A3: Architectural Enhancement

CISC 322

December 1st, 2025

Authors:

Matt Dobaj, Shaun Thomas, Babinson Batala,

Esmé Mirchandani, Sandy Mourad, Nihal Kodukula

0 - Table of Contents

1 - Abstract

This report extends our earlier analysis of Void by proposing an architectural enhancement that adds automatic context generation to its AI workflow. In our previous report we identified that Void's current design lacks a mechanism for automatically retrieving relevant workspace information before sending requests to the model. To address this, we investigated two architectural approaches, search-based retrieval and persistent local RAG, and evaluated them using the SEI SAAM method. Our analysis compared their impact on key non-functional requirements such as accuracy, maintainability, scalability, and response time. Based on these findings, we recommend the search-based approach, as it integrates cleanly with Void's existing layered architecture while improving the quality of its AI-assisted development features.

2 - Introduction and Overview

Artificial intelligence is transforming how developers interact with their code. Features such as inline completions, natural language prompting, and automated refactoring are reshaping

the developer experience. Void, a fork of Visual Studio Code, reflects this change by introducing AI-driven capabilities while maintaining VS Code's modular and lightweight structure. However, Void currently lacks an automated way to gather relevant context from a user's workspace to use in its code generation. Without this, its AI models respond without knowledge of related files or project dependencies, which can lead to duplicated code or a lack of accuracy in its results.

In our previous reports, we examined Void's conceptual and concrete architectures, identifying major components like the Void UI, VS Code Editor, Common, and Platform modules. We found that the Common component acts as a bridge between the user interface and backend layers, much like the base component in VS Code. During our A2 analysis, we also discovered that Void's context-handling logic existed in the source code but was commented out, leaving the AI without automatic access to related workspace information. This limitation stands in contrast to competitors such as Cursor and Claude, both of which have implemented their own forms of automatic context generation. To address this, we propose an enhancement that integrates automatic context generation into Void's architecture to enable more intelligent LLM responses.

Currently, Void's AI functions depend either on direct prompts from the user with a small amount of manually selected context, or automatic inline completion prompts with no context. Automated context generation improves on this by adding a retrieval step before model generation. When a user sends a request, the system first searches the workspace for relevant code files or documentation. It then combines the retrieved material with the user's query to form a more well-rounded prompt. The model can then produce an output that is far more accurate and relevant to the user's specific project.

This report will explain how RAG can be introduced into Void's system, describing which components require modification, what interfaces must adapt, and what new dependencies are introduced. It will also examine the performance, maintainability, and scalability impacts of this enhancement. Finally, we will evaluate two different implementation options, search-based retrieval and persistent local RAG, using the SEI SAAM method to assess their impact on both key stakeholders and non-functional requirements.

In modern AI assisted code editors, different tools use different strategies to gather relevant information from the codebase before sending a prompt to the model. Search based retrieval, used by editors like Claude, relies on scanning the codebase for exact or keyword matches using grep or similar tools [1]. Retrieval Augmented Generation, used by tools like Cursor, improves on this by using semantic or vector based search to find code and documentation that are conceptually related to the user's request. Instead of relying on exact keywords, it looks for similar looking words [1].

Architecturally, this enhancement builds on the patterns Void already uses. The Layered structure separates presentation, service, and data logic, making it easy to add the context functionality without breaking existing components or their connections between them.

In summary, this enhancement makes Void more intelligent, efficient, and user-friendly by automatically retrieving relevant project information before each AI-generated response. The

proposed automatic context generation naturally extends Void's existing architecture and reflects the direction of most modern AI-assisted development environments. With this improvement, Void can deliver more accurate assistance while keeping good maintainability, evolvability, and accuracy.

3 - Derivation Process

We created this idea after recognizing that Void had a context generation feature implemented that was commented out and unused. We recognized this as a core feature that was missing from Void in comparison to its main competitors Cursor and Claude. We looked at how context generation was implemented in the two main competitors' products and realized that they both had fundamental differences, with Cursor making use of RAG and Claude using lexical search. We decided to compare these two main implementations of context generation to see which would suit the Void project best.

