Concrete Architecture of the Void Editor

CISC 322 — A2 (Group 26: Moneyballers)

November 7, 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

This report is the second part of the course project and covers the concrete architecture of Void through its subsystems along with its integration of Electron and VS Code. In A1 we described the conceptual architecture, described the design of Void and touched on Void's subsystems. This report goes deeper into how Void actually works using the Understand software to see dependencies.

The dependency diagram shows layering, subsystem boundaries, and dependencies that were derived by analyzing the source code and folders. While building this report, we found some consistencies and divergences between the conceptual and concrete architectures discussed in how subsystems were not always clearly within their own module. The top-level analysis now includes important foundational software being VS Code and Electron. The in-depth analysis covers the interactions between Void's components and how Void integrates the LLM to add editor functionality.

The major architectural intent is the layered structure separating the UI, Orchestration and LLM Integration components. This report also gives an insight on how the architecture of Void has evolved from the initial conceptual architecture to the actual concrete architecture.

Introduction

This A2 report explains the concrete architecture of Void, which builds upon the conceptual architecture that we had in A1. While in A1 we explained the conceptual architecture of Void and its subsystems, in this report we will be focusing on the concrete architecture and how void actually interacts with other components. This report gives insight to how Void's real implementation aligns and differs from the initial design. This was done by identifying the factors responsible for this architectural drift and the inclusion of dependencies during the actual development process.

As discussed in A1, Void is an AI powered extension of VS Code which integrates LLMs for tasks like natural language code generation. Since Void is an extension of VS Code, the architecture of Void inherits VS Code's framework. This means this report will be focused on the top-level architecture that connects Void and VS Code, and the subsystems of Void. Void primarily follows a layered architectural style. At a high level, the system is organized into three logical layers:

1) Interface Layer
2) Orchestration Layer
3) LLM integration Layer,

The Understand software tool was used to obtain the dependency diagrams for comparing Void's conceptual and concrete architectures. The report will start by explaining the concrete architecture of Void using the source code, it derives the subsystem within the Void and VS Code environment, also it explains the interactions in the form of diagrams. Also, this report contains a detailed explanation of one subsystem, identifying its internal components and dependencies.

Then there is a reflection analysis, which compares the conceptual and concrete architecture, and highlights both the consistencies and divergences.

This report will give a deeper understanding of how the Void's architecture aligns with the conceptual architecture, as it will unveil the actual dependencies and clarify how Void integrates with VS Code. Also, by examining divergences, this report aims to provide the conceptual architecture from A1 to better reflect the system's actual reality. This report shows the importance of reflection analysis in software architecture, as it shows how the actual planning differs from the actual implementation.

Derivation Process

This section will dive into how Void's concrete architecture was derived. The process began systematically, combining the conceptual architecture from A1, the dependency extraction from the source code on GitHub and use of manual validation. Ultimately, our objective was to map high level conceptual components discussed in A1 into refined concrete subsystems to visualize their interactions in the codebase.

Initially, we reviewed A1 feedback to refine understanding of Void's architecture, specifically we concluded that another major architectural style relevant to Void is the client-server style. Reviewing the conceptual architecture also provided a refresh on the systems primary components and their responsibilities (Editor Core, Chat orchestration, LLM integration, etc.). These components formed the basis for mapping the concrete architecture.

Next, the source code was imported into SciTools Understand software to visualize dependencies and communication flows between the various components. Each top-level entity was represented as a subsystem which was mapped to by code files. In the case that a code filename was ambiguous or hard to track, looking through documentation and the actual code itself had to be done to manually validate the correct assignment. We did this iteratively until all dependencies had been determined to formulate our concrete architecture. Afterwards, we used Understand's internal dependency graph and visualization tools to identify how the subsystems actually interacted with one another and analyzed the flow of data between them. We compared this with our dependency diagram of our conceptual architecture in A1. The dependencies were examined by looking at the source code on GitHub. This step was vital to formulate our reflection analysis and develop our two use cases and their accompanying sequential diagrams.

