Void AI - Concrete Architecture Report (CISC 322/326 A2)

Due: Nov 7th, 2025

Group 23: We'll Fix It in Post

Jaivik Joshi - jaivik.joshi@queensu.ca
Zack Wood - 22RQLJ@queensu.ca
Robin Houiler - 22BT35@queensu.ca
Charlie Kevill - 21cmk11@queensu.ca
Kai Maddocks - 25xx12@queensu.ca
Jared Simon - 22WGK@queensu.ca

1 Abstract

This report describes an enhancement to VoidEditor called the "Common & Suggested Questions" tab. The feature adds a panel that shows both frequently used prompts and suggestions created from the user's current code. New components in the extension host watch editor events, build a light context snapshot, and generate ranked prompts while staying careful about performance. The renderer shows these prompts in a suggestions panel where users can click and turn them into chat questions. We explain how this design affects performance, maintainability, evolvability, testability, privacy, and the main stakeholders such as Glass Devtools, open source contributors, and end users. Using two non-trivial use cases and four figures, we show how the enhancement fits into the high level and low level architecture. We then compare two ways to realize the feature, a simple shortcuts tab and a proactive context-aware assistant, using a SAAM-style analysis over key non-functional requirements.

2 Introduction and Motivation

Today we will be looking at the VoidEditor, and we will be looking for an enhancement that can help the VoidEditor's users in a quality of life improvement. This enhancement will also be meant to provide the VoidEditor some advantage over its most direct competitor which is Cursor, another IDE with integrated chatbot support.

An enhancement we could make could be a new tab at the top with common questions people ask and questions that they could ask based on the code that they are currently editing. This would mean that our code would ask common questions like "resume the opened folders", "change this code to" with a programming language that the user might want it to change to based on the files and code, or it could have a question like "write a function" with options based on what the current file or files might need a function for. This directly helps users who use our VoidEditor a lot and need to ask these common questions hundreds of times a day as this would lower the time they take writing the same questions.

Another scenario is if our users do not think of asking these questions that could help them. Now with this tab the common questions could help them, and the specialized questions generated based on the code of the open files would suggest to them solutions to problems they did not think of or forgot to address. Figure 1 shows where this new tab and its backing services sit in the existing runtime structure.

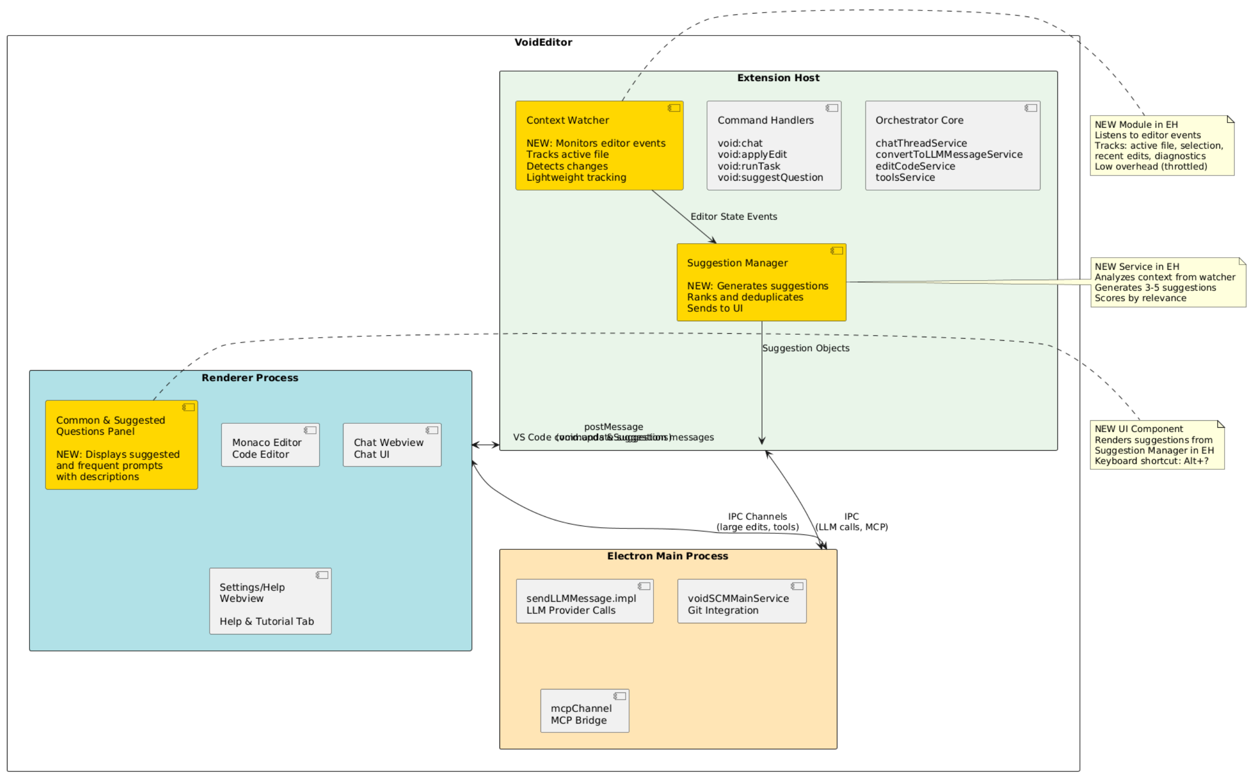

Figure 1: Modified Top-Level Concrete Architecture. The three processes remain unchanged, but the renderer gains the Common & Suggested Questions Panel and the extension host gains new context and suggestion services.