4 - Architecture

Here is the concrete architecture we derived in our previous report:

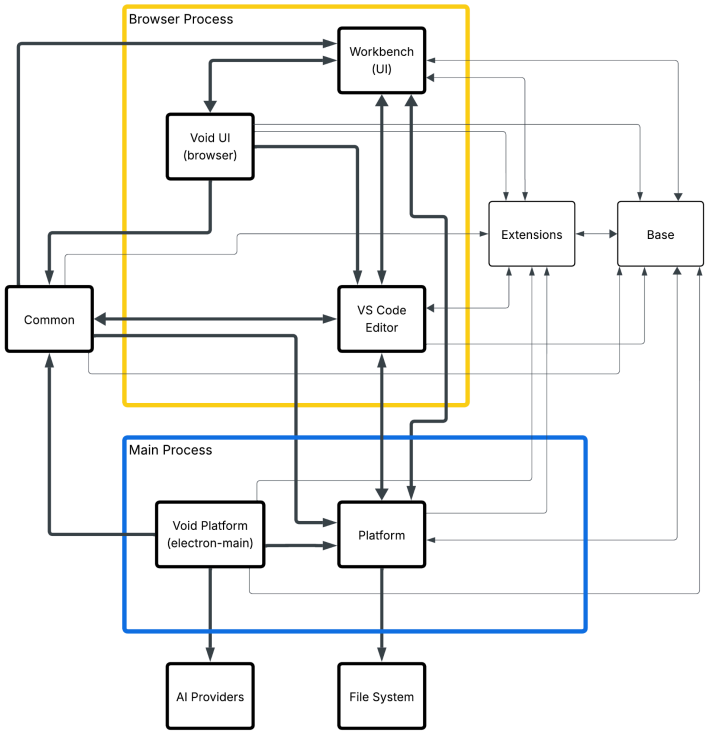

Figure 1: A diagram summarizing Void's concrete architecture. Each arrow represents a dependency, with double sided arrows showing components that are dependent on each other. Bold arrows are key dependencies and thin arrows are relatively minor code dependencies. The components in the yellow area run on Electron's browser process while components in the blue area run in Electron's main process. Components outside of this divide can be run on either process.

As previously mentioned we discovered that the context generation in Void is limited and manual, with the user having to manually select different files to be included in their prompt context window. We have decided to fix this problem by proposing automatic context generation to Void's prompt process. The priority will be on Void's chat functionality, but we will also consider extending this to the autocomplete service. Tools required for the LLM to read and edit files are already implemented via the Model Context Protocol (MCP) in the Void platform.

Automatic context generation provides several benefits for Void's architecture and overall user experience. By gathering relevant information from the workspace before prompting the LLM, Void can produce responses that better reflect the user's actual project structure instead of relying only on the active file. This reduces incorrect suggestions and makes the AI's behaviour more consistent across files.

4.1 Possible Implementations

The first implementation we have decided on for our proposed automatic context generation feature is on demand search of the code base. This implementation will search through the user's codebase on every request as needed by the LLM provider using a search subagent. This can be done using grep or similar lexical search to find matching keywords in the code. Matching segments will be appended to the prompt as context. This is the technique that the Claude code editor makes use of for its context generation. [1] The logic for this feature would reside in the Void Platform component and will not require architectural changes. Files will be read via events through the VS Code Platform and results will be added to the prompt in the Void Platform component when interacting with the LLM. Because Void already has an MCP server implemented, further extensions will also be relatively straightforward.

At the moment, Void formats its prompts for the LLM in the Void UI component with the file *convertToLLMMessageService.ts*. Because using lexical search to find context in the codebase relies on the LLM to generate the required regular expressions, this file will need to be moved from the Void UI Component to the Void Platform component. This is so that additional commands can be given to the LLM before adding context to the user's prompt. This file can likely be moved to the Void Platform without many significant modifications. This would also mean that the *sendLLMMessageService.ts* file in the Common component and the *sendLLMMessageChannel.ts* file in the Void Platform would need to be modified to accommodate this move in logic. A new file would be added to the Void Platform, which we will call *gatherContext.ts*, which will prompt the LLM to generate a search pattern to use on the codebase to retrieve context. This file will use the now-moved *convertToLLMMessageService.ts* file and the files in the users codebase using grep to search for keywords and read the files via the VS Code platform. From what we can see, no other significant changes would need to be made to accommodate this change.

The second implementation will focus on creating a local persistent RAG, which stores context about the user's code space in a persistent file. Each function or component in the user's codebase will be represented in a vector database which will store the function name and a brief description about it. This local file will need to be updated with changes to the code base and will be searched to provide extra context. File updates can be handled thanks to Void's Publish-Subscribe architecture, with an event handler set up to listen to any file changes. This is most similar to how Cursor implements their context generation feature [1]. This will require a

local database to be maintained by Void, the responsibilities of which may need to be abstracted out of the Void platform component into its own Context component. This component will be dependent on the VS Code platform to read files and the Void platform will be dependent on it to retrieve information from the database.