High-Level View of Concrete Architecture

Electron:

Electron is a framework that combines both chromium and Node.js for rendering web pages and to build cross platform applications respectively. Electron's main components are two core processes: the main process and the browser process. The main process controls the lifecycle of the application, like creating tabs, handling menu features, events and managing inter-process communication. The browser process renders the user interface with web technologies and handles user interactions; it also manages running web pages inside the browser powered by chromium.

VS Code architecture:

VS Code follows the layered architectural style and is divided into four core layers: the workbench, editor, platform, and base layers. The workbench component is the topmost layer, it provides an overall user interface and coordinates different UI components such as the terminal, file explorer, debug views and source control. The workbench also handles user commands, theme preferences, menus and extensions.

The other main layer is the editor layer which implements the Monaco editor. This is the main layer in VS Code as it manages VS code's text-editing feature. The

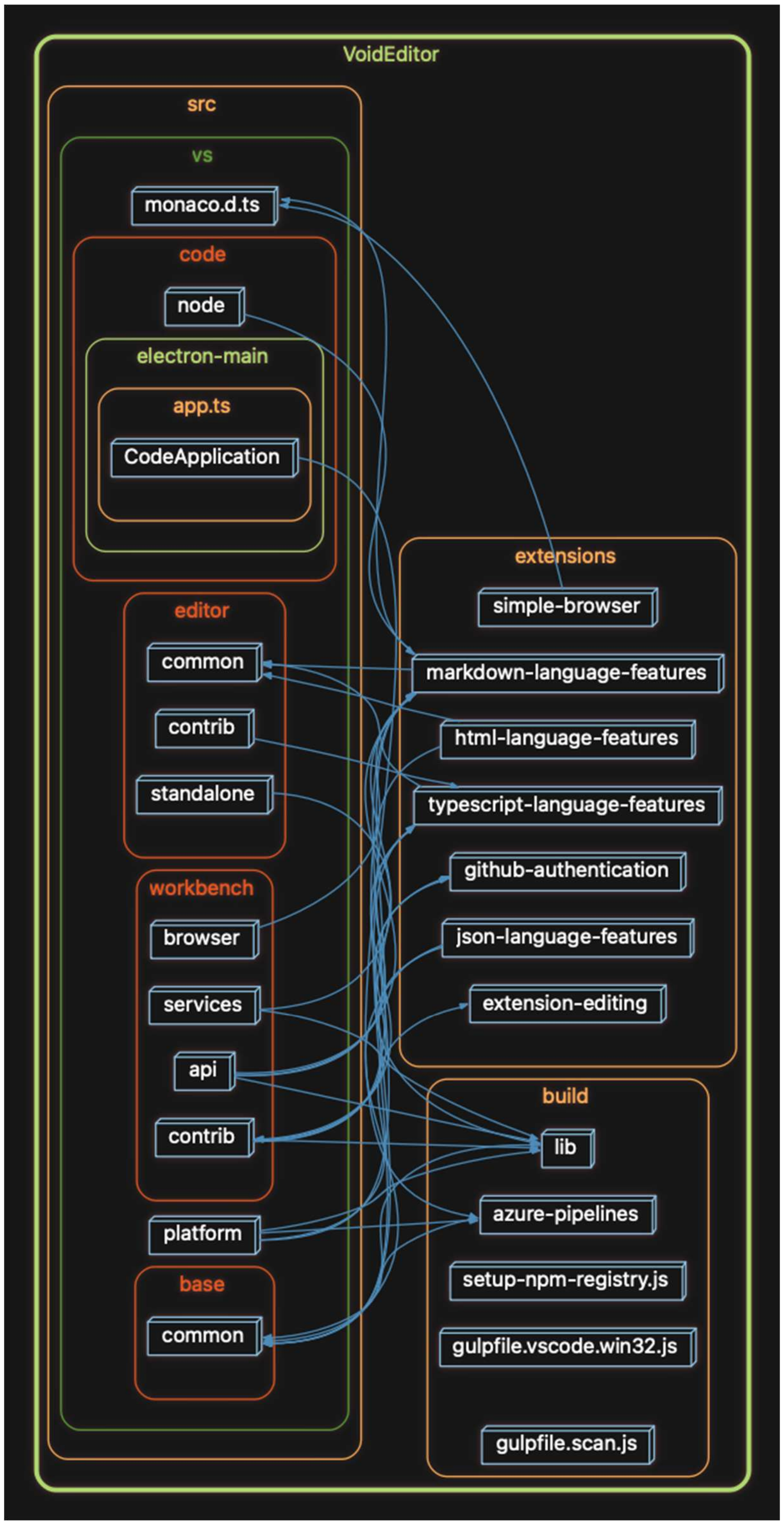

Figure 1: Void Dependency Diagram

editor is built upon low level layers to provide features such as text and code editing. The Workbench and Editor layers are part of Electron's browser process.

