Enhancement for Void

CISC 322 - A3 Enhancement Proposal
December 1, 2025

Adam Abou Zaki (22dgl3@queensu.ca), Mehr Chelani (22twb2@queensu.ca), Miller Gao (22fy14@queensu.ca), Sarah Mohammad (sarah.mohammad@queensu.ca), Wangyi Lu (22xb30@queensu.ca), Yubo Wu (22cm58@queensu.ca)

Abstract

To enhance Void's ability to handle complex coding tasks, improve output quality, and increase execution accuracy, this report proposes integrating an AI multi-agent workflow system into the Void framework. Currently, Void mainly relies on single-step LLM calls, which limit multi-stage reasoning and collaboration between tools. Our enhancement evolves Void from a single-agent assistant into a multi-step, composable IDE, where multiple agents cooperate on tasks such as analysis, refactoring, testing, and explanation. We explore two implementation alternatives: an in-process orchestrator embedded inside the Void extension, and an external orchestration service built on Temporal. Using an SEI SAAM analysis, we evaluate both options against stakeholder NFRs and show that the in-process orchestrator better aligns with the dominant stakeholders and the existing Void layering. We also detail the architectural impacts, impacted directories, representative use cases, interactions with other features, and the main risks and mitigations for deploying multi-agent workflows inside Void.

Enhancement

Description

To enhance Void's capability in handling complex tasks, improving output quality, and increasing task execution accuracy, we propose integrating an AI multi-agent workflow system into the Void framework. Currently, Void primarily relies on single-step LLM invocations to generate code or answer queries, resulting in a lack of multi-stage, collaborative intelligent reasoning. This enhancement aims to evolve Void from a "single-agent assistant" into a multi-step, orchestrator, and composable intelligent IDE, enabling multiple agents to collaborate like a team in task handling. This enhancement primarily focuses on two areas: the Capability Module and Workflow Orchestration. The Capability Module enhancements mainly involve implementing local task-oriented agents (such as analysis, refactoring, and test generation) based on frameworks like AutoGen. The Workflow Orchestration enhancements enable the construction, scheduling, and execution of multi-step AI workflows, supporting both internal (built-in to Void) and external (service-oriented) implementation approaches.

Benefits of Enhancement

Extensibility

After integrating the Agent Orchestrator and Agent Plugin interfaces, new complex functionalities can be implemented by adding new agents or workflows, rather than modifying the existing core logic.

Evolvability

An agent (such as a Test Agent) can be reused across multiple workflows. Furthermore, different projects can be configured with distinct workflows while the underlying architecture remains unchanged.

Productivity

For users, transitioning from multiple rounds of Q&A (requiring repeated user input) to "executing the entire development task workflow with a single click" significantly reduces the time spent manually copying and pasting prompts.

Support for Advanced Use Cases

Workflow logic is centralized within Orchestrator/Workflow Definitions, while each Agent focuses on its respective subtasks: analysis, code generation, test writing, and so on. This prevents any single component from becoming overloaded and causing issues, while also enabling the handling of more complex tasks.

Implementation Alternatives

Integrated Agent Orchestrator inside Void Extension(Primary implementation)

This implementation integrates an agent orchestrator within Void's implementation layer (i.e., Void's internal architecture), which means the workflow orchestration logic for multiple agents is executed internally within Void. The Void extension will utilize mechanisms like GroupChat/AgentManager from AutoGen (Microsoft) to combine multiple agents (such as code analysis agents, refactoring agents, test generation agents, documentation agents, etc.) into a manageable multi-agent system. Within the Void internal architecture, a new In-Process Orchestrator will be introduced to handle the following responsibilities: Transmitting context and messages between multiple agents; Controlling workflow step sequencing (e.g., analyze → refactor → test); Managing internal agent dialogues, tool invocations, and output aggregation; Dynamically selecting appropriate agents based on user tasks; Directly interacting with VS Code extension APIs (files, editor areas, diagnostic information, etc.). The entire agent workflow executes entirely within the local extension process from start to finish, without relying on external services. Since the internal orchestrator runs within Void, agents gain high-speed, direct access to user code, editor state, and the file system, which can significantly improve overall execution efficiency.

External Orchestrator Service

This implementation would integrate AutoGen using Temporal as an external workflow orchestration service to schedule agents, thereby implementing an external agent workflow

service. In this approach, Void no longer embeds multi-step workflow logic within extensions. Instead, it migrates workflow orchestration capabilities to a separate external service (Temporal), achieving a true client-server architecture. The Void extension itself handles only UI commands, context collection, editor interactions, and the execution logic for local AutoGen agents. However, the structured flow of the entire multi-step task (e.g., analysis → refactoring → testing → documentation generation → review) is managed and scheduled by Temporal. As an external Workflow Orchestrator, Temporal schedules which agent runs when, handles retries and error recovery, runs steps in parallel where possible, persists long-term state so workflows can pause and resume, and exposes history/versioning/monitoring, with Void communicating over RPC or HTTP to send capabilities and receive results for display in the IDE.

