates independently of the core logic of the overall system, tools can be added or removed without altering the system’s architecture.

7. Configuration Service

The Configuration Service centralizes user and system settings across the entire system. Preferences such as theme, keybindings, AI provider slection and feature toggles are all managed by the Configuration Service to ensure a consistent experience between user sessions.

The Configuration Service interacts continuously with the UI, LLM Integration and Chat Orchestration subsystems. The UI reads from and writes to the configuration files when the user changes preferences. The LLM Integration Service queries it for provider details and authentication credentials. Chat Orchestration references configuration flags to change program behavior at runtime.

8. Update Distribution Service

The Update Distribution Service handles application packaging, version control and cross-platform updates. The component ensures users always have acces to the most stable and recent version of the software. This service has the responsibilities of checking for updates, notifying the user of an update, downloading necessary data and coordinating the build of the application.

This subsystem primarily works with the UI subsystem to display update prompts, release notes and progress indicators to the user during an update. The subsystem also reads the application's build metadata and configuration information from the Configuration Service to ensure the correct version and settings are applied after the updates. By automating the update process, this service minimizes downtime and prevents version drift.

Control and Data Flow

The control and data flow of void is separated into the following:

User initiated action

The process starts when the user initiates AI assistance like typing, asking, or completing code. This action tells the the system to gather context and make an AI request

Prompt formation

As the action begins, Void collects contextual data like surrounding code, cursor position and language to form a well structured prompt. Like this, the AI model is provided with sufficient information to produce precise and context related responses.

Asynchronous request

The prompt is made as an asynchronous request, communication which is non-blocked enables the editor to stay completely active and be responsive, which allows users to continue on tasks as AI handles the request.

Response processing

After getting the answer from the AI, Void translates it and add it to the editor. This can be inline suggestions, explanations, or chat messages, depending on user action and request type.

Feedback loop

Finally, user interactions - like accepting, editing, or rejecting suggestions - are captured as feedback. This improves future requests and makes the system more adaptable and accurate over time.

Concurrency

Concurrency in Void is a fundamental design feature that enables smooth interaction between the developer, editor, and AI backends without loss of the performance. Because Void highly relies on foreign AI inference services locally or remotely hosted API endpoints, there is inherent latency and unreliability of response times. To avoid these, AI interactions are always made

asynchronously focused on event-driven programming, non-blocking execution in IDE. This means all operations such as, editing and debugging run in the background even when AI requests are being processed.

Void uses Node.js' single-threaded event loop and the asynchronous I/O model to control the concurrent operations with efficiency. Each AI request is executed as an autonomous promise or callback, and requests are dynamically queueable, throttleable, and cancellable based on user input. For example, Void can cancel tasks if code is modified by the user before an AI response is returned. This helps in ensuring real-time responsiveness while conserving system and network resources.

Parallelism is done by independent threads or worker processes as and when the need arises for it, particularly while handling many open files, active chat sessions, or code analysis tasks. With this kind of the architecture, Void supports concurrent AI-based features like auto-completion, document lookup, and semantic refactoring while also scaling linearly across many multi-core computers. To avoid these kind of situations between concurrent operations and to preserve the data consistency, concurrency control strategies like the rate limiting, mutex locks over shared state, and request queues are used.

Void offers high responsiveness and also stability under large workloads by combining intelligent concurrency control with task scheduling which is asynchronous. The system can be expanded in the future with distributed or higher-level AI services without sacrificing performance or dependability thanks to this design decision.

Application Evolvability

The Void application was created with evolvability in mind. Like VS code, Void adopts a layered architectural style, which allows for future modifications with minimal impact on the overall system. Each layer of the system is responsible for a different task and interacts with other layers through clearly defined interfaces. These tasks and defined interactions help to reduce layer interdependence. As a result, Void can easily change a module without having to make massive changes to the overall system.

The Repository-style data management within the Editor Core also supports evolvability by maintaining a single centralized workspace state. This centralized data model helps in avoiding layer conflicts where different layers have separate versions of the same code.

Additional functionality to the system is provided through the Tool Service and Update Service subsystems. The Tool Service allows external plugins or formatters to be added or removed independently, extending the editor’s functionality without altering its architecture. The Update Service streamlines the deployment of improvements, ensuring that architectural updates and new capabilities reach users easily. Together, these design choices allow Void to evolve continuously without the need for intrusive modifications to its foundational codebase.