The platform layer comes under the main process layer in Electron, and both workbench layer and editor interact with the platform as this component handles all system-level services that interact with corresponding OS or runtime environment, this layer manages file systems, I/O operations. It provides APIs for storage, networking and for process management.

All three of the above directly interact with the base layer which is the foundation block of the VS code. It consists of libraries, low-level utilities and other core components which are essential for the system.

Void Subcomponents:

1. User Interface (UI)

The User Interface (UI) is the presentational and interactive layer of the system, functioning within the browser process of Electron. 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 key bindings 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. Since the browser process handles rendering and user interactions, the UI communicates closely with underlying services through inter-process communication managed by Electron. 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. Like VS Code's Monaco editor, Editor Core runs in Electron's browser process along with the UI. It's the lower half of the Workbench layer, sitting on top of the underlying Platform and Base layers that provide file system and runtime services. 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. In the Electron framework, the service resides in the main process layer and communicates with external APIs to exchange data securely with remote LLM providers through defined protocols. 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. At a high level, architecturally, Chat Orchestration bridges between the browser process (UI and Editor Core) and main process (LLM Integration Service), utilizing Electron's IPC channels to forward prompts to the LLM and receive streaming responses back in real-time. 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 occasional 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. In Electron, the AI Code Application Service runs primarily in the browser process, interacting with the Editor Core and the UI to apply validated edits. It depends indirectly on the main process through Chat Orchestration and LLM Integration for receiving AI-generated suggestions. 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 abstracts the execution of tools behind a service layer, allowing it to support asynchronous operations and keep the UI responsive by separating out these potentially blocking tasks. This is somewhat similar to how VS Code decouples editor operations from external tools, enabling Void to safely extend or replace tooling without affecting the performance of the core editor. 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, key bindings, AI provider selection and feature toggles are all managed by the Configuration Service to ensure a consistent experience between user sessions. This service underpins consistency across both Void and VS Code layers, enabling UI components, LLM Integration Service, and Chat Orchestration to dynamically query or update preferences. By abstracting configuration management, it reduces coupling and ensures that changes in user settings propagate safely throughout the IDE. 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 access 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. The service is set up to tightly integrate with both the UI and configuration layers for consistency in providing a seamless user experience: update notifications, progress indicators, and post-update settings reflect the user's preference. Similarly, it is designed to mimic the way VS Code does updates to be consistent across platforms, so users do not experience disruption with this automated update. 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.

Architecture Styles

Layered Style:

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 allows 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.

Client Server Style:

As Void splits into Electron's browser process and main process it also follows a client -server architecture style, though with nuances due to being a desktop application. In this the browser process acts as the client, hosting the UI where the users interact with the chat system and the editor. The main process serves as the server, that the browser process cannot perform directly, like communication with local or external LLM providers, managing files access, and running tools, IPC channels implement the communication between the client and server, such as llmMessageChannel.ts allow structured data such as LLM responses, chat prompts and outputs to move back and forth efficiently. This style separates UI logic from that of business logic and important operations, allowing the server to process the requests originated from the client, and return the responses back in real time. This mirroring of client-server style internally in Electron enables modularity and extensibility.

Pipe and Filter Style:

The design of Void does not fit the pipe-and-filter architectural style, since the system is not based on data flowing sequentially through independent processing components. Pipe-and-filter architectures are best utilized where data flows in a straight-line sequence of transformations, for example, compilers or data processing systems. Contrasting with such an architectural style, Void is an interactive, event-driven platform wherein a multitude of subsystems interact with each other, along with responding to the user's actions in a dynamic instead of linear way. Consequently, its architecture heavily concentrates on the well-structured distribution of responsibilities among different layers: the User Interface Layer handles user interactions and presentation; the Editor Core Layer provides text models and operations of editing; the Service Layer connects various subsystems around LLM Integration, Chat Orchestration, and Tooling. These layers interact in a hierarchical manner, where higher layers depend on the service of lower ones but there's no continuous streams of data. So, as mentioned above, the layered architecture style fits better, emphasizing on modular design and also controlled dependencies.