While this implementation does not make additional use of the LLM functionality, in order to add context to the prompt in the Void Platform, the *convertToLLMMessageService.ts* should still be moved to the Void Platform. This would also mean that the *sendLLMMessageService.ts* file in the Common component and the *sendLLMMessageChannel.ts* file in the Void Platform would need to be modified, like with the previous implementation. As previously mentioned, this implementation would also require multiple additional files, likely enough to warrant its own component, which we have called the Context component. This component will have multiple files such as *vectorizeCodebase.ts* to organize the codebase into a format that can be stored in the database and a *database.ts* file, which will handle the database's lifecycle and track updates. The *vectorizeCodebase.ts* can make use of a more specific *vectorizeFile.ts* for more specificity. An additional event handler will need to be registered with the VS Code platform in order to track updates to the files in the codebase. When an event is triggered, the *vectorizeFile.ts* file can update the database. When context needs to be added to the prompt, a search can be performed on the database based on keywords in the prompt.

Here is what an updated concrete architecture would look like with the added context component:

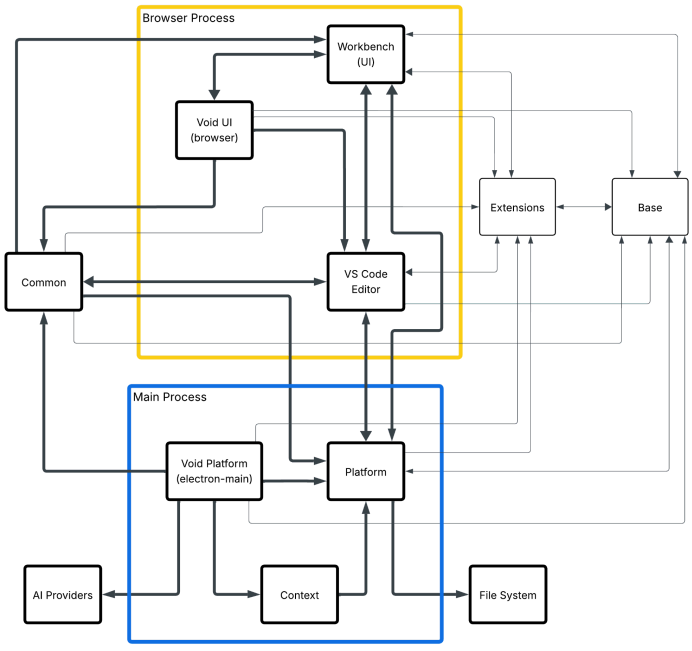

Figure 2: A diagram summarizing Void's updated concrete architecture with an added context component. Each arrow represents a dependency, with double sided arrows showing components that are dependent on each other. Bold arrows are key dependencies and thin arrows are relatively minor code dependencies. The components in the yellow area run on Electron's browser process while components in the blue area run in Electron's main process. Components outside of this divide can be run on either process.

4.2 Stakeholders

The primary stakeholders for this enhancement are Void's maintainers and End Users, with each group being affected in a different way. For Void developers, introducing automatic context generation changes how prompts are built inside the Platform and Void UI components. They need to maintain the new context retrieval logic, update *convertToLLMMessageService* after it is moved, and ensure that any new files such as *gatherContext.ts* or the vectorization and database utilities in the RAG approach fit cleanly into the existing IPC and event handling structure. To ensure extendability and maintainability, the architecture should stay easy to understand and easy to change. Added logic must be organized in a way that does not create unnecessary complexity or confusion.

Users experience the enhancement from a practical perspective through more accurate and helpful AI interactions. Automatic context generation removes the need to manually select supporting files and allows Void to automatically gather the information the LLM needs. This improves the quality of chat responses, autocomplete suggestions, and code edits, especially when working in larger or unfamiliar projects. Users value fast responses, relevant suggestions, and a workflow that feels smooth and intuitive. While developers are focused on the clarity and maintainability of the software architecture, users benefit from more intelligent assistance that can reflect the structure of their actual project.

4.3 Non Functional Requirements

When comparing the two architectural approaches for automatic context generation, several non-functional requirements were found that affect either Users or Void developers. These requirements help reveal how each design behaves under real conditions and which option brings the most overall value. The key NFRs include response time, accuracy, evolvability, testability, maintainability, and scalability.

Response time is especially important for Users because delays directly affect the functionality of chat responses and the accuracy of inline autocomplete in the context of the user's recent changes. The search-based approach becomes slower as the codebase grows, since it must scan all files each time the LLM requests context. RAG performs better here because the lookup is constant relative to codebase size and is limited to a small vector database. Overall, RAG provides better response time for large projects, which benefit users.