SAAM Analysis

Major Stakeholders and NFRs

Void Users

The Void users are the everyday programmers who install Void on their own devices and use it as their primary code editor and IDE. They use Void to work on local projects, write and refactor code, run tests, and occasionally tweak settings or extensions, but generally do not manage servers or infrastructure. The NFRs these stakeholders value are performance, simplicity, and privacy. The multi-agent system should not severely impact the speed of the editor. The typing, scrolling, file management, and navigation should not lag, and the UI should stay quick. Heavy workflows should also stream incrementally, be cancellable, and not block the main editing thread. Users expect Void to work after basic installation and model configuration, without the need to deploy extra infrastructure. Simplicity in the setup means no separate servers, clusters, or databases are required to start working in the Void IDE. Many individual developers are concerned with sending code and information to third parties, so the Void architecture should continue to support this privacy. Running agents with local models and state should be an option, and remote models should be user-controlled.

Organizations using Void

These stakeholders are organizations that roll out Void across many teams. They are focused on reliability, scalability, and observability. With large teams come large workflows that must survive crashes, network blips, or reboots without losing work or data. Upon failures, the system should be able to bounce back and resume satisfying reliability. To work across several teams, the orchestration layer should be able to handle many concurrent workflows without overloading any one user's IDE or causing timeouts. Capacity should be able to expand so throughput grows with team size, satisfying scalability. Additionally, organizations need observability of workflows on repos, changes made, and who made them. Thus, logs, metrics, and histories should be accessible to support debugging, management, and compliance.

Void Developers / Maintainers

This stakeholder is the team that maintains the Void fork. They take Void changes, merge them, maintain the AI features, ship additions, and are responsible for the overall Void architecture. This stakeholder is concerned with modifiability, architectural clarity, and operational complexity. The ability to add new agent features, swap LLM providers, or change orchestration strategies should be possible without destabilizing the editor. Since Void already sits on top of VS Code and things get messy when architectural boundaries blur, dependencies between the new and existing components should be clean and intentional. Operational complexity should be minimized in the sense that Void should stay an app, plus an extension, plus a configuration, to keep the system reasonable as users grow.

Extension/Tool Authors

This stakeholder consists of people (internal or external) who build tools and extensions on top of Void. They value integration flexibility, API stability, and deployment portability. They expect clear contracts to plug in new agents/tools or trigger workflows. The APIs they use should not break when changes to Void are made, so compatibility is required. Their extensions may run in different contexts and environments, so the code should behave predictably across desktop, remote, and web environments.

SAAM Analysis Across Alternatives

Stakeholder	NFR	Alternative #1	Alternative #2
Void Users	Performance	High responsiveness: With the agent workflow system implemented within Void, orchestration runs with direct access to editor models and workspace, which is faster than network hopping. Short multi-agent tasks can be similar to the Quik Edit / Agent Mode currently seen in Void. Risk: Larger projects or heavy workflows can overload the extension system if not handled carefully or throttled	Adds latency: With an external workflow orchestrator, every step would require RPC to Temporal plus a worker dispatch. This can be tolerable for long flows, but very slow for short commands.
	Simplicity	Simple: Having the orchestrator integrated does	Complex: Using Temporal requires running or

		not require any additional infrastructure beyond Void and should work with the existing extension model. For the user, this entails simple engagement with Void.	accessing a cluster plus workers. For the user, this is not trivial. Too complex for everyday use
	Privacy	Strong: Tapping into Void's "any LLM, anywhere" philosophy, implementing the enhancement would ensure that orchestration, agent state, and code context can remain fully local. For the user, this ensures privacy as information does not need to be sent outside of Void.	Weaker: Using external orchestration could mean that workflow state and code snippets are sent to a remote Temporal service. This could be mitigated with self-hosting Temporal, but this is very difficult.
Organizations	Reliability	Limited: The in-house Void extension host is not necessarily a durable execution engine This means that long workflows can be fragile unless Void implements its own recovery systems	Strong: Temporal is specifically designed for durable execution. It automatically persists workflow history and can resume easily after crashes or restarts.
	Observability	Harder: Void would have to build its own AI workflow logs, dashboards, and history views inside the IDE, which increases effort	Built-in: Temporal offers AI workflow histories, queries, and monitoring, giving teams centralized visibility into each workflow step and failure
	Scalability	Okay: Scaling up would require more Void instances. Since the orchestrator is tied to each IDE session, coordinating multiple large workflows is harder	Strong: Temporal scales by adding workers and cluster capacity. This means many large and long-running workflows across repositories and users can be queued and processed