Detailed Subsystem Analysis

The Chat Orchestration subsystem is implemented primarily within the workbench layer and is one of the most important components in the overall architecture of Void. This component is the central coordinator that manages all interactions between the user interface and the LLM Integration Service. The subsystem's main logic is contained within the chatThreadService.ts file in the workbench directory. This file defines the ChatThreadService class which handles the creation, updating and management of all chat conversations - known in the system as threads.

Each thread represents a single conversation instance between the user and their chosen AI assistant. The thread contains all user messages, AI responses as well as important metadata such as file context, timestamps and message order. Each thread is identified by a unique ID called threadID which the ChatThreadService uses for keeping track of active and saved chat sessions.

This allows for separation of multiple independent conversations, each with its own context and history.

ChatThreadService also manages both incoming messages from the user in the UI layer and the outgoing responses generated by the LLM through the appendUserMessage(threadId: string, message: IChatMessage) and appendAssistantMessage(threadId: string, message: IChatMessage) methods.

Firstly, when the user types a prompt into the chat input box, the input is captured by the executeChatCommand(editor: ICodeEditor, prompt: string) method within the UI layer. The user's text is forwarded to appendUserMessage() which stores it as part of the current thread's message list for the next stage of processing.

Once the user message is registered, the subsystem prepares it to be readable to the LLM using the buildLLMMessage(messages: IChatMessage[], context: IChatContext): ILLMRequest method within the convertToLLMMessageService.ts module. The method constructs a package including the conversation history, relevant editor context and message roles into an ILLMRequest object.

The prepared ILLMRequest is sent to the LLM Integration Layer through sendLLMMessage(request: ILLMRequest) located in llmMessageChannel.ts which is an electron method.

Once the request reaches the LLM, the model begins generating a response. Most LLMs stream its output token by token—each token being a small piece of the final text. The ChatThreadService listens for partial responses through the onMessageChunkReceived(threadId: string, chunk: string) which progressively appends each incoming token to the assistant's current message. When the complete response has been received, appendAssistantMessage() finalizes the assistant's message and updates the chat UI through event emitters.

If the model's output includes tool invocation or a code edit, the LLM Integration Layer component would delegate those tasks to the Tool Service or to the AI Code Application Service components. The tool invocation would be called through runTool(toolName: string, args: any[]): Promise<void> within the toolsService.ts file. Code edits would be handled through the applyCodeEdit(edit: IEditBlock, target: ICodeEditor) within the editCodeService.ts file.

Finally, every conversation is kept for long term storage and retrieval using the threadHistoryService.ts module using saveThread(threadId: string, data: IChatThreadData). This ensures users can close and reopen the editor without losing chat history.

These modules form a layered pipeline where user input is captured and stored by chatExecuteActions.ts, transformed into LLM ready format by convertToLLMMessageService.ts, streamed back token by token to chatThreadService.ts and finally persisted by threadHistoryService.ts.

Reflection Analysis: Conceptual and Concrete Architecture

Reflection Analysis was conducted by mapping high level components from the conceptual architecture to the concrete subsystems using the understand software, the source code from GitHub, and manual inspection. Subsystems included UI, Editor Core, LLM Integration Service, Chat Orchestration, AI Code Application Service, Tool Service, Configuration Service, and Update Distribution Service. Divergences were analyzed and extracted from the source code through use of understand.