Accuracy is another major concern for Users, since they depend on the AI to produce correct code suggestions and avoid missing important information. Search has a natural advantage because it looks for exact keyword matches across the entire codebase. This avoids cases where vector similarity might return related strings that do not match what the developer actually needs, which is a known issue for vector search models [1]. Due to this, the search approach provides more reliable accuracy for users to provide better LLM responses.

Evolvability matters primarily to Void developers, who maintain the Platform and Common components and will be responsible for future improvements and extensions. RAG has limited evolvability because once the database and embedding strategy are chosen, there are few ways to extend or refine the system without switching the way data is embedded. Search, on the other hand, can grow naturally with the addition of new sub-agents or more advanced lexical

techniques. For example, developers could later add a testing agent that verifies the LLM's generated code compiles correctly. This makes the search design easier to evolve for both Void's developers and future user expectations.

Testability is also more relevant to Void developers. Search relies on the LLM to generate search patterns, which introduces non-deterministic behaviour and makes it harder to test conditions consistently. RAG does not depend on the LLM for retrieval, so developers can test retrieval quality using fixed datasets and compare expected matches directly. This makes RAG the more testable approach.

Maintainability is important to both groups but particularly to Void's developers, since RAG introduces extra files, a local vector store, database logic, and must stay in sync with all file changes. This creates additional long-term maintenance work. Search requires fewer new components. Once the *gatherContext* file is complete, maintenance is limited to keeping the search agent functioning correctly. Since its logic is simpler and does not introduce a persistent data component, search is easier to maintain.

Scalability affects both Users and Developers. Search becomes slower as the number of files increases, and it may return a large amount of context when many matches appear. RAG scales better for retrieval because results are restricted to a fixed number of closest matches, which keeps context size smaller and more consistent. Smaller context windows are important for accuracy, cost, and responsiveness [2]. Although the initial database initialization for RAG may be slower as the code size increases, it remains fast when incrementally updating the database with user changes. Query-time performance remains fast, meaning RAG is better in terms of scalability.

4.4 Risks

There are also risks to consider. RAG introduces more architectural complexity and increases the chance of outdated embeddings if file events are missed. It also creates data management concerns because the vector store must remain consistent with the user's project. Search avoids these risks but may become slower and less accurate for large projects, which may frustrate users during chat or autocomplete. It also relies heavily on the LLM to generate effective search patterns, which may result in inconsistent retrieval quality.

After weighing the NFRs and considering the stakeholders, we determined that the search-based method is the better choice for Void. While RAG offers advantages in response time and scalability, search performs better in accuracy, evolvability, and maintainability. These qualities matter most for Void because the feature should integrate well with the current architecture, remain sustainable for the open source contributors to maintain, and provide users with accurate and useful results. In the end, the search based approach is best when considering both Void's contributors and users' key non functional requirements, making it the approach that best moves Void forward.

4.5 Plans for Testing

To ensure that our proposed automatic context-generation feature integrates smoothly with Void's existing AI workflow, we will test its behavior across all possibly impacted features. For manual prompting, we will verify that user-selected context is still respected and that prompt formatting in *convertToLLMMessageService* retains its functionality after moving and extending it to allow for additional context. For inline completions, we will measure response time and verify that suggestions continue to appear as expected. This can be done using a mock code base that will replicate an end user's use case. The tests could then be automated using this mock code base to verify that a response from an LLM answers the prompt given to it, compiles properly, and makes use of external utilities. The Void developers can also validate that the new IPC messages introduced for context retrieval do not interfere with existing Browser-Main interactions. Finally, we will test both defined use cases end-to-end to ensure that a user's message or autocomplete event flows correctly through the Void UI, Common, and Void Platform layers without impacting existing functionality .

5 - External Interfaces

Void continues to exchange information through several key interfaces that connect its user-facing components, AI backends, and extension frameworks. These interfaces define how data flows into and out of the system, ensuring consistent communication between the browser and main processes while maintaining modularity and platform independence.

5.1 Graphical User Interfaces (GUIs)

Information sent to the Platforms:

- User prompts, inline completions messages and the selected model
- File requests, system commands

Information received from the Platforms:

- Finalized AI responses (code completions / changes)
- Contextual updates from the VS Code Platform

5.2 AI Provider Interfaces

Information sent to AI providers:

- Structured prompts including user input, message history, code context, and metadata
- Model configuration parameters

Information received from providers

- Model responses code edits, service requests

Automatic Context Generation Usage:

- Structured prompts that include user input and message history to generate search patterns

5.3 File and Platform Interfaces

Information Transmitted

- File and workspace operations (open, save, rename, delete) from the VS Code Platform

Automatic Context Generation Usage:

- Search files via command line tools
- Reading codespace files via the VS Code Platform

6 - Use Cases

We have kept our same Use Cases but will reconsider them through the lens of our new automatic context generation feature. The lines are labelled to represent the file being referenced rather than the specific method called.