			centrally.
Void Developers	Modifyability	Easy: Code modifications are localized to Void, so modifying agents and orchestration patterns is straightforward Risk: Increase in complexity inside the extension host	Harder: Temporal workflows can be modified independently from the IDE, with the cost of learning Temporal's model and managing versioning constraints
	Maintainability	Riskier: Having the enhancement within Void increases the chance of overloading the system/extension host.	Better Layering: Orchestration as a separate external service allows Void's main and extension systems to handle UI, context, and client responsibilities, aligning with a client-server style architecture
	Operational Complexity	Low: With the integrations, there are no new runtimes beyond what Void already depends on, making it easier to ship as a single extension with AutoGen libraries	High: Needs documented Temporal deployment, worker management, and configuration, which is heavy for an IDE product.
Extension / Tool Authors	Flexibility	Good: APIs can expose in-process registration of new agents or workflows, and third-party extensions can link with the orchestrator	Decent: Authors can implement supported Temporal activities or workers, and workflows can be triggered from Void via RPC, but this can be complex
	Stability	Weaker: If the integration API lives inside Void, it would be closely tied to the orchestrator's internal types and data structures. If usage or workflow models are changed, extensions may be more exposed to breaking	Stronger: With an RPC workflow API to Temporal in place, extension authors are more protected from internal changes.
	Portability	Mixed: If the orchestrator runs inside the VS Code/Void	Mixed: The Temporal integration calls a workflow

		extension host runtime, it may not be identical across desktop, remote, and web environments. The same integration may not behave consistently across all hosts if some workflows are not supported on some hosts.	API over a network, so the same code can run in desktop, remote, or web extension hosts. This, however, depends on the connectivity, and some environments may not reach the required endpoint.

Observations

From the SAAM analysis, we can deduce that the in-process orchestrator inside Void is the better realization for this enhancement. This alternative aligns goals with the dominant stakeholders and their high-priority NFRs. Individual users get speed, a simple setup, and privacy through local workflows using the existing model connector and extension host. Void developers avoid supporting a foreign external platform, Temporal, and can keep orchestration localized inside the familiar VS Code/Void layering, which supports modifyability, architectural simplicity, and reduces operational complexity. Extension and tool authors also see benefits from the simple, local API surface that does not depend on network connectivity or a separate workflow cluster. This supports their desire for flexibility and portability, and some stability. Where the external Temporal orchestrator may see better results lies with the organization's valuing reliability, stability, observability, and scalability. Temporal is a better option for these NFRs because it hosts built-in features that satisfy them for long-running workflows.

Effects on high and low-level Conceptual Architecture

At the high level, the original Void architecture has a mostly linear AI pipeline: UI/Workbench → Bridge/Command Layer → Model Connector & Tools → Data/Config, which essentially turns user intent into a single LLM call or tool action. With multi‑agent workflows, you insert a new AI Workflow Layer between Bridge/Commands and the Model/Tools layer; now the UI/Bridge sends high‑level tasks (like “refactor + generate tests + write docs”) to this layer, which decomposes them into ordered steps, picks the right agents/tools for each step, and orchestrates their execution before handing concrete model/tool calls down to the existing Model Connector and Tools subsystems. Conceptually, Void stops being “just a smart prompt router” and becomes a small orchestration engine: the user still sees a single command in the Workbench, but internally that command is expanded into a multi‑step plan that the new layer coordinates.

At a lower level, the implementation realizes this by adding a new in‑process “AI Workflow” subsystem inside Void’s existing AI Integration/Bridge stack, rather than introducing any external service. Concretely, the UI/Workbench and Bridge/Command code paths remain the same entry points (e.g., Quick Edit, Agent Mode, or a new “Run Workflow” command), but

instead of calling the Model Connector directly, they call a WorkflowService (or equivalent) that: interprets the request, constructs a workflow graph or plan, invokes specific capability modules (AutoGen-style agents, ToolsService actions, model calls via LLMMessageService), and tracks state and intermediate results. All of this runs inside the existing renderer/extension host and main-process boundaries Void already uses, so dependencies stay within Void's own components (Workbench, Void UI, Model Service, ToolsService, Platform), and VS Code/Base layers are only touched through the same service interfaces they already expose.