High Level Architecture Divergences:

Consistencies:

- Layered Structure: Conceptual layering into the interface, orchestration and LLM integration layers is reflected within the concrete architecture, through alignment of the UI, editor Core, and Service layer within those 3 layers
- VS Code and Void integration: The concrete architecture preserves the separation between both void and VS code. This is shown through Electron's browser process hosting the UI/editor while the main process manages LLM and tool interactions
- Architectural Styles: The layered architectural style identified in the conceptual architectural report in A1 remains applicable as at a high level the system remains organized into 3 logical layers

Divergences:

- Unexpected Dependencies:
    - From the concrete architecture we discovered that LLM messaging is handled primarily in the UI layer before being sent to workbench contributions. This diverges from the centered orchestration route in the conceptual architecture
    - In the concrete architecture, services like the Tool Service and the AI Code Application show coupling which is tighter to the UI then what was originally conceptualized
- Relocated Components: Conceptual components like the orchestration layer were split across the UI and the Chat Orchestration. This allowed for more distribution in the concrete implementation then what was originally conceptualized.
- Architectural Styles: Though the layered architectural style is still applicable, workings of Electron's browser process revealed the validity of a client-server architectural style as well. Electron's browser process functions as the client (UI/Editor) whereas the main process functions as the server (LLM int., Tool service).

Subsystem-Level Divergences (Void):

Missing or merged components:

- Orchestration Layer: In the concrete architecture the orchestration layer is split between the UI and the Chat Orchestration subsystems. In the conceptual architecture tasks were expected to be centralized in orchestration (I.e. LLM message preparation). Now, in the concrete architecture they are partially handled in the UI layer.
- Editor Core & Tool service: In the conceptual architecture the Editor Core and Tool Service were intended to be isolated from AI interactions, however, in the concrete architecture some AI interaction utilities were integrated into the Editor Core and Tool service, resulting in more tightly coupled subsystems
- Configuration & Update Services: some conceptual configuration utilities and other modules were merged into other services like the Configuration service and the update distribution service in the concrete architecture to make way for more simplified implementation.

Bidirectional Dependencies:

- Chat orchestration ↔ Editor Core: chat orchestration will read and write to the editor core, which violates strict layering outlined in the conceptual architecture. This will allow realtime contextual information to be able to flow properly for LLM operations.
- Chat Orchestration ↔ Tool Service: In Agent Mode, chat orchestration is able to get tools from the tool service and receive results back. This was not anticipated in the conceptual architecture
- UI ↔ Chat Orchestration: the AI interaction is mostly handled in the UI layers before being forwarded to the Chat Orchestration, which opposes the conceptual architectural reasoning of an orchestration centric routing.
- IPC Feedback Loops: feedback loops between the UI/Editor Core and the LLM Integration service and tool service are produced by Electron IPC channels, which was not conceptualized.

Use Cases:

Use Case 1: Code Generation

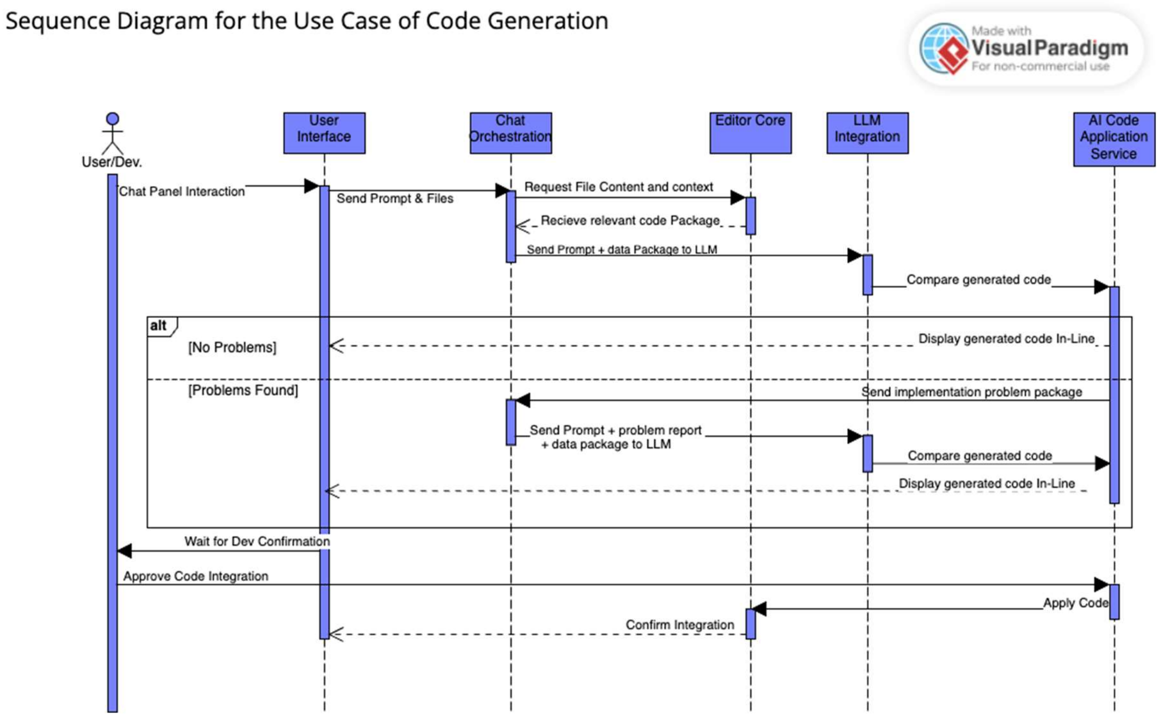
Sequence Diagram for the Use Case of Code Generation
Made with Visual Paradigm For non-commercial use
User/Dev.
User Interface
Chat Orchestration
Editor Core
LLM Integration
AI Code Application Service
Chat Panel Interaction
Send Prompt & Files
Request File Content and context
Recieve relevant code Package
Send Prompt + data Package to LLM
Compare generated code
alt
[No Problems]
Display generated code In-Line
[Problems Found]
Send implementation problem package
Send Prompt + problem report + data package to LLM
Compare generated code
Display generated code In-Line
Wait for Dev Confirmation
Approve Code Integration
Apply Code
Confirm Integration