3 Stakeholders and Non-Functional Requirements

The major stakeholders of VoidEditor start with the core development team which is Glass Devtools. Their main non functional requirements consist of data privacy and security by protecting user data, performance and responsiveness as Void needs to work on large codebases and answer fast to the user requests, and maintainability and code quality as the code should be easily understood by anyone of the open source contributors who want to help improve Void and the whole software needs to be maintained so that users can use it.

Then we have the open source contributors, which is the community at large of VoidEditor. They also focus on the maintainability and code quality as well, then we also have documentation as every part of the Void that is added by these contributors needs to be properly documented to be allowed or blocked by the main development team quickly. Finally, the code also needs to be modular so that the contributors and the community at large can easily help by adding new modules.

The last major stakeholder we have is the end users who need security so that none of their projects and their code and their prompts are leaked to anyone. Performance also matters as the end users need VoidEditor to be able to analyze large amounts of files and respond effectively. Then finally, the end users need VoidEditor to be easily usable and familiar for them, which is why it should always stay in the overall format from Visual Studio Code. These stakeholders and non functional needs are later used in our SAAM comparison of alternatives.

4 Current Architecture and Need for Change

Our proposed enhancement, a dedicated "Common & Suggested Questions" tab which uses frequently asked prompts as well as automatically generated suggestions based on the user's active code, fits very well within Void's architecture of integrating AI assistance into the code editing workflow. However, introducing such a feature requires a careful look at how the current architecture supports, restricts, and will need to evolve to accommodate this enhancement.

In many ways, Void's architecture is a solid foundation for an IDE helper. Void's AI features already live mostly within renderer webviews, utilizing VS Code's environment. These webviews are designed to communicate with the rest of the system through VS Code commands. Because of this, the new questions panel can be added without changing the core editor. The orchestrator system, which exists in the extension host, makes this even easier as it already collects file context, summarizes the editor state, and manages communication with the language model. This means that much of the data needed to generate suggestions for the user such as open files, recent edits, and relevant code snippets are already accessible. If deeper analysis than that which is currently possible is needed, Void's optional MCP (Model Context Protocol) bridge can use external tools, allowing the editor to receive analysis or insights without using large amounts of logic directly inside the UI layer.

Despite these advantages, several barriers make implementation difficult. While renderer webviews are flexible, they do not automatically receive information about the editor's state. They also do not know which file is open, what the user has selected, or when the user switches between workspaces. As of right now, the system only knows this when the user triggers a command. A system which is meant to automatically generate suggestions needs a continuous, event driven method of tracking the state of the editor.

Also, while the orchestrator makes for optimal routing, it does not contain any subsystem which is capable of generating recommendations on its own. The architecture is designed around user initiated prompts, not for autonomous suggestions. Constantly analyzing workspace events also raises performance concerns. The editor must still be fast, especially with large projects, and any system that monitors file changes or diagnostics must do so without affecting the UI or overwhelming the backend.

These problems show why we need both high level conceptual changes and low level concrete changes, which we outline next.

5 Proposed Enhancement and Architectural Changes

Due to these issues, multiple high level architectural modifications are needed to implement this change. The most significant change is the introduction of a new suggestion manager within the extension host. This component would act as a lightweight recommendation system, continuously receiving signals from the editor such as changes in the active file, diagnostics updates, or patterns in recent user prompts, before turning them into candidate suggestions.

To support it, a small event driven context watcher module is needed to monitor VS Code's APIs and provide low cost summaries of what the user is doing. These two additions would allow the editor to form an up to date understanding of the workspace without needing to do expensive, full workspace rescans.

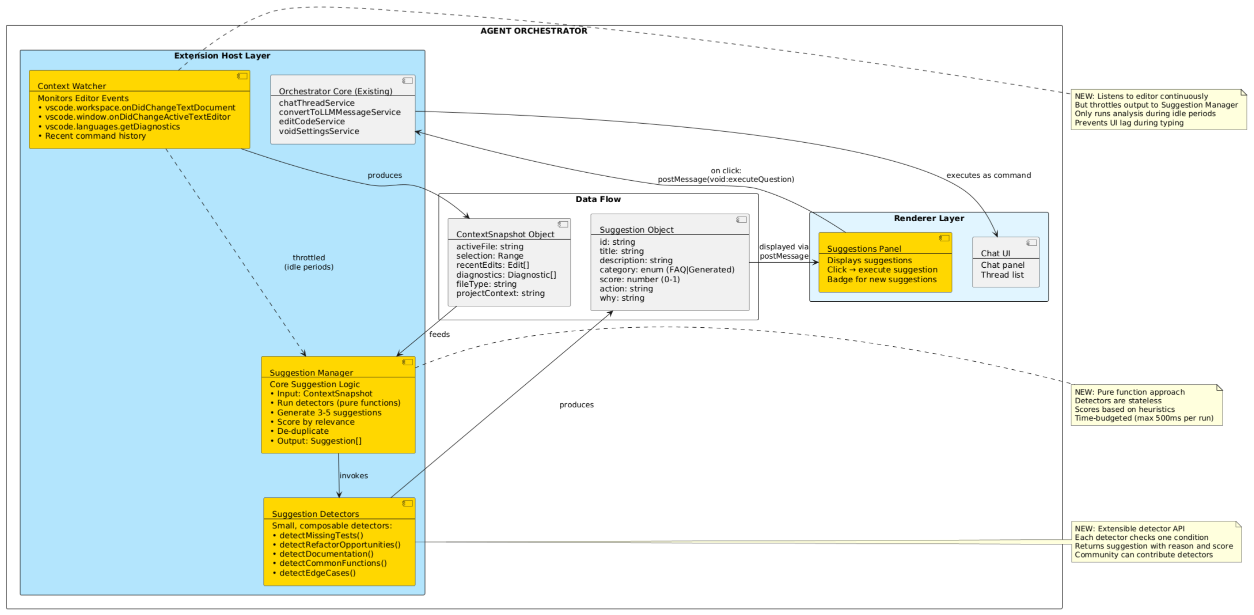
AGENT ORCHESTRATOR
Extension Host Layer
Context Watcher
Monitors Editor Events
• vscode.workspace.onDidChangeTextDocument
• vscode.window.onDidChangeActiveTextEditor
• vscode.languages.getDiagnostics
• Recent command history
Orchestrator Core (Existing)
chatThreadService
convertToLLMMessageService
editCodeService
voidSettingsService
produces
throttled
(idle periods)
Data Flow
ContextSnapshot Object
activeFile: string
selection: Range
recentEdits: Edit[]
diagnostics: Diagnostic[]
fileType: string
projectContext: string
Suggestion Object
id: string
title: string
description: string
category: enum (FAQ|Generated)
score: number (0-1)
action: string
why: string
on click:
postMessage(void:executeQuestion)
executes as command
displayed via
postMessage
Renderer Layer
Suggestions Panel
Displays suggestions
Click → execute suggestion
Badge for new suggestions
Chat UI
Chat panel
Thread list
feeds
Suggestion Manager
Core Suggestion Logic
• Input: ContextSnapshot
• Run detectors (pure functions)
• Generate 3-5 suggestions
• Score by relevance
• De-duplicate
• Output: Suggestion[]
produces
invokes
Suggestion Detectors
Small, composable detectors:
• detectMissingTests()
• detectRefactorOpportunities()
• detectDocumentation()
• detectCommonFunctions()
• detectEdgeCases()
NEW: Listens to editor continuously
But throttles output to Suggestion Manager
Only runs analysis during idle periods
Prevents UI lag during typing
NEW: Pure function approach
Detectors are stateless
Scores based on heuristics
Time-budgeted (max 500ms per run)
NEW: Extensible detector API
Each detector checks one condition
Returns suggestion with reason and score
Community can contribute detectors