Impacted Directories

Due to the newly introduced In-Process Orchestrator and the AutoGen-based multi-agent workflows, several existing modules in the VoidEditor codebase will be extended or reused:

VoidEditor/src/workbench/... (Void UI / Workbench UI command wiring)
VoidEditor/src/voidPlatform/scripts/... (bootstrap / main scripts for Void Platform)
VoidEditor/src/modelService/aiEmbeddingVector/...
VoidEditor/src/modelService/aiRelatedInformation/...
VoidEditor/src/aiProviders/base/...
VoidEditor/src/vsCodeEditor/editor/...
VoidEditor/src/fileSystem/files/...
VoidEditor/src/extensionSystem/extensions/…

The orchestrator and configuration files also need to be updated to adapt to the new internal architecture and the AutoGen integration:
VoidEditor/src/voidPlatform/scripts/main.ts
VoidEditor/src/voidPlatform/scripts/server-main.ts
VoidEditor/src/voidPlatform/scripts/bootstrap-node.ts
VoidEditor/src/voidPlatform/scripts/bootstrap-window.ts
VoidEditor/resources/config/void-ai-settings.json
VoidEditor/extensions/void/package.json

On the AutoGen side, the enhancement relies on the following Python packages and directories:
autogen/python/packages/autogen-core/...
autogen/python/packages/autogen-agentchat/...
autogen/python/packages/autogen-ext/…

Maintainability, Evolvability, Testability, & Performance

Maintainability
With an in-process orchestrator, all multi-agent logic lives inside Void's existing extension/workbench architecture, so maintainers only need to understand and modify code in

the same Electron browser/main processes and service layers they already work in (Void UI, Model Service, ToolsService, etc.). This increases the size and internal complexity of the Void codebase, but the impact is localized. Orchestration can be implemented as another service behind existing service contracts, rather than spreading logic across an external platform, which keeps debugging within the familiar VS Code/Void structure.

Evolvability
Evolvability improves because the in-process orchestrator can be refactored like any other Void subsystem. Maintainers can change planning algorithms, agent role definitions, context-selection strategies, or tool-calling policies by editing a single service pipeline, while preserving stable interfaces to the Workbench UI, Model Connector, and ToolsService. New agent patterns (different coordination styles or safety checks) can be added incrementally by composing or swapping internal components, without versioning or migrating external workflow definitions or databases, so architectural change remains under Void's direct control.

Testability
Testability benefits because the orchestrator can be tested the same way as for other Void services. Maintainers can use the extension-host environment, mock the Model Connector and ToolsService, and run deterministic scenario tests for agent workflows (Quick Edit, multi-file refactors, Gather/Agent modes) without bringing up an external workflow engine. Unit tests can target pure orchestration logic (planning, step sequencing, error handling) while integration tests validate end-to-end flows through the Bridge, LLMMessageService, and editor services, giving fine-grained coverage without network flakiness or remote dependencies.

Performance
Performance is generally better for end users because the orchestrator runs in-process. Agent steps invoke the existing Model Connector directly, avoiding the extra network hop and serialization overhead of sending every decision through an external Temporal cluster. At the same time, this design puts more CPU and memory load on the IDE processes, so Void must continue to enforce the concurrency and cancellation patterns already described in its AI pipeline (prioritizing UI responsiveness, streaming, back-pressure, and cooperative cancellation) to ensure long-running workflows do not starve editor interactions.