Figure 2: Code Generation Sequence Diagram

When reviewing the past Use Cases, the conclusion was made that the logic for this Use Case's sequence diagram of the Conceptual architecture closely followed the logic of the actual program when searching through Understand. Minor adjustments have been made, including but not limited to correcting the names of the Modules, and fixing missing context/data from messages that were discovered during the creation of this concrete architecture. In this Situation, the user interacts with the chat panel, requesting Void to assist implementing a search algorithm within the program. This is an asynchronous request, allowing the developer to continue working while the User interface sends the prompt and relevant files to the Chat Orchestration, which requests the files content and context from the Editor Core. Once received it then sends that prompt plus the data and context package 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 awaits a developer response for implementation. If problems are found in code, AI Code Application Service sends a summary of issues to the Chat Orchestration module, which packages along with original prompt and data package, to LLM and 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 asynchronously that the code has been updated.

Use Case 2: Code Refactoring

Sequence Diagram for the Use Case of Code Refactoring

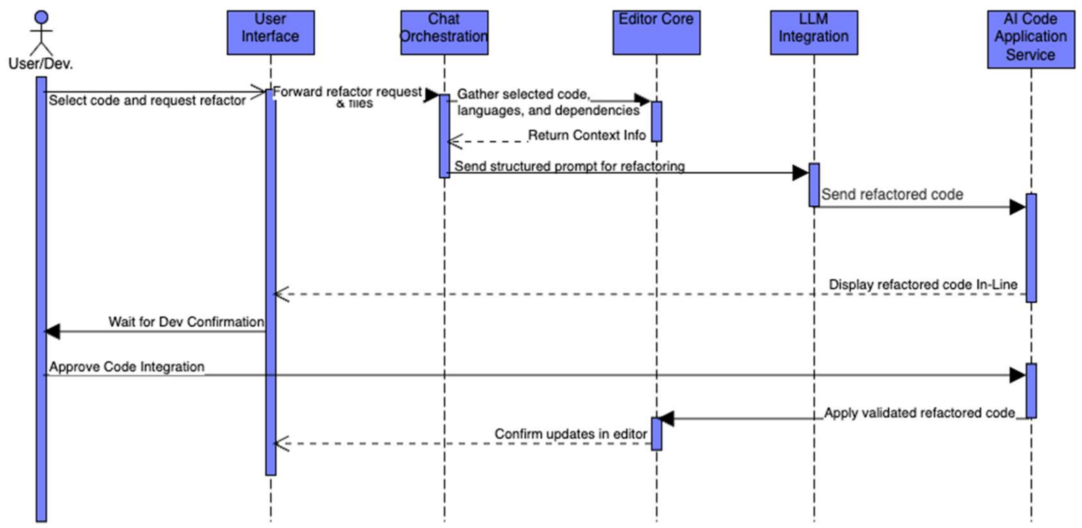
User/Dev.
User Interface
Chat Orchestration
Editor Core
LLM Integration
AI Code Application Service
Select code and request refactor
Forward refactor request & files
Gather selected code, languages, and dependencies
Return Context Info
Send structured prompt for refactoring
Send refactored code
Display refactored code In-Line
Wait for Dev Confirmation
Approve Code Integration
Apply validated refactored code
Confirm updates in editor