Figure 2: Modified Agent Orchestrator Subsystem. The Context Watcher, Suggestion Manager, and Suggestion Detectors cooperate to turn editor events into suggestion objects that are sent to the renderer.

Figure 2 shows these additions in more detail. The Context Watcher feeds a `ContextSnapshot` object into the Suggestion Manager. The manager runs several Suggestion Detectors and produces `Suggestion` objects that are sent toward the renderer layer.

For the UI side, the renderer layer would need to be updated so that the new suggestions panel can receive messages from the newly implemented suggestion manager. This would mean extending the existing command based messaging channels and changing the webview so that it can show new suggestions as they arrive. While these changes operate at the surface level, they maintain the structure and workflow of Void's existing architecture which has the renderer handling presentation, the extension host handling logic, and communication occurring through VS Code's command pipeline.

These adjustments allow the new feature to fit cohesively with the rest of Void's architecture without feeling like its own feature outside of Void. The renderer's flexibility makes it possible to add the new UI changes without disrupting the editor. The orchestrator gets more functionality by holding the suggestion manager, but remains modular and maintainable so it does not use too many resources. The context watcher increases responsiveness by reacting only to incremental changes, in order to avoid performance problems.

Together, these modifications preserve the system's non functional requirements of performance, maintainability, modularity, and security while enabling Void to assist its users in real time rather than simply responding to prompts it is given. Conceptually the same three main subsystems stay in place, but the agent orchestrator subsystem in Figure 2 becomes richer and more aware of context.

Overall, this architectural analysis shows that while Void's current design provides a strong baseline for adding a suggestions tab, completing the enhancement requires a good amount of change to the orchestrator layer and the surrounding communication interfaces. If these adjustments are made, Void can offer a very useful feature to the user, one which reduces repetitive prompting, helps users discover actions they may not have thought to request, and makes Void better than other AI enhanced IDEs.

6 Use case Sequence Diagrams

6.1 Use Case 1: Auto-Generate and Display Suggestions

Figure 3 shows the first main use case.

The auto-suggestion feature activates when the user opens a project with multiple files and begins editing. As soon as the project loads, the renderer registers VS Code editor event listeners, and the Context Watcher starts monitoring events such as text document changes, active editor switches, and incoming diagnostics. While the user types, these events fire rapidly, so the Context Watcher buffers them and applies a one second throttle. Only after the user stops typing for at least one second does the system initiate a suggestion run. At that point, the Context Watcher generates a Context Snapshot containing the active file, the user's current selection, a record of the most recent edits, any diagnostics produced by the language server, the file type, and a sense of the broader project context.

The Suggestion Manager then uses this snapshot to run several detectors in parallel. Each detector independently analyzes the code and returns a potential suggestion. For example, one detector identifies missing test files and suggests writing unit tests, another notices a long function and recommends extracting logic into a helper, a third sees a public function without documentation and suggests adding JSDoc, while a fourth recognizes repeated parsing patterns and recommends creating a parsing helper. Each of these suggestions comes with a confidence score. The Suggestion Manager ranks them, removes duplicates or redundant entries, and keeps the top three.

Once the suggestions are ready, the Suggestion Manager sends them back to the renderer through a message. The renderer updates its internal state, and the Suggestions Panel displays the final three suggestions. Each suggestion includes a title, description, confidence percentage, and a short explanation of why it was generated. The panel also highlights new suggestions with a visible badge.

When the user clicks a suggestion such as "Write unit tests", the renderer converts that suggestion into a structured prompt and inserts it into the chat input with a "Suggested for you" label. The user can then review the prompt and send it to the LLM if they want. Throughout this entire flow, the system operates asynchronously. Analysis only runs during idle periods, never interfering with the user's typing or slowing down the editor. The components are transparent about their reasoning, and optional for the user, ensuring suggestions help without interrupting the workflow.