Use Cases

Use Case #1 Safe Refactor

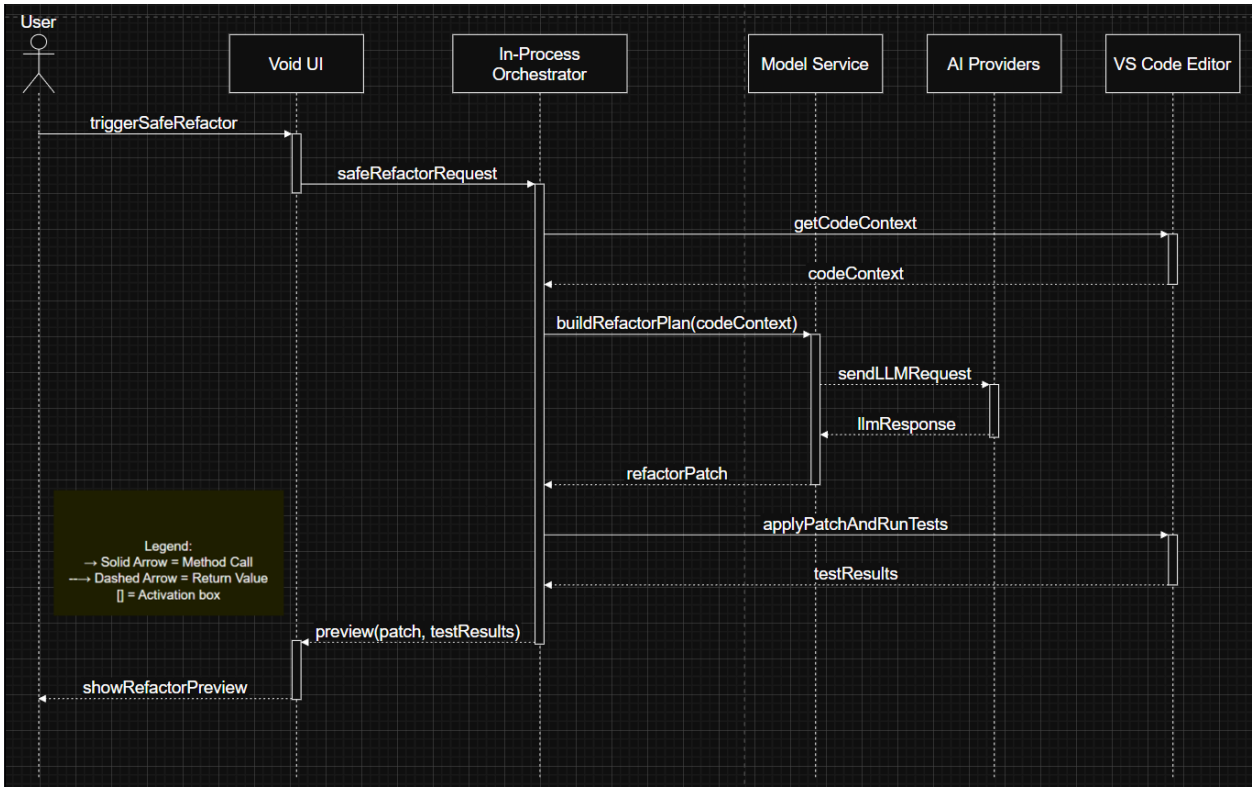

Figure: *Use Case 1 Safe Refactor*

While the developer is editing a source file in Void, they trigger the Safe Refactor command on the current selection. The Void UI forwards this request and basic editor context to the In-Process Orchestrator inside the Void extension. The orchestrator first queries the VS Code Editor to obtain the relevant code and symbols, then asks the Model Service to perform an automatic analysis → refactor → test workflow. The Model Service builds an analysis request and calls the AI Providers to analyze the code and identify refactor opportunities. Based on this analysis, it requests a concrete refactor patch from the AI Providers. Finally, it generates test updates for the affected code. The resulting patch and tests are returned to the orchestrator, which applies them in the editor environment and runs the relevant tests via the VS Code APIs. The orchestrator then sends the patch and test results back to the Void UI, which presents a Safe Refactor preview so the developer can inspect the automatically analyzed, refactored, and tested changes before applying them to the workspace.