Figure 3: Code Refactoring Sequence Diagram

Initially, the user highlights/selects a code block and requests the agent to refactor the code through the UI, which sends that request to the Chat Orchestration module. In comparison from the conceptual architecture report, Code refactoring's sequence diagram is slightly different then what was portrayed, as after inspecting Understand the conclusion was made that the LLM integration module would not return the suggestions to the Chat Orchestration module, instead forwarding the refactored code into the AI Code Application Service module, which displays the refactored code in-line for the developer to confirm or deny to code. Once approved, that message is sent back to the AI Application Service, which applies the validated refactored code into the Editor Core, then confirms the updates in the UI to the developer.

Discussion and Observations

In the process of recovering Void's concrete architecture, we learned about the strengths and limitations of translating conceptual architecture to the actual concrete architecture. We found some similarities in the high-level layered architecture, where the UI Layer, Chat Orchestration, and LLM service are similar, this shows that the modularity was preserved successfully by the developers. The other interesting finding was how closely the subsystem boundaries from the conceptual model are reflected in code, despite the integration complexity of VS Code and Electron.

The Layered Architecture style and Client-Server architecture style, shows the effectiveness for maintaining the application, especially maintaining clear interfaces between the VS Code environment and Void's AI features. Analyzing the Chat Orchestration subsystem in detail, showed how the design principles are applied in practice. The separation between the message handling, context building, LLM requests, and result formatting supports scalability and ease of maintenance. Upon examining the code dependencies, we found some components were intertwined. Because of this tight coupling, it makes it a little harder and more complicated to modify or extend one part of a subsystem. This means that the developers should do regular checks to verify that the code still follows the intended architecture, as the architectural drift can increase over time, and would make the system harder to maintain.

By doing this project, we understood the importance of the reflection analysis and it helped us understand how the actual system matches with the design that was originally planned, which we discussed in the A1 report. It allowed us to identify where the actual structure aligns and it drifts from the actual dependencies, also it gives detailed information about how to make architectural improvements for future. All in all, it can be said that the reflection analysis was really helpful, as it showed exactly where the real code structure matches from the planned conceptual architecture, and tools like Understand made this process easy by giving the simple visual dependency diagrams, and mapping back to the architecture.

Conclusion

The report on the concrete architecture gave a strong understanding of Void's built architecture. Also, it highlighted Void's integration with VS Code and Electron. In this report we have shown the detailed concrete architecture, and the similarities and differences from the conceptual architecture that we had in A1 report. We derived the dependencies based on the actual source code folders using the Understand Tool, through this dependency extraction, subsystem, and reflection analysis we learned about how real-world implementation diverges for functional and performance reasons.

By comparing the conceptual architecture from A1 and concrete architecture, we found that Void's high level concrete architecture was similar to the conceptual architecture, especially the User interface, Orchestration and LLM integration layer. However, there were some dependencies, which were tightly connected across the subsystems because it must work with VS Code's architecture.

Upon comparing the conceptual and concrete architectures, it can be said that most conceptual components had matching with the concrete architecture components. Overall, this report shows the importance of maintaining the balance between the design and the actual implementation. This concrete architecture of Void not only confirms the soundness of its conceptual architecture design but it also shows how the practical implementation reshapes the theoretical design. This report gave a great understanding about the relations, and how the developers and architects can manage the complexity and the overall system.