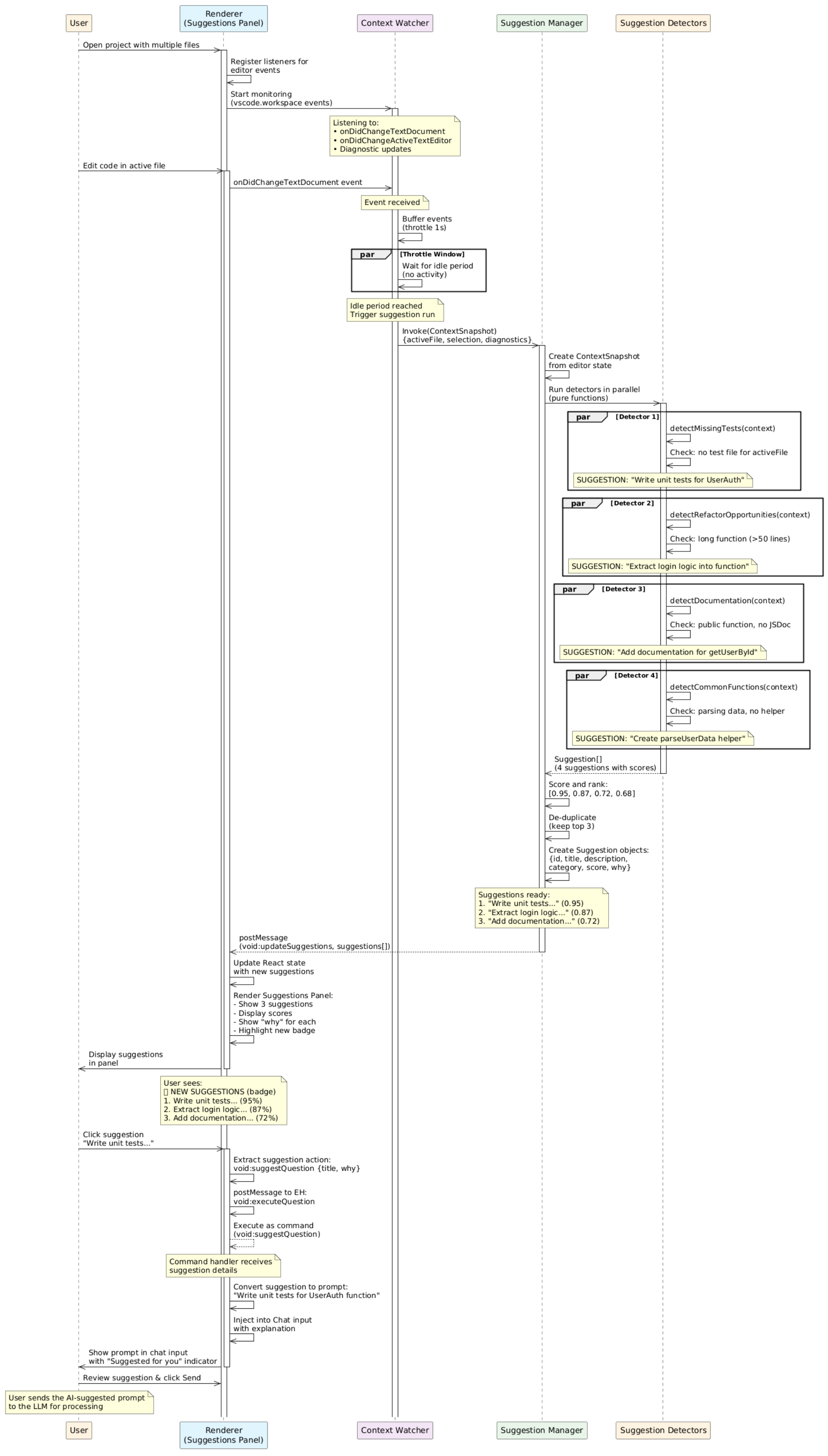
User
Renderer (Suggestions Panel)
Context Watcher
Suggestion Manager
Suggestion Detectors
Open project with multiple files
Register listeners for editor events
Start monitoring (vscode.workspace events)
Listening to:
• onDidChangeTextDocument
• onDidChangeActiveTextEditor
• Diagnostic updates
Edit code in active file
onDidChangeTextDocument event
Event received
Buffer events (throttle 1s)
par [Throttle Window]
Wait for idle period (no activity)
Idle period reached
Trigger suggestion run
Invoke(ContextSnapshot) {activeFile, selection, diagnostics}
Create ContextSnapshot from editor state
Run detectors in parallel (pure functions)
par [Detector 1]
detectMissingTests(context)
Check: no test file for activeFile
SUGGESTION: "Write unit tests for UserAuth"
par [Detector 2]
detectRefactorOpportunities(context)
Check: long function (>50 lines)
SUGGESTION: "Extract login logic into function"
par [Detector 3]
detectDocumentation(context)
Check: public function, no JSDoc
SUGGESTION: "Add documentation for getUserById"
par [Detector 4]
detectCommonFunctions(context)
Check: parsing data, no helper
SUGGESTION: "Create parseUserData helper"
Suggestion[] (4 suggestions with scores)
Score and rank: [0.95, 0.87, 0.72, 0.68]
De-duplicate (keep top 3)
Create Suggestion objects: {id, title, description, category, score, why}
Suggestions ready:
1. "Write unit tests..." (0.95)
2. "Extract login logic..." (0.87)
3. "Add documentation..." (0.72)
postMessage (void:updateSuggestions, suggestions[])
Update React state with new suggestions
Render Suggestions Panel:
- Show 3 suggestions
- Display scores
- Show "why" for each
- Highlight new badge
Display suggestions in panel
User sees:
▯ NEW SUGGESTIONS (badge)
1. Write unit tests... (95%)
2. Extract login logic... (87%)
3. Add documentation... (72%)
Click suggestion "Write unit tests..."
Extract suggestion action: void:suggestQuestion {title, why}
postMessage to EH: void:executeQuestion
Execute as command (void:suggestQuestion)
Command handler receives suggestion details
Convert suggestion to prompt: "Write unit tests for UserAuth function"
Inject into Chat input with explanation
Show prompt in chat input with "Suggested for you" indicator
Review suggestion & click Send
User sends the AI-suggested prompt to the LLM for processing
User
Renderer (Suggestions Panel)
Context Watcher
Suggestion Manager
Suggestion Detectors

Figure 3: Sequence diagram for UC1: Auto-Generate and Display Suggestions.