Use Case #2 Auto Patch from Failing Test and Error Log

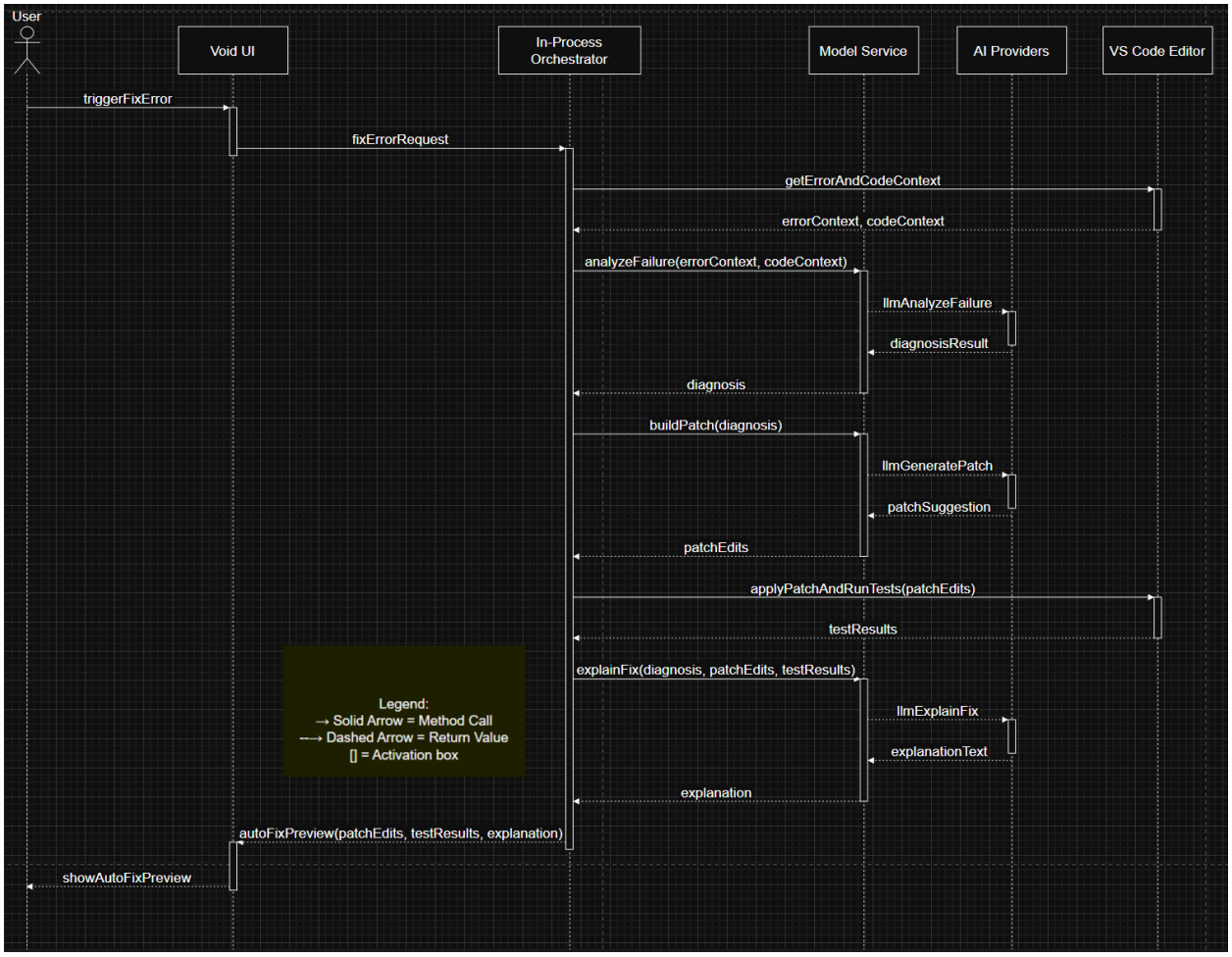

Figure: *Use Case 2 Auto Patch from Failing Test and Error Log*

While working in Void, the developer encounters a failing test or a runtime error in the VS Code testing or problems view. They invoke the Fix This Error with Agents command from the Void UI. Similar to Use Case 1, the Void UI forwards the failure information and basic editor context to the In-Process Orchestrator, which uses the VS Code Editor APIs to collect the relevant error and code context. However, unlike Use Case 1, this use case is reactive and is driven by a concrete failure signal. From this point, an automatic analyze → patch → verify → explain workflow is executed inside Void. The orchestrator asks the Model Service to analyze the failure, and the Model Service calls the AI Providers to diagnose a likely root cause. Based on this diagnosis, the Model Service requests a concrete patch for the affected code and returns the patch to the orchestrator. The orchestrator applies the patch in the editor environment and re-runs the relevant tests to verify that the failure is resolved via the VS Code APIs. Finally, the orchestrator also asks the Model Service to generate a short natural-language error log, what was changed, and any remaining risks. The resulting patch, test results, and explanation are sent back to the

Void UI, which presents an Auto Fix preview so the developer can inspect the automatically analyzed, patched, verified, and explained change before deciding whether to apply it.

Enhancement & Other Features

Plans for Testing with Other Features

We will verify that the multi-agent enhancement is compatible with Void's other subsystems and find out how it interacts with current features by testing end-to-end workflows across all major services. Each workflow will be guided by realistic user stories with human-in-the-loop checkpoints where AutoGen suggests them, and the Agent Orchestrator will coordinate AutoGen-style agents. After tagging the results to link them to particular interactions, we will drill down by feature area as shown below.

For Editor and Refactor, agents will propose single-file and multi-file patches. After applying patches, we will confirm diagnostics refresh and verify undo/redo behaviour. We will interleave manual edits and alternative code actions to create conflicts in order to test concurrency. Next, we will verify that the tool safely cancels the application without making any partial changes. Because AutoGen encourages iterative critique, we also need to make sure that review rounds don't cause the user interface to stall. In the platform-side alternative, IPC backpressure should prevent UI stutter while still streaming partial responses to the editor, while in the in-process alternative, the extension host should remain responsive during lengthy LLM steps.
For File System & Git, changes will be applied as a single batch. The entire batch must roll back if any write fails (permissions, low disk, etc.). After a successful apply, we will stage and commit once and verify there are no leftover temporary files. We will also force a merge conflict and check that the workflow pauses, asks for user input, and resumes correctly once the conflict is resolved.

For the Test Runner and Terminal, after refactors, a TesterAgent will run the project’s tests, stream logs to the UI, and mark failures with file/line links. We will simulate a missing framework, a broken PATH, and long-running suites that time out. In terms of Providers and Policy, we will first block the network to verify that a local model is being used. Next, we will cause provider errors to make sure the system automatically tries again or fails with an unambiguous message.