Implications for Division of Responsibilities

The architectural design of Void allows for a well-ordered division of responsibilities among developers. Since Void is an AI-enhanced fork of Visual Studio Code, the modular organization of the system inherently ensures segregation of concerns within different layers of functionality. This ensures teams are able to work independently and can also avoid potential integration conflicts. Developers at the extension integration layer are responsible for integration with the Visual Studio Code API, event handling, command management, and seamless integration in the context of the IDE.

The AI interaction layer is managed by developers interacting with local or remote models, including prompt construction, asynchronous request and response, and response processing. Meanwhile, the backend and infrastructure team takes care of API communication, caching, rate limiting, and authentication-related operations to ensure smooth and efficient data transfer between the editor and AI services.

The user interface and experience team is focused on front-end features. For example, chat windows, code suggestion panels, and configuration menus, with concern for usability and design consistency. Data management and persistence experts handle secure storage of user settings, logs, and cached responses.

This division of labor not only increases modularity and maintainability but also allows for scalability and iterative development. It is even feasible that teams may add or replace components-separate items like the addition of new AI providers or prompt optimization improvement-without introducing instability at the system level.

Use Cases

Use case 1: Code Generation

In this Situation, the Developer requests the integration of a search algorithm within a file using the chat bot in the User Interface. This is an asynchronous request, allowing the developer to continue working while the IDE is working on the prompt in the background. The UI/Client then sends the prompt requests, and limited files to the Chat Service. The Service then requests the files' contextual data from the Editor/IDE as well as any applicable settings and formatting and returns. The Chat Service then sends the prompt package containing all data and contexts needed to efficiently and accurately implement the request to the LLM. Generated code is sent to the AI Code Application Service. The Application Service compares the generated code to a previous version, raising issues that arise. If no problems are found, code is overlayed in UI, and UI requests Developer for implementation. If problems are found in code, AI Code Service sends a summary of issues to the Chat service, which packages along with original prompt and data package, to LLM. Then re-compares code. UI Sends requests and waits for user approval, once approved, sends code to Application Service, which inserts code in the Editor Core, and replies async that the code has been updated.