AI Collaboration Report

AI Member Profile:

For this project, our team decided to cooperate with ChatGPT as our AI collaborator. The key reason behind this was its profound capability in explaining and reasoning about concrete architecture of the systems. Different from some other models we tried like Claude and Gemini, ChatGPT gave crystal-clear and elaborate guidance on how to derive subcomponents from the overall structure of the system and how they interlink together. As our project, and specifically this deliverable, was focused on concrete architecture of Void and building its dependency diagram using the Understand application, we needed an AI capable of assisting in uncovering complex relationships of architecture and their translation into some ordered structure based on components. From comparing different tools, we found that in ChatGPT, outputs were consistently more accurate, contextually rich, and understandable and it was easy to implement them in the Understand application to map the files. After some initial sessions evaluating this tool, we decided to go ahead for the rest of the project with ChatGPT.

Tasks Assigned to AI Teammate:

1. To clarify subsystem roles and to review how to use Understand:

We used ChatGPT in concert with the Understand platform from SciTools to analyze Void's codebase and architecture. This helped us make sense of and simplify complex technical details extracted from Understand, as well as information found in the GitHub repository and overview pages for Void. Since most of these explanations contained dense terminology and unknown architectural terms, having ChatGPT made it much easier to understand and connect those terms to specific subsystems which were mapped in the Understand application. It also helped us in describing each subsystem, while performing all the requirements of the assignment. By combining the visualizations and dependency graph from Understand with the explanations provided by ChatGPT, our team had a clear view of how the subsystems interact with and depend on one another, which was crucial in the construction of accurate dependency and sequence diagrams.

2. 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.

3. Supplementary Research on Concrete Architecture:

Each of the team members interacted with ChatGPT GPT-5 independently, according to their responsibility. The team lead mostly utilized it to organize and distribute tasks to other members, while members utilized it to clarify subcomponents and key architectural concepts regarding Void. Using the explanation given by ChatGPT as our foundation, we explored the architecture overview and GitHub repository in greater detail to complete our objectives. At a high level, our approach was an iterative prompting strategy: based on an initial query, analyze the response, and refine questions to uncover how various subsystems connect and interact. ChatGPT also helped us get a clear idea of files and folders mapping in the Understand application and how to arrange the subsystems to get an accurate architecture analysis and diagrams. By doing this, we knew more about each subsystem and its role inside of the big system. Through this process, we knew every subsystem piece by piece and its role inside the big system. Throughout, we applied our judgment to what the AI was telling us and further made sure that the information remained accurate and contextually relevant.

4. Validation and Quality Control Procedures:

The team members reviewed and verified everything generated by ChatGPT before including it in the report. Every time, we cross-referenced the AI's information with the Void IDE GitHub repository and the results we got through Understand graph services. Whenever ChatGPT provided architectural explanations, we always cross-checked them against class notes or credible references available on the internet. No AI-generated material was used without thorough review for contextual relevance and factual accuracy.

5. Quantitative Contribution to Final Deliverable:

The AI contributed approximately 15% of the overall effort toward the final deliverable. Its main contribution was to help clarify how each subsystem and subcomponent interact as we derived that from Understand, and also to understand the mapping done by Understand itself. This was very important in the development of both the sequence and dependency diagrams, as well as in deepening our overall understanding of Void's concrete architecture. Establishing a common understanding of the structure of the system through AI really supported the success of the group in this task. Furthermore, the capability to organize and divide work efficiently using AI-generated prompts minimized wasting time on repetitive activities, hence allowing more time for analysis and report writing, therefore enhancing the quality of our final submission. While all major writing, analysis, and diagram creation were completed solely by the team, the assistance of ChatGPT did improve the overall quality and coherence of our work noticeably.

6. Reflection on Human-AI Team Dynamics:

Working with the AI teammate had a very supportive, constructive influence on our project. ChatGPT contributed to streamlining the workflow and deepened our understanding of the concrete architecture of Void and to understand, but most importantly, it got us to know more about how things work on the Understand platform and how to use it to meet the requirements of the assignment. This saved time in the planning process because we could focus on our core tasks. It also helped with brainstorming by explaining functions of subsystems and how everything connected. At times, information given by AI had to be checked against the official Void sources and also documentation related to the Understand in SciTools, but that was worth it, taking into consideration the clarity and productivity it provided. Generally, the experience showed that what is important when using AI is human judgment, underlining that AI works well as a supportive tool during research, but not instead of human analysis and decision-making.