6.1 Use Case 1

Use Case 1 investigates how Void handles inline code completion. We explore the User typing in the editor and how Void automatically suggests an inline code completion with our new enhancement.

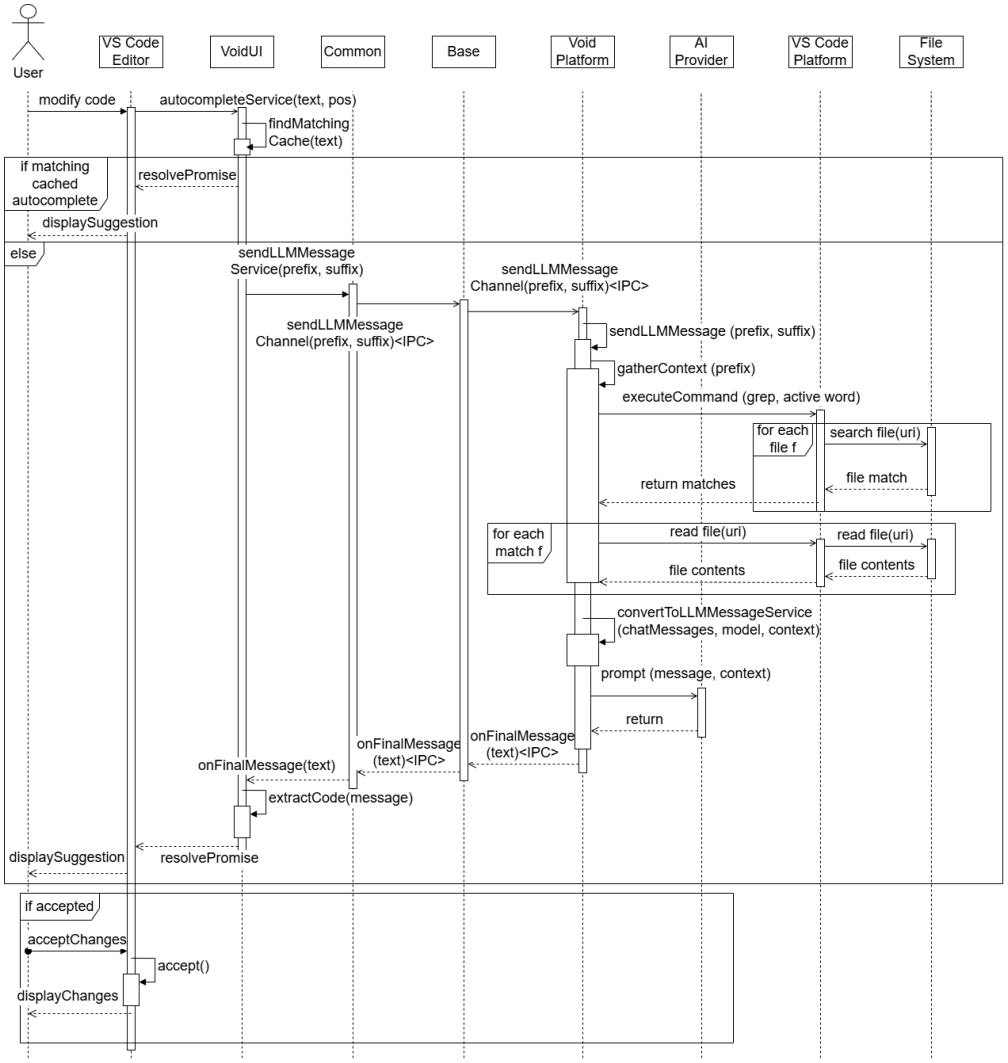

Figure 3: The updated sequence diagram for Use Case 1, code autocomplete, based on Void's concrete architecture with our proposed changes. The lines are labelled to represent the file being referenced rather than the specific method called.

As per the current version of Void, when the user types in the code editor, VS Code sends an event to all registered inlineSuggestion handlers. In the Void UI component, a handler is registered to this event and is triggered by the user's input. Before prompting the LLM, the UI component checks to ensure that a previous autocomplete that matches the text in the user's editor already exists. If so, then this result is displayed to the user.