6.2 Use Case 2: Select Common Question via Keyboard Shortcut

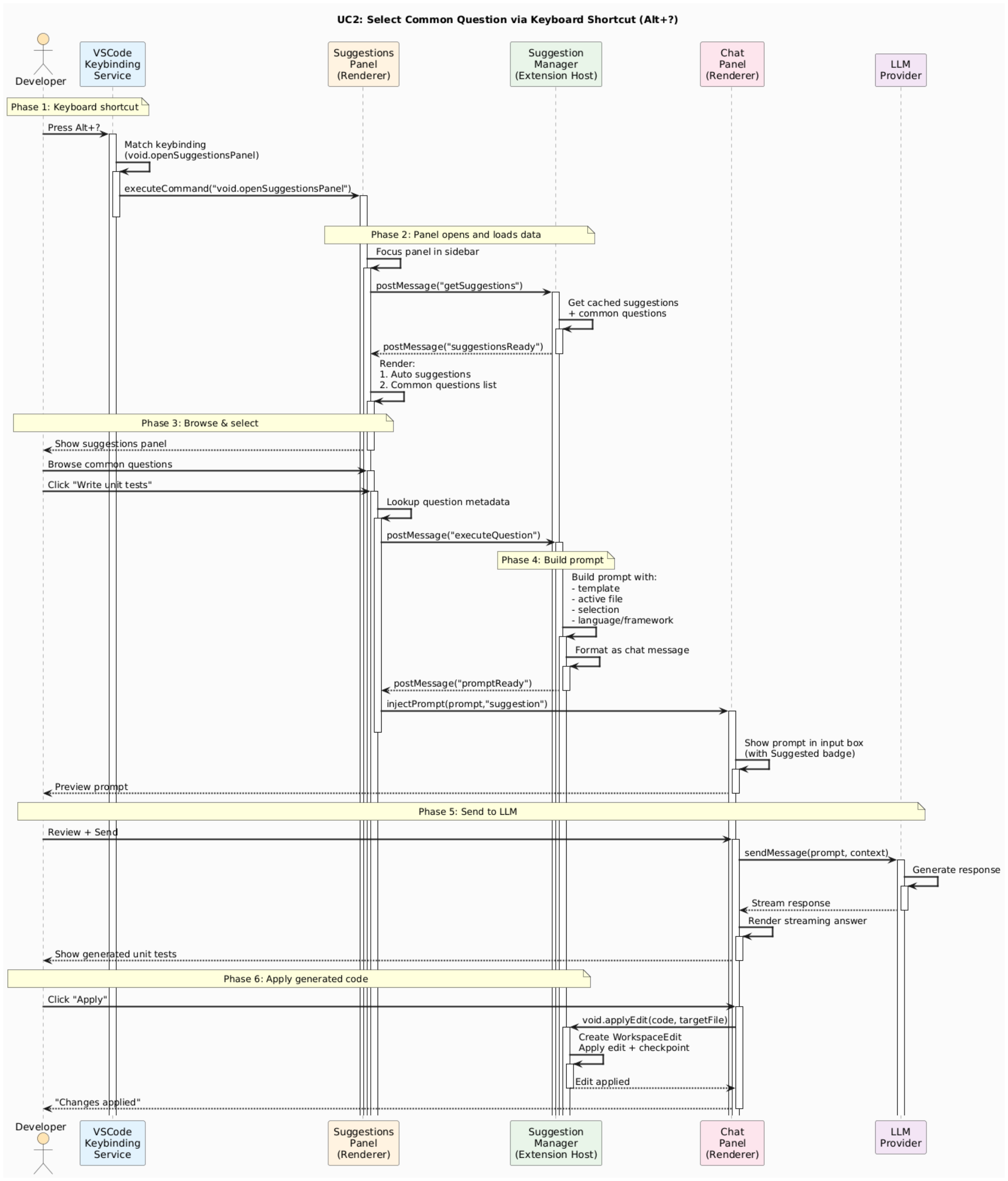
UC2: Select Common Question via Keyboard Shortcut (Alt+?)
Developer
VSCode Keybinding Service
Suggestions Panel (Renderer)
Suggestion Manager (Extension Host)
Chat Panel (Renderer)
LLM Provider
Phase 1: Keyboard shortcut
Press Alt+?
Match keybinding (void.openSuggestionsPanel)
executeCommand("void.openSuggestionsPanel")
Phase 2: Panel opens and loads data
Focus panel in sidebar
postMessage("getSuggestions")
Get cached suggestions + common questions
postMessage("suggestionsReady")
Render: 1. Auto suggestions 2. Common questions list
Phase 3: Browse & select
Show suggestions panel
Browse common questions
Click "Write unit tests"
Lookup question metadata
postMessage("executeQuestion")
Phase 4: Build prompt
Build prompt with: - template - active file - selection - language/framework
Format as chat message
postMessage("promptReady")
injectPrompt(prompt,"suggestion")
Show prompt in input box (with Suggested badge)
Preview prompt
Phase 5: Send to LLM
Review + Send
sendMessage(prompt, context)
Generate response
Stream response
Render streaming answer
Show generated unit tests
Phase 6: Apply generated code
Click "Apply"
void.applyEdit(code, targetFile)
Create WorkspaceEdit Apply edit + checkpoint
Edit applied
"Changes applied"

Figure 4: Sequence diagram for UC2: Select Common Question via Keyboard Shortcut.

Figure 4 describes the second use case. This use case focuses on the simpler shortcuts tab behavior.

When the developer presses the keyboard shortcut, such as Alt + ?, the VSCode keybinding service matches it and calls the command that opens the Suggestions Panel. The panel gets focus in the sidebar and sends a message to the Suggestion Manager asking for data. The manager returns any cached auto suggestions as well as the list of common questions, and the panel renders both lists for the user.

The developer then browses the common questions and clicks one, for example “Write unit tests”. The panel looks up the metadata for that question and sends a message back to the Suggestion Manager asking it to execute the question. The manager builds a full prompt using the template for that question plus the active file, selection, and language. It returns the prompt to the Chat panel, which injects it into the chat input with a “Suggested” label so the user can review it.

When the user sends the prompt, the Chat panel forwards it to the LLM provider, shows the streaming response, and then applies the generated code back into the file if the user clicks Apply. This use case shows how the feature still helps even when auto suggestions are off and the user mainly wants quick access to repeatable prompts.

7 Impact on Quality Attributes

This section links the enhancement to the main quality attributes in the rubric: maintainability, evolvability, testability, performance, and security. Many of these ideas appear in the earlier parts; here we group them.

7.1 Maintainability and Evolvability

The suggested changes to the VoidEditor architecture required to realise the FAQ and suggested prompts tab will produce the desired functionality, but as with any alteration to a system, they risk affecting non functional requirements of the whole product. Any concrete implementation would need to take care not to degrade the overall quality based on them.

Adding a suggested prompt and FAQ feature has a significant effect on Glass Devtools (the organisation with stewardship over the VoidEditor open source project) as it presents an entirely new stream of functionality in the editor that must be organised and maintained. With this added feature comes the added responsibility of keeping it in a usable state that integrates well with the rest of the product. Since VoidEditor is an open source application, its maintenance and improvement largely depends on the good will of contributors, so any increase in scale like this must be carefully considered, as there is not necessarily a body of people ready and willing to work on it. VoidEditor also relies heavily on its ability to integrate with the external APIs of LLM companies like OpenAI, which means additional features relying on increased traffic to an external product increase the chance of problems with no clear fix. In general, the primary non functional requirement that this stakeholder will be concerned about with our feature is maintainability.

The modular design in Figure 2 supports evolvability. The Suggestion Manager and detectors are separate units. New detectors can be added or removed without changing the rest of the system. This helps the feature evolve over time as users ask for new kinds of suggestions.

7.2 Testability

The new components also improve testability. Because the Suggestion Detectors are small functions that take a `ContextSnapshot` and return a suggestion, they are easy to unit test with mocked editor state. The Suggestion Manager can be tested to ensure it merges detector outputs correctly, sorts by score, and keeps the top items.

Integration tests can drive the flows shown in Figures 3 and 4: open a project, simulate edits, open the panel, pick a suggestion, and verify that the right prompt and apply-edit calls are sent. This supports the rubric item on testability and also ties into the testing plans we outline later.

7.3 Performance

Performance is also a strong concern when adding any extra feature. Most of VoidEditor right now operates on the principle that additional, non VS Code features (such as calls to external APIs) are happening infrequently, and only when the user initiates it, but the changes we are suggesting will increase the backend overhead for general usage. We now require all changes to the codebase to be monitored by the new Watcher module, so that the Suggestion module can avoid having stale suggestions based on versions of the code that no longer exist.

The implementation must take care to have this monitoring be as lightweight as possible, which is why we would intend to leverage the existing MCP tooling that VSCode supports to reduce the computational cost of hooking into the extended environment. It is important that the performance of the editor is not impacted by these checks, as an unresponsive IDE is essentially unusable.

Given the amount of typing and changes that the average user carries out even just on a small project, and the need to support large codebases distributed across numerous files, any kind of delay during an average workflow quickly becomes a pain point that could push users away from using it altogether, especially given that AI integrated IDEs are a competitive space with alternatives like Cursor readily available.

The design in Figures 1 and 2 addresses this by throttling analysis to idle periods, by limiting detectors to small snapshots, and by running suggestion logic in the extension host rather than the renderer thread.

7.4 Security and Data Privacy

Data privacy is another key non functional requirement which is both essential and at risk with the new features being added. As it stands, VoidEditor does not in and of itself store any information about the user or their code. VoidEditor is essentially a middleman between the user and their code, and their LLM of choice, allowing for a more tightly integrated workflow.

The FAQ and suggestion tab however relies on VoidEditor handling that information itself, taking what it sees in the codebase and independently from the user producing the appropriate suggestions. With this in mind, we need to make sure the software is only looking at the correct files, so that the user's device is not scanned unnecessarily, and that the suggested prompts do not end up with sensitive information in them that the user would not want being sent out to a remote LLM.

Keeping context analysis local in the extension host and letting the user choose when to send prompts out, as shown in Figures 3 and 4, helps meet this requirement.

8 Alternatives for Realizing the Enhancement

An alternative approach to the same enhancement is an assistant that reads your current code and generates possible questions with suggested answers. This version of the feature does more than pin a set of common prompts in a new tab. It watches what the user is doing and proposes questions they might not think to ask. The panel would look at the active file, recent edits, diagnostics, and a few simple patterns, then surface ideas like "extract this into a function", "write a quick unit test for the new branch", "summarize these changes for a commit", or "explain the API used on this line". The goal is to save time and also help people notice work they would otherwise miss.

We now name our two alternatives:

- **Alternative A - Shortcuts panel.** A simple panel that mostly shows common prompts and lets users trigger them quickly. UC2 and Figure 4 show how this behaves.
- **Alternative B - Proactive assistant.** A context aware panel that reads the current code, runs detectors, and suggests questions with answers. UC1 and Figures 2 and 3 show this design.

For end users the two big non functional requirements are usability and performance. Usability improves if the suggestions feel timely and relevant, but it gets worse if the panel nags or clutters the screen. The safest design is to keep suggestions inside the new tab, show a small badge when new ones arrive, and include a short "why this" note beside each item. People can then decide if they trust it. Personalisation matters too. We can let users mute classes of suggestions and prefer the kinds they accept most often.

Performance is the other concern, as we cannot scan a huge workspace on every keystroke. The feature should rely on signals we already have, like LSP diagnostics, the active buffer, and a short history of commands. Run analysis in the extension host, not the UI thread, and throttle it to idle periods or after a quiet interval. Done this way, the editor stays responsive while the tab feels helpful rather than busy.

For the core team at Glass Devtools, the key concerns are maintainability, code quality, and performance at scale. This enhancement becomes a long lived capability, so it needs clear ownership and stable structure. A good implementation is a small Suggestion Manager in the extension host, supported by a lightweight Context Watcher, which keeps the renderer simple and the editor core untouched. Inside the manager, use small detectors that each check one condition and return a suggestion with a reason and a score. That design is easy to unit test with mocked editor state and diagnostics, which helps sustain code quality over time. Performance is non negotiable in an editor. Even small delays break flow. Keep detectors pure and cheap, run them on small snapshots, apply strict time budgets, and throttle to idle periods so typing and navigation stay instant. Privacy stays intact because all analysis runs locally, and content leaves the machine only when the user chooses an action that calls a provider.

Open source contributors want modularity and stable, well documented hooks. This approach can be contributor friendly if we expose a very small detector API. A detector takes a snapshot, proposes a suggestion, and nothing else. The manager de duplicates and ranks. That keeps contributions narrow and reviewable, and it lowers the chance that a plugin will slow the system down. The enhancement can create a natural extension point where the community can add detectors and ideas that target specific languages and frameworks, which is positive for modularity and ecosystem growth. The price is documentation. We should ship a short authoring guide, a handful of examples, and clear rules about noise budgets so the panel does not turn into a stream of low value tips. Good docs also help the core team review pull requests faster, which supports the maintainability goal.

9 SAAM Analysis of the Alternatives

Our enhancement aims to make Void faster to use by adding a "Common & Suggested Questions" tab, with two ways to realize it. The first way is a simple shortcuts panel that surfaces frequent prompts. The second is a proactive panel that reads the current context and proposes questions with suggested answers. The enhancement's values are consistent with Void: reduce repetitive prompting, help users notice work they might miss, and keep Void competitive with AI-enabled IDEs. The changes stay in the extension host and renderer, with no changes to the editor core. The added pieces are small, event driven, and align with how VS Code extensions already work.

In SAAM terms, we look at each stakeholder and their main non functional requirements.

- **End users.** With Alternative A, usability improves a bit for people who already know what to ask, and impact on performance and privacy is small. With Alternative B, usability improves more. Users get help they did not think to request. Performance and privacy need more care because the system looks at context continuously, even if analysis stays local.

- **Glass Devtools.** Alternative A is easy to ship and cheap to maintain. It looks more like a standard VS Code extension. Alternative B requires clear module boundaries (Context Watcher, Suggestion Manager, detectors), a test suite, and performance ownership, but it also provides a strong long term capability that can grow.

- **Open source contributors.** Alternative A offers limited contribution paths, mostly editing a list of prompts. Alternative B creates a real extension point through detectors, which is good for modularity and ecosystem growth but increases the documentation and review load.

- **Team and enterprise stakeholders.** Alternative A is simple to govern and easy to explain. Alternative B benefits more from policy controls and clear "what leaves the machine" guarantees, which can strengthen Void's privacy story if it is done well.

Comparing the two realizations, the trade off is straightforward. The shortcuts tab is easy to ship and low risk. It improves workflow for people who already know what to ask. The proactive AI generated tab demands tighter performance budgets, but it delivers more value. It raises discoverability, reduces cognitive load, and matches how people actually work in large codebases. Both can meet privacy and responsiveness if implemented as described, but only the AI version clearly differentiates Void from look alike editors.

From a SAAM viewpoint, Alternative B satisfies more important non functional requirements across the main stakeholders, so it is the preferred realization of the enhancement.

10 Impacted Directories and Files

To realise the enhancement, several concrete parts of the codebase will be touched.

- **Extension host code.** New modules will be added for the Context Watcher, Suggestion Manager, and Suggestion Detectors, for example under a folder such as `src/extension/suggestions/`. The main extension entry file will be updated to register commands for opening the Suggestions Panel, running auto suggestions, and executing a selected question.
- **Renderer webviews.** A new Suggestions Panel component will be added next to the existing chat webview. The chat webview will get a small change so it can label prompts that came from suggestions and display the "Suggested for you" badge shown in Figures 3 and 4.
- **Shared models or utilities.** A small shared file will define the `ContextSnapshot` and `Suggestion` object types used in Figures 2 and 3. Both the extension host and renderer will import these so the message formats stay consistent.

Listing these groups in this way helps reviewers check that the concrete changes match the conceptual and low level diagrams.

11 Testing Plan and Interaction with Existing Features

To test the enhancement and its interaction with current features, we plan a mix of unit tests, integration tests, and manual runs.

At the unit level, each Suggestion Detector will be tested with mocked ContextSnapshots. We will confirm that each detector only fires when its own conditions are met and that it returns the right suggestion text and score. The Suggestion Manager will be tested to make sure that it merges detector output, sorts by score, removes duplicates, and keeps the top suggestions.

At the integration level, automated tests will drive the flows from Figures 3 and 4. For UC1, tests will open a workspace, simulate edits, wait for idle, and verify that suggestions arrive in the panel. For UC2, tests will trigger the keyboard shortcut, load the suggestions list, pick a question, and check that the chat panel receives a correctly formatted prompt.

We will also test interactions with existing features such as normal chat prompts, apply-edit commands, and MCP tool usage. The goal is to show that suggestions never block typing, that the editor can still send direct prompts when the panel is closed, and that failures in external providers only affect the specific request rather than the whole UI. Basic performance tests on larger workspaces will help tune throttle intervals and time budgets for detectors so that the enhancement stays safe to use in everyday work.

12 AI Collaboration Report

12.1 Interaction with the AI Teammate

We brought in GPT-5 only after our main writing, diagrams, and design choices for the VoidEditor enhancement were finished. We treated it as an assistant for editing and structure, not as a co-author. GPT-5 is good at following detailed instructions, working with long documents, and keeping tone stable across sections, so we used it for small and mechanical tasks.

Most prompts were narrow. Examples included:

- "Check grammar and punctuation in this section, do not change meaning, keep our terms like VoidEditor, Context Watcher, Suggestion Manager."
- "Add one or two short sentences that point to Figure 1 and Figure 2, without changing the technical content."
- "Rewrite this paragraph so it mentions both use cases and clearly names UC1 and UC2, but keep our description of the flow."
- "Suggest section headings that match the rubric items, do not add new claims about the architecture."

With prompts like these, the model helped fix tense issues, remove repeated words, clean up long sentences, and make sure that figure captions and references were consistent. It also helped tune the wording around the four figures so that the text points clearly to Figure 1 (Modified Top-Level Concrete Architecture), Figure 2 (Modified Agent Orchestrator Subsystem), Figure 3 (UC1 sequence diagram), and Figure 4 (UC2 sequence diagram).

Every change went through a manual review. We pasted AI suggestions into a separate document, compared them against our original text, and only accepted edits that did not change the meaning. Any sentence that touched technical details, such as how the Context Watcher throttles events or how the Suggestion Manager runs detectors, was checked against the diagrams and our own notes.

12.2 Quality and Impact of the AI Teammate

We also used GPT-5 to check how well the report covered the rubric. We asked it to scan the draft and point out where we explained maintainability, testability, performance, and risks, and where we were missing detail. Based on this, we added our own sentences about test plans for interactions with existing features and about possible performance issues when monitoring large workspaces. In the same way, it reminded us to describe both alternatives for the enhancement in a more structured SAAM comparison, while the actual trade offs and stakeholder impacts remained our ideas.

For the diagrams, the model helped tidy the captions and suggested short linking phrases such as "as shown in Figure 2" or "this flow is captured in UC1 in Figure 3." It also provided LaTeX tips so the tall sequence diagrams would fit on the page and not float into odd places.

In terms of impact, the AI mostly saved us time on editing and formatting. It caught small language problems, a few missing figure references, and some inconsistent names for components, for example switching between "Common & Suggested Questions Panel" and "Suggestions Panel." It also reduced the effort of checking that our explanation of the use cases matched what happens in the sequence diagrams. The model's role was to clean up the surface, help align the structure with the rubric, and make the report easier to read without changing what we are actually proposing.

13 Lessons Learned and Team Notes

As a team we learned that even a small enhancement like the Common & Suggested Questions tab touches many parts of the architecture. We had to think about the renderer, the extension host, background events, and user flow at the same time. The diagrams in Figures 1–4 helped us see how all pieces connect and where performance or privacy problems might appear.

We also found that splitting the work by focus area worked well. Some of us concentrated on stakeholders and non functional requirements, others on the orchestrator changes, and others on the two use cases and SAAM alternatives. Regular check ins were important so that wording, names, and diagrams stayed aligned.

14 Conclusion

This report described an enhancement to VoidEditor that adds a "Common & Suggested Questions" tab. The feature introduces a new Suggestions Panel in the renderer and new context aware services in the extension host, as shown in Figures 1 and 2. Through two use cases and sequence diagrams in Figures 3 and 4, we showed how the system can both auto generate suggestions and let users trigger common prompts by shortcut.

We examined the impact of the enhancement on maintainability, evolvability, testability, performance, and privacy, and we discussed concrete risks such as extra maintenance load and possible performance regressions. We also compared two realizations of the feature using a SAAM style analysis. The simple shortcuts panel is easy to ship, while the proactive assistant offers more long term value and better fits the goals of VoidEditor as an AI aware IDE.

With careful attention to the quality attributes and testing plans outlined here, the Common & Suggested Questions tab can give VoidEditor users a smoother daily workflow and help Void stay competitive in the growing field of AI enhanced development tools.

References

[1] VoidEditor, “void,” GitHub repository. Accessed: Nov. 7, 2025. [Online]. Available: `https://github.com/voideditor/void/`

[2] Zread.ai, “Architecture Overview — Void Editor.” Accessed: Nov. 7, 2025. [Online]. Available: `https://zread.ai/voideditor/void/9-architecture-overview`

[3] Microsoft, “Visual Studio Code Wiki,” GitHub. Accessed: Nov. 7, 2025. [Online]. Available: `https://github.com/microsoft/vscode/wiki/`

[4] DESOSA, “Visual Studio Code — Architecture” (2021). Accessed: Nov. 7, 2025. [Online]. Available: `https://2021.desosa.nl/projects/vscode/posts/essay2/`

[5] Microsoft, “VS Code Extension API,” code.visualstudio.com. Accessed: Nov. 7, 2025. [Online]. Available: `https://code.visualstudio.com/api`