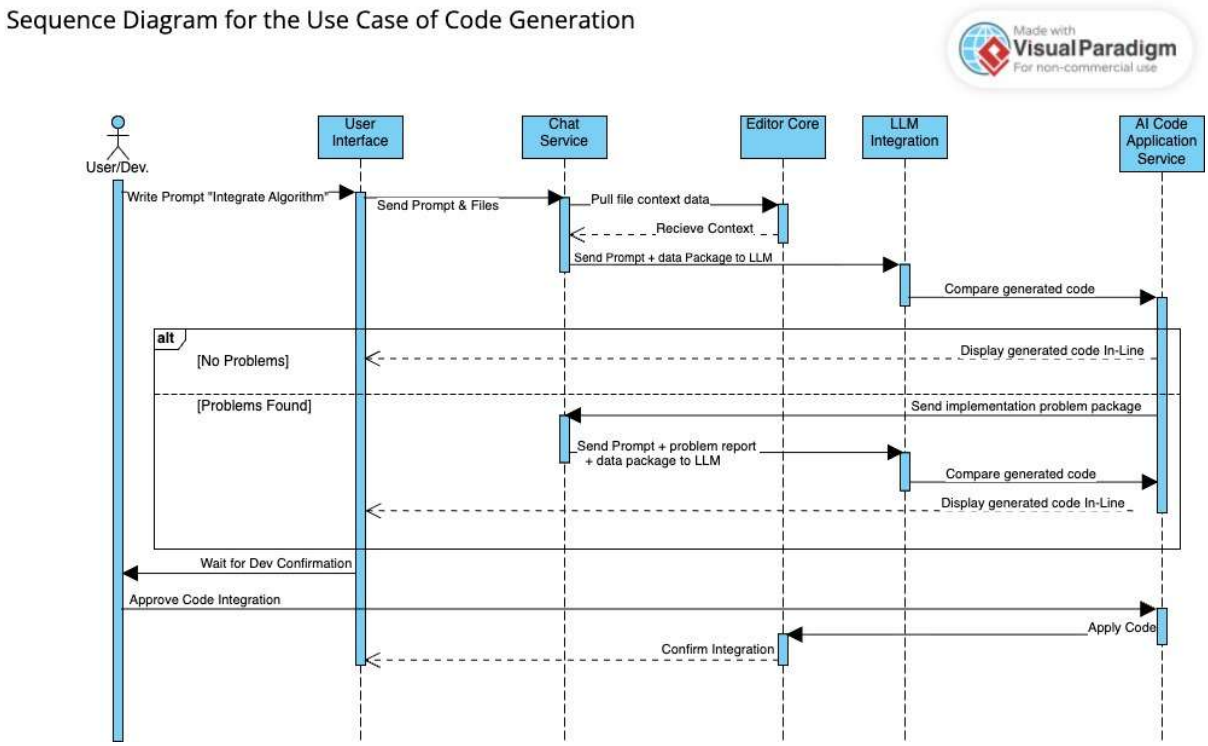

Figure 2 Code Generation Sequence Diagram

Use case 2: Code Refactoring From selected code

The interaction begins when the user (developer) selects code in the IDE and issues a request through the user interface with VOID's in-line assistance to modify said piece of code. The UI forwards the request to the chat service, which uses the editor core to gather necessary context, such as selected code, language, relevant dependencies. The editor core returns the necessary contextual information to the chat service, which then constructs a proper prompt and sends it to the LLM integration service, which actually generates the refactored code. Throughout this time, the IDE is usable, as the request was made async. Once the refactored code is generated, the LLM integration service sends it to the chat service, which then to the AI code application service for the validation of correct implementation without errors. There, it is then sent to the user interface for user review. Once it's approved, the code is sent to the editor core to be applied to the file locally, and then replies asynchronously to the UI to inform that the implementation was successful.

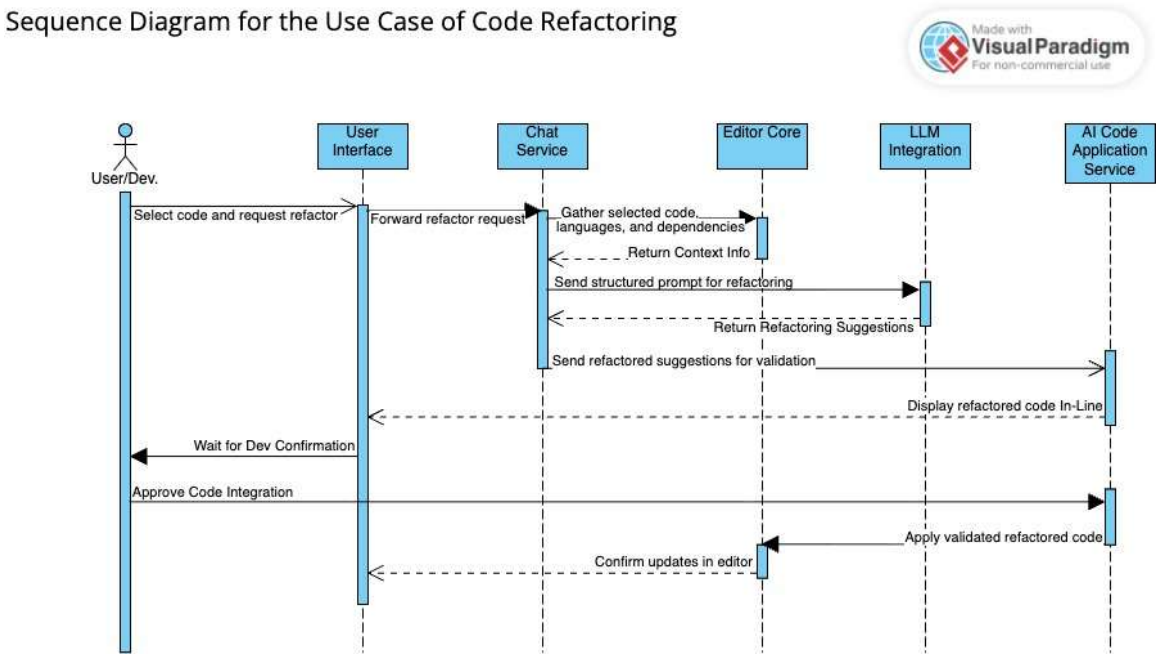

Figure 3 Code Refactoring Sequence Diagram

Naming Conventions

Each subsystem or module is named using a descriptive noun phrase reflecting its role. For example, LLM Integration Service or Chat Orchestration. Broader architectural tiers are referred to as Layers such as the UI Layer. Data items and interaction artifacts, such as Prompts, Diffs, or Edit Proposals, follow CamelCase formatting consistent with TypeScript naming conventions.

Data Dictionary (Glossary)

Editor Core: Editor core handles the core editing operations such as text manipulation, syntax highlighting, and exporting context to AI services for prompt generation and code application.

Chat Orchestration: Chat Orchestration is a sub-system that handles user-AI communication such as prompt construction, message routing, and response handling.

AI Code Application Service: Responsible for applying, comparing, and validating code changes produced by AI.

Tool Service: In-built collections of tools and plugins that facilitate supplementary work such as debugging, testing, and formatting.

Configuration Service: Configuration Service manages user settings, API keys, model selections, and other locally persisted or system configuration-managed customizable values.

Asynchronous Request: An asynchronous communication pattern used when making requests to AI models so that the user can modify concurrently while waiting for returns.

Concurrency: The system capability to execute more than one operation at a time or in partially overlapping time periods so that the system responds promptly in real-time and is stable as well.

Prompt Creation: The process of creating an AI query by obtaining relevant contextual information from the current environment of the user, i.e., selected code, language, or comments.

Extension Integration Layer: The architecture layer that allows Void's services from within to communicate with the VS Code API, manage commands, and facilitate frictionless IDE usage.

LLM Integration Pipeline: The official process through which user input is transformed into prompts, sent to AI models, processed, and returned as actionable code or text in the editor.

Caching: Temporary data storage used to reduce latency by keeping commonly accessed AI completions, refactorings, or configurations.

API: A defined set of communication protocols allowing Void to interact with external systems such as AI model suppliers or internal VS Code components.

Singleton Service: A software design pattern used in the architecture of Void to ensure each core service (i.e., Configuration or Chat Orchestration) will have only a single global instance available throughout the system.

Conclusion

The Conceptual Architecture of Void, demonstrates the application's structure and modular approach to integrating AI capabilities into a development environment. Void is an example of blending IDE features with advanced AI capabilities helping developers to write, refactor, and optimize code more efficiently. The layered architecture style ensures clear separation of application concerns, maintainability, and scalability, all crucial for the evolvability of a system with a lot of moving parts like Void. The Repository style architecture allows teams to work independently on different layers and subsystems without having integration conflicts. Concurrency and Asynchronous communication enable real-time AI interactions without interrupting developer workflow. These AI interactions are controlled through an AI integration pipeline using the Chat Orchestration and LLM Integration services helping to balance automation and user oversight. Functionalities like code suggestions, edits, and conversational interaction are validated through the AI Core Application Service before being applied to the user's source code. Overall, Void highlights the importance of Layered and Respository styles of architectures in integrating AI into development systems. These architectural styles also demonstrated that dividing the system into layers can be a large help in improving the maintainability and scalability of a program. This system as a whole empowers developers by through integration of AI assistance into the development model.

Lessons Learnt

Working with the Void IDE challenged us to think about the different mechanisms and subsystems for a large AI assisted software system. Through vigorous research and

documentation, we learned how large, complex systems have properties decomposed into separable layers, such as User Interface, Chat Service, Editor Core, LLM Integration, etc.. Furthermore, early in the project we spent a lot of time describing how the code actually works, this gradually shifted to conversations about architecture functions and relationships. This distinction was useful in enabling us to think like software architects instead of individual developers. This also helped us to focus on aspects of void that would actually be beneficial to this term project; principles like modularity, separation of responsibilities and asynchronous flow. Finally, in this project, and specifically this deliverable, we had used an AI in order to help organize responsibilities and obtain supplementary research in architecture analysis and of inner workings of void. AI aided in explaining purpose and interactions of each subcomponents as well as dividing up team responsibilities evenly so that every group member was able to contribute to the group's success. However, we also learned that human judgement is indispensable for true, contextual accuracy. Often times our AI helper provided incorrect information that needed to be verified to avoid falling into an incorrect understanding of how the system actually works. Overall, this experience strengthened our understanding of architectural thinking and teamwork, showing us a balance between structured design, reliable collaboration, and the careful use of AI tools.

AI Collaboration Report

AI Member Profile and Selection Process:

For this project, our team chose to collaborate with ChatGPT (OpenAI GPT-5 October 2025 version) as our AI teammate. Our rationale behind choosing this AI is because of its strong capabilities in architectural explanation, as opposed to other models we tried like Claude or Gemini. Given that our project, and specifically this deliverable, was focused around architectural understanding and diagramming Void's architecture, we needed an AI that could help us understand and give context to complex subsystem interactions and explain it in a way that was easy to digest. ChatGPT did this. Our selection process was entirely centered around comparing different tools (Claude and Gemini) and seeing which one had the most ideal output in terms of accuracy, information digestibility, and practicality. After our early group meetings, we decided to continue with ChatGPT.

Tasks Assigned to the AI Teammate:

1. **To clarify subsystem roles and interactions:**

   We asked ChatGPT to simply subsystem information found within the GitHub repository and overview pages for Void due to the complex wording of explainations and a lot of web terms we had not learned. The AI also helped by suggesting how we would explain each subsystem in the report (User Interface, Chat Service, Editor Core, LLM Integration, File System, etc.) fulfilling all assignment requirements. This helped the team deepen our general understanding of how these components interacted and by extension helped us in gaining a architect point of view of Void. The information learned from this was vital in order to design the sequence diagrams.

2. **Sequence diagrams Narrative descriptions:**

ChatGPT assisted making sure we included all the important subsystem parts in structuring the narrative descriptions for the "Code generation" and "Code refactoring" use cases. This was helpful in ensuring the use case diagrams were complete and correct. AI was also used to help organize the flow of readability so that it would be easier to understand.

3. **Organizing and Distributing Team Responsibilities:**

Early in the deliverable, the AI helped us divide this report up evenly between the group members in appropriate sections so that everyone would contribute equally to the work. The AI was used to draft an email to the rest of the group outlining division of responsibilities and what needed to be done so that workflow was easy to follow.

4. **Supplementary Research on Software Architecture:**

The AI was asked to summarize architectural concepts such as "asynchronous communication", and "service layering" to provide foundation for the report.

Interaction Protocol and Prompting Strategy:

Each team member interacted with ChatGPT (GPT-5) independently based on their assigned tasks. The team lead used it to help organize responsibilities, while others used it to clarify subcomponents and general architectural terms to better understand Void. We used the basic understanding of the systems to delve deeper into architecture overviews and GitHub repository so we could accomplish our tasks. We all used an iterative strategy of prompting to gain more information about subsystems from the AI. We offered an initial prompt, reviewed the response, then tried to gain more information such as how a subsystem interacted with another subsystem repeatedly until we had a basic understanding of a system and where to look in the repository for more details. Overall, we used human judgement in reviewing AI work to ensure it was contextually accurate.

Validation and Quality Control Procedures:

All ChatGPT generated work was reviewed and verified by one ore more team members before inclusion into the report. A member, usually Maaz, was assigned to cross reference chatGPT information with either the void IDE github repository or other credible secondary sources online. If the AI was asked to provide architectural explanations we compared them against notes from class or verifiable sources on the internet. Nothing generated from AI was copied without being reviewed for contextual and factual accuracy.

Quantitative Contribution to Final Deliverable:

The AI contributed approximately 20% in aid to the final deliverable. Our primary use of AI was to clarify purpose and interactions of individual subcomponents/subsystems. This information was key to drafting sequence and dependency diagrams, as well as actually understanding Void. The ability to use AI to develop a shared understanding of the system architecture was something that was really helpful towards the group's success in this deliverable. Also being able to organize and break up the work between the team just by using a prompt aided in not wasting time on grunt work which meant there was more time spent doing the actual report, increasing quality of work. Though all major writing, analysis, and diagram creation was completed entirely by the team, the AI’s role in supporting the team brought a substantial difference to work quality then if we had not had it.

Reflection on Human-AI Team Dynamics:

Integrating an AI teammate into our workflow only had a supportive and positive impact to the project. The AI helped streamline and broaden our understanding of system architecture and specifically the system architecture of void. AI also saved a lot of time on initial planning and workflow so that we had the ability to jump straight into the actual project itself. Furthermore, AI influenced brainstorming by providing explanations on specific subsystems allowing us to expand on the basic ideas of how the components work. It was also helpful in reminding us of how a subsystem worked with other system when we had misunderstandings. There were few disagreements caused by the AI, but the team had to be careful about trusting AI to educate us instead of looking into Void sources themselves. However, this verification time was worth the

effort given how much help AI was able to give with clearing up certain points of a subsystem without us having to go down rabbit holes in sometimes unclear documentation. Overall, this experience highlighted the fundamental importance of human judgement when working with AI. We learned that AI is most useful as a supplementary research tool rather than an author. These lessons taught us to emphasize the value of AI as a support rather than a replacement for human decision making.