Next, the UI component then uses the *sendLLMMessage* service to send the message via IPC through the base component to the Void Platform component, where the LLM is communicated with. The *sendLLMMessage* on the Void Platform then calls the gatherContext function. Here, rather than prompt the LLM for a search pattern, we have decided to show Void searching only for the active word that the user is looking at. This was done because response time is crucial for inline code completion and prompting the LLM twice may affect the response time significantly. All matching files are then read by the Void Platform and bundled with the

user's active file in the *convertToLLMMessageService*, which we have moved from the Void UI to the Void Platform. Void then prompts the LLM and returns the resultant message.

Like before, after receiving the LLM response the Void UI component extracts the code and displays the inline suggestion. If the user accepts this suggestion, the editor adds the text to the open window.

6.2 Use Case 2

Use Case 2 investigates how Void handles requests through its LLM chat window. We explore the User sending a message and how Void responds. It has been updated to include the context generation flow that is based on searching the codebase for keywords.

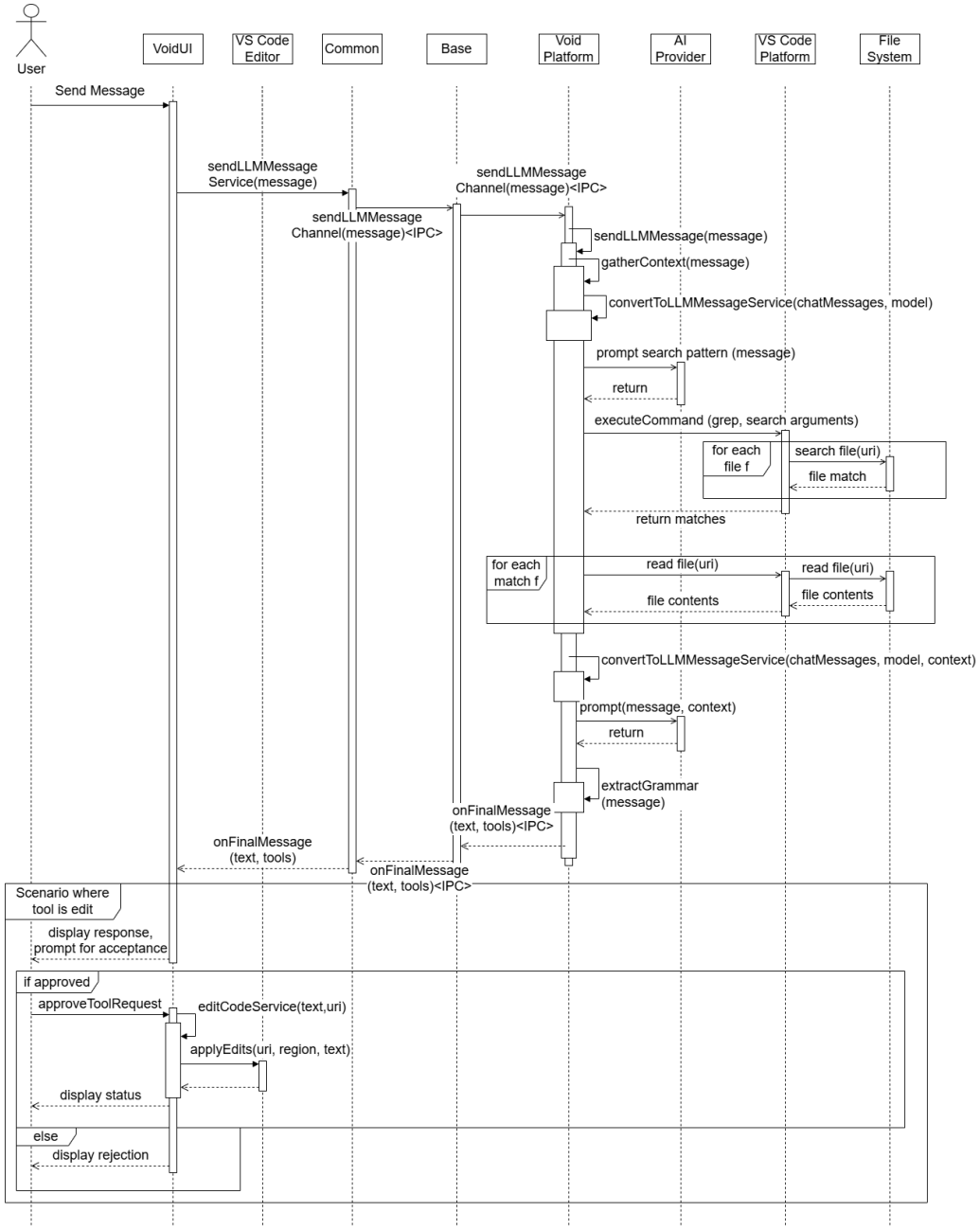

Figure 4: The updated sequence diagram for Use Case 2, chat based interactions in Void, based on Void's concrete architecture with our proposed changes. The lines are labelled to represent the file being referenced rather than the specific method called.