For Extensions and the Extension Host, we will load one harmless extension that provides a read-only helper and another that attempts unsafe actions. The first should register and run with limited permissions, while the second must be denied and logged without interrupting the workflow.

We will also test Remote/WSL/Containers by running the same flows on a remote project. The file paths must work, the file watchers must go off, and the progress must keep going after a short disconnect. For Docs and Notebooks, an agent will update README sections and insert a notebook result cell. We will sync workflow templates between profiles for Settings Sync and Profiles, but we will make sure that secrets stay local. We will also check that a corrupted template fails validation without crashing.

Last but not least, we will measure performance and recovery. While multiple agents are running, we will record per-step time, memory usage, and UI responsiveness. In order to review results across all features, we will also record basic telemetry (start, step, error, policy block).

Interactions with Other Features
Agent-driven patches will touch the Editor more often, which means that diffs, diagnostics, and formatting will update in smaller loops. If the file changed in the meantime, cancel or rebase instead of making partial edits. The editor should always be able to respond and undo or redo right away.

On File System and Git, making changes to multiple files makes it more important to stage them and make single commits with clear messages. When there are conflicts, the workflow should stop and ask the user to take action before starting up again. They will start with "changed files first," show streamed progress, and let you stop or time out without stopping unrelated steps. Providers and policies affect behaviour at every step. For example, they should follow local-only/no-egress rules, try again when they hit rate limits, and switch providers when they are allowed, with clear status. Extensions can register helpers, but only with clear and limited permissions. Unsafe actions in untrusted workspaces are blocked, and extensions that don't follow the rules are kept separate so that workflows can continue.

For Remote/WSL/Containers, file ops and tests run on the remote side, paths resolve automatically, and a short disconnect pauses then resumes without duplicate edits.
Settings, profiles, and telemetry influence how teams adopt these workflows. Workflow templates and per-agent settings can sync between profiles, but secrets must remain local. Invalid or outdated templates should be caught by validation rather than causing runtime errors. Telemetry should only send information about high-level events and not code or secrets. This way, teams can see enough to improve performance and fix problems while still keeping their privacy. These interactions are meant to make multi-agent workflows feel like a natural part of the IDE. They should be faster when possible, careful when risky, and always able to recover.

Potential Risks of Enhancement

Security and privacy are the most important risk areas because multi-agent workflows increase both the input surface and the action surface. Any artifact that agents use, such as source files,

build logs, test failures, markdown, or even commit messages, can contain prompt-injection content that attempts to convince an agent to leak code, breaking the rules, or using unsafe tools. The risk increases when third-party extensions add agents or tools, or when the workspace is not trusted. To prevent this from happening, every agent must run under a capability manifest that is deny-by-default and evaluated centrally. For example, reading files may be allowed, but shell, git write, network, and editor-apply should all be separate. The policy engine should enforce "local-only / no-egress" modes, hide secrets and PII during context assembly, and stop outbound calls that break policy. This will give the user a clear reason to act. Moreover, high-risk steps such as executing a patch should require an explicit human checkpoint with a pre-apply diff; in untrusted workspaces, they should be blocked entirely. Finally, keep track of non-PII audit events for workflow.start, agent.step, tool.invoke, policy.block, and provider.fallback. This way, security reviews can check that sensitive actions occurred with consent and within policy.

Another deep risk area is reliability and state consistency across long, multi-step runs. Agents will read, plan, edit, run tests, and edit again. If the workflow tries again after a crash, disconnect, or provider timeout, it could leave half-applied patches, duplicate commits, or orphan temp files. The design must make each mutating step transactional and idempotent. In further detail, file edits first write to a staging area and then "flip" into place in an atomic way. Git operations are one commit per step with a unique idempotency token. Notebook and README updates always create the same bytes when they are run again. When you resume, the system checks to see if the repo and working files match that checkpoint. It skips side effects that have already happened and only asks for the non-deterministic parts again, like a model response. This is especially important if you choose a platform-side or external orchestrator with replay semantics, because after a crash, activities may run again, so steps must be able to tell when "I already applied this patch/commit" and exit without changing the state. When a user edits the same area, the workflow should either cancel or rebase instead of partially applying, and it should never commit a mixed state that makes linters or indexers confused. You can validate these guarantees by killing the orchestrator while it's applying, cutting the network during provider calls, and simulating a power loss between writing and committing. After restarting, the workspace should be clean, with no duplicate edits.

Lessons Learned

While completing this project, we were able to look at the Void / VS Code ecosystem from a more objective architectural perspective, rather than just as everyday users of the IDE. As a result, we better understood the trade-offs between embedding AI workflows directly in the client and offloading them to external infrastructure, especially around performance and complexity. In addition, by applying SAAM, we gained a deeper understanding of how to relate architectural decisions to stakeholder non-functional requirements and how to use scenario-based analysis to uncover risks early.

Conclusion

This enhancement evolves Void from a simple LLM integration into an IDE with an internal multi-agent orchestration capability. By introducing an In-Process Orchestrator, the design supports richer multi-step AI workflows while keeping deployment lightweight. Compared with the Temporal-based alternative, the internal approach better matches Void's desktop context, avoids extra infrastructure, and still leaves a clear path for future scaling and refinement.

References

1. Engineering, D. (2020, February 12). *temporal: Workflow Orchestration Engine | Douglas Engineering*. Engineering; temporal: Workflow Orchestration Engine | Douglas Engineering. https://blog.douglas.engineering/posts/temporal-overview/
2. *Extension Guides*. (n.d.). Code.visualstudio.com. https://code.visualstudio.com/api/extension-guides/overview
3. *Group Chat — AutoGen*. (2024). Github.io. https://microsoft.github.io/autogen/stable//user-guide/core-user-guide/design-patterns/group-chat.html
4. *How we scale workflow orchestration with Temporal | Airbyte*. (2025). Airbyte.com. https://airbyte.com/blog/scale-workflow-orchestration-with-temporal
5. Kumar, A. (2025, March 24). *Void IDE: The Comprehensive Guide to the Open-Source Cursor Alternative*. Medium. https://medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235
6. Loren. (2023, May 23). *How Durable Execution Works*. DEV Community. https://dev.to/temporalio/how-durable-execution-works-2b88
7. *What is AI Agent Orchestration? Benefits & How It Works*. (2025). Simplyask.ai. https://www.simplyask.ai/blog/what-is-ai-agent-orchestration-benefits-how-it-works
8. *What is Temporal? | Temporal Platform Documentation*. (2025). Temporal.io. https://docs.temporal.io/temporal
9. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, S., Zhang, X., Liu, J., Awadallah, A. H., White, R. W., Burger, D., & Wang, C. (2023). *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework*. GitHub. https://github.com/microsoft/autogen

AI Collaboration Report

AI Member Profile and Selection Process

- GPT-5 (OpenAI) – a large language model used for text generation, refinement, and architectural reasoning. GPT-5 was chosen for its advanced reasoning, technical writing, and context-aware architecture analysis, outperforming other models (e.g., Gemini 2.0, Claude 3) in producing coherent, structured documentation.

- Mermaid Diagram AI tool – used to automatically generate sequence diagrams from the textual descriptions we provided.

Tasks Assigned to the AI

Sequence Diagram Generation – GPT-5 drafted Mermaid code from the provided descriptions, including real message calls, of sequence diagrams that were converted by Mermaid AI into finalized diagrams.

Research for SAAM analysis – Helped connect enhancement impacts to NFRs specific to stakeholders

Proofreading and Polishing – GPT-5 performed grammar and clarity checks, ensuring consistent terminology.

AI Report Drafting – GPT-5 generated the first outline of this collaboration report

Interaction Protocol and Prompting Strategy

Our group maintained a shared prompt document to coordinate all AI interactions, allowing every member to contribute edits, add context, and refine instructions collaboratively.

Prompts were developed through a structured three-step process: Firstly, clearly defining the course, deliverable, and technical objectives; Secondly, specifying the required output format; Lastly, reviewing AI outputs, identifying inaccuracies, and adjusting prompts to improve clarity and technical precision.

Example prompt: “Review the following use case description for clarity and coherence. Suggest improvements in wording, structure, and technical accuracy while keeping the tone formal and consistent.”

Validation and Quality Control Procedures

To ensure the accuracy and reliability of AI-generated content, our team implemented a multi-step validation and review process:

- All statements produced by GPT-5 concerning design principles, architectural patterns, and technical terminology were cross-verified with lecture materials, course readings, and textbook sources to confirm factual correctness.

- Each Mermaid-generated diagram was manually compared with the team's finalized use cases and implemented logic to ensure consistency between visual representations and actual system behaviour.

- Human team members reviewed all GPT-5 outputs for clarity, conciseness, tone consistency, and compliance with assignment requirements.

Quantitative Contribution to Final Deliverable
GPT-5: 25% (research, drafting, editing, formatting, and consistency)
Mermaid AI: 5% (diagram generation)
Human team: 70% (writing, concept development, validation, integration, analysis)

Reflection on Human-AI Team Dynamics
Overall, integrating AI collaborators improved efficiency and workflow coherence. GPT-5 streamlined brainstorming and editing, while Mermaid AI eliminated manual diagram formatting. The AI tools encouraged more structured communication within the group, as members needed to articulate precise instructions and evaluate outcomes critically. Minor disagreements arose when interpreting AI suggestions, but these discussions deepened our understanding of architectural reasoning.