When the user sends a message in Void's chat sidebar in the UI component, it sends the message via the *sendLLMMessageService*.ts file in the Common component to the Void Platform via IPC through the Base component.

Our addition to Void will be inserted here. Before prompting the LLM to generate code, the Void Platform will first request the LLM to generate a search pattern to find relevant context in the codebase using the *gatherContext* file. A grep command will then be run with this search pattern and all matching files will be appended to the user's prompt when formatting the message for the LLM in the moved *convertToLLMMessageService* file. Next, the Void Platform requests the LLM to generate code. It then calls the *extractGrammar* file to extract the code content before calling the *onFinalMessage* function via IPC to update the chat window.

Like before, if the scenario is an edit to a file, the UI will ask the user if they approve of the changes. If true, then the text will be given to the VS Code Editor component, which will tentatively apply the edits to the file. It does not directly write to the file, instead waiting for the user to save the changes, matching its behaviour with the previous use case.

7 - Data Dictionary

AI Provider: Third-party model (e.g., OpenAI, Anthropic) that processes user prompts and returns code completions or code suggestions.

Browser Process: Client-side Electron process that runs Void's user interface and captures user interactions.

Electron Framework: Cross-platform system for building desktop apps using web technologies; used by both VS Code and Void.

Inter-Process Communication (IPC): A protocol used to pass information between multiple processes.

Large Language Model (LLM): Model that generates human-like text or code from natural language input.

Main Process: Backend Electron process controlling app lifecycle, system logic, and external communication.

Model Context Protocol (MCP): A standard for connecting AI applications with external systems.

Non Functional Requirement (NFR): A stakeholder requirement that defines how a system should perform its functionality (e.g. maintainability, response time, scalability).

Retrieval Augmented Generation (RAG): An AI technique that improves LLM model accuracy by retrieving related information from a data source (e.g., the workspace) and appending it to the user's prompt before generation.

Source Code Management (SCM): A methodology to organize, track, and log changes to source code over time (i.e. Git).

Prompt: The text or code query written by the user and sent to an AI model for code completion or code suggestions.

8 - Naming Conventions

For this report we kept naming simple and easy to follow. Whenever we mention a function or file we use the exact name from Void's actual code so readers can easily find it in the source if they want to. As with our previous reports, we followed the same general structure that Visual Studio Code uses for its own files and components, keeping things consistent with their standards. When we discussed potential features for our enhancement such as the Context component and files like *gatherContext.ts* or *vectorizeCodebase.ts*, we used the same naming patterns already seen in the codebase. Functions and variables use lowerCamelCase, while components and modules use PascalCase. File names were based on the action they performed, matching Void's style. Overall, our goal was to stay consistent with Void's existing style so that any developer reading this report could easily understand where everything fits and how the new parts connect to the rest of the system.

9 - Conclusion

This report expanded our architectural work by examining how Void's existing subsystems support AI-driven features and where the structure allows opportunities for improvement. By tracing how information moves through the Browser and Main processes, and analyzing the responsibilities of the relevant components in our previous report, we gained a clearer understanding of how Void handles prompting, file access, and model communication in practice. This deeper view of the concrete architecture helped us identify architectural constraints, uncover unused or unfinished functionality, and evaluate where a new enhancement could be introduced without disrupting the system's modular design.

We focused on the absence of automatic context generation, an important capability found in competitors like Cursor and Claude. By proposing two architectural approaches, search-based retrieval and persistent local RAG, we evaluated how each option interacts with Void's existing components and communication patterns. Our SAAM analysis compared their impacts on users and developers across response time, accuracy, maintainability, testability, evolvability, and scalability.

Based on this evaluation, we determined that the search-based approach is the most suitable enhancement for Void. It integrates cleanly with the current architecture, preserves maintainability, and provides reliable accuracy without adding unnecessary complexity. Implementing automatic context generation in this way strengthens Void's AI capabilities and moves the system toward a more complete, context-aware development experience that remains easily maintainable by the project's contributors.

10 - Lessons Learned

During this phase of the project, we learned how much more challenging it is to design an architectural enhancement than to simply document an existing system. Proposing automatic context generation forced us to understand Void's concrete architecture at a deeper level, especially how the Platform, Common, and UI layers communicate through IPC. We also realized that even small changes, such as moving a file like *convertToLLMMessageService*, can affect multiple components and require careful thought.

Another key lesson came from comparing architectural alternatives using the SAAM method. This made us think critically about the non-functional requirements and putting NFRs into the context of a real project that we have begun to understand well helped us understand them better. It was clear that a design that looks powerful on paper, like RAG, can introduce more complexity than the system actually needs. SAAM helped us structure our evaluation and gave us a clearer understanding of tradeoffs rather than focusing only on technical possibilities. It also helped us base our decision on which implementation was better for the project through requirements rather than feeling.

Finally, we learned the importance of collaboration and consistent communication when working on a system this large. Dividing the work across use cases, diagrams, interfaces, and the SAAM analysis made the process smoother and helped us avoid repeating earlier mistakes from A2. This assignment required less sequential work where one section had to be done before all the others could be started, which allowed for easier delegation. As a group, we gained a better sense of how architectural decisions affect a project's codebase and how structured analysis tools and diagrams can guide those decisions. This assignment strengthened our understanding of real-world architectural reasoning and how enhancements need to be decided on by looking through the project's requirements in an objective manner.

11 - References

[1] "Why Cursor Is About To Ditch Vector Search" [Online]. Available: https://www.tigerdata.com/blog/why-cursor-is-about-to-ditch-vector-search-and-you-should-too

[2] "The Claude 3 Model Family: Opus, Sonnet, Haiku" [Online]. Available: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf

12 - AI Report

For this deliverable, our team used ChatGPT (GPT 5.1, November 2025 version) as a virtual AI teammate to support our A3 enhancement proposal and SAAM comparison. This deliverable required us to propose an enhancement, explain the high-level architecture changes, and compare two ways to implement it using SEI SAAM.

Unlike A2, where we mainly used ChatGPT for code understanding, in A3 we used it more for structured planning and writing support. Our enhancement was automatic context generation in Void, and we needed to compare two options: search-based retrieval and persistent local RAG.

We assigned ChatGPT clearly defined support tasks that focused on helping us organize ideas and check our reasoning. These tasks included:

1. The SAAM definition: We reached out to ChatGPT to help explain the steps required for the SAAM process based on the description given to us in the assignment.
2. SAAM scenario drafting: Creating possible change scenarios for our enhancement (ex: “handle very large projects”, “add a new model provider”, “add a new file type”) so we could compare how each architecture handles change.
3. Documentation support: Helping rephrase technical content for consistency.

We kept our AI content separate so we could track what came from AI and verify it. We wanted to make sure we carefully used and reported the AI’s contribution clearly. We also asked ChatGPT to respond in simple formats like bullet lists, direct vs indirect change notes, and short SAAM scenario descriptions, since those were easy for us to review and verify. To ensure the accuracy of AI generated content, we followed strict steps:

1. Cross checking: We checked AI outputs against our own architecture notes, various web sources, and the design we described in the report. For example, when we asked ChatGPT to confirm our understanding of a component or scenario, we used its answer as a starting point for group discussion instead of treating it as a final decision.
2. Peer review: At least two team members reviewed AI generated content to ensure correctness and clarity. We reviewed and discussed each AI explanation as a group before including it in the report.
3. Revision and refinement: When the AI output was vague, we rewrote the prompt to make it a single, concrete change. If we realized that it was limited in its capabilities on the subject we would discard the output and look for information elsewhere.

ChatGPT’s main value in A3 was speed and structure. It helped us generate scenario options quickly, and it helped us keep our SAAM comparison organized. But all final decisions were made by the group, especially our final choice that the search-based approach fits Void better because it integrates cleanly and avoids extra long-term complexity for Void’s project maintainers.

We estimate that ChatGPT contributed about 15% overall. This includes scenario drafting, NFR wording help, and editing for clarity. The remaining 85%, including the

enhancement choice, the architecture changes, the sequence diagrams, and the final SAAM conclusion, was completed manually by our team.

Overall, using ChatGPT in A3 helped our workflow, but it also created some extra work because we had to validate and tighten prompts. The biggest lesson we learned is that AI is useful when you give it clear constraints, especially for architecture tradeoffs where small wording changes can affect the conclusion. It is also oftentimes inaccurate, and AI output requires strict review and verification. Overall, its assistance made the drafting process of this report faster and smoother.

Example Usage: