Queen's University - Fall 2025

CISC 322

Assignment 3

Enhancement Proposal

Group 7: The Trailblazers

Zijie Gan 21ZG6@queensu.ca

Nathan Daneliak 22TMX2@queensu.ca

Jinpeng Deng 22SS117@queensu.ca

Jeff Hong 22RQ20@queensu.ca

Emily Cheng 22TWS1@queensu.ca

Zipeng Chen 23QH10@queensu.ca

Abstract

This report proposes integration of a Voice-Enabled Interaction Layer into the Void IDE architecture, designed to augment the developer experience without disrupting the existing system structure. This enhancement introduces a new input method, audio input. It supports three key workflows: Voice-to-Chat, Voice Quick Edit, and Voice Commands. By utilizing a Speech-to-Text (STT) service as an input adapter named SpeechService in our architecture implementation, spoken audio is transcribed into text before entering existing Void pipelines. This design choice ensures that the core logic of ChatThreadService and LLMMessageService remains unchanged, keeping the same concrete architecture as in the previous report. Enhancement strictly adheres to a Text-in/Text-out Invariant. Our Voice-Enabled Interaction Layer functions solely as an input adapter, ensuring that the input data types, structures, and control flows entering the LLM Message Pipeline (as defined in A2) remain identical, thereby requiring no modifications to Void's core business logic. We perform a Software Architecture Analysis Method SAAM evaluation to compare two implementation strategies. Processing STT in the renderer process versus the main Void Platform process, analyzing their impacts on performance, maintainability, and security. The result is a low-coupling audio input modality that improves accessibility and encoding flow while conforming to the existing layered architecture style.

Introduction

We discuss how to integrate the Voice-Enabled Interaction Layer into the current Void architecture without changing the existing layered architecture. It supports Voice-to-Chat, Voice-quick-editing, and Voice Commands. The core principle is Audio→Text→Pipeline. We use SAAM to compare two implementation paths: static and real-time. The static implementation sends the entire audio file to the SpeechService component at once, and then sends the text to the LLM component after the conversion is complete. Real-time implementation segments the audio into frames and streams to SpeechService, allowing users to see partial transcriptions as they speak. User interaction begins in the Void chatbox by clicking a microphone button to activate the VoiceInputUI. This component renders real-time audio waveforms to visualize the recording. The audio stream is controlled by the VoiceSessionController in the renderer process, coordinating with the external SpeechService (defaulting to Azure AI) to transcribe speech. The system enforces a turn-taking protocol where the UI pauses audio input acceptance immediately after submission and remains locked until the LLM has fully generated its response. This enhancement preserves the conceptual and concrete architecture of previous reports because it strictly follows a text-in/text-out invariant: the SpeechService converts all audio into text before it enters the LLMMessageService or EditCodeService. High-level changes are confined to the VoiceInputUI for state management and the VoiceSessionController for lifecycle handling. We explicitly preserve the concrete architecture defined in Assignment 2. This architectural stability stems from our strict control over data flow: text transcribed by the SpeechService is routed via the VoiceSessionController to the Model Service (a core component identified in A2) in exactly the same manner as keyboard input from the Void UI. Consequently, we modify neither the data structures, interface protocols, nor processing logic of the Model Service or LLM Message Pipeline, achieving zero architectural erosion. This design minimizes architectural erosion but introduces managed risks regarding network latency and data privacy due to reliance on external STT providers.

Motivation of Proposal

Void is excellent at AI-assisted coding, but its interaction is bottlenecked by keyboard input. Developers often speak faster than they can type. To fix this, we propose the Voice-Enabled Interaction Layer. This lets users perform Voice-to-Chat, apply Voice Quick Edits to selected code, and execute Voice Commands through spoken audio. This functional upgrade improves accessibility for users who prefer verbal communication over typing and reduces the cognitive load associated with typing long, natural-language queries. The primary benefit of this enhancement is removing friction between user intent and AI execution, making the collaboration between users and AI more efficient.

Alternative Implementation 1

The first alternative to implementing our architectural enhancement is through a **static processing** approach. A voice-enabled interaction layer introducing new voice-based methods to utilize Void's various tools would be created. This layer uses a static Speech-To-Text (STT) component, SpeechService, to interpret the user's spoken commands. SpeechService also handles static text-to-speech (TTS) to return responses audibly. When users interact with the new speech-based components, a VoiceInputUI component acts as the interface. Lastly, there would be a VoiceSessionController component to handle spoken commands for shortcuts throughout Void.

SpeechService would easily allow a myriad of features. Firstly, using voice prompts to communicate with the user's chosen LLM. A big advantage to this alternative is a more stable interaction with Void's pre-established LLM Message Pipeline. The architecture of the LLM Message Pipeline is relatively straightforward, and the only changes needed to accommodate this new feature would be using the SpeechService component to convert the user's voice to text before passing it to the LLMMessageService component [7]. After the LLM Message Pipeline returns a message, it gets passed back to the SpeechService component and read aloud to the user. Furthermore, a similar concept can be used, connecting the STT component to the pre-existing quick-edit pipeline to let the user ask the LLM for code edits [4].

Finally, the VoiceSessionController can interpret voice commands from SpeechService for any shortcuts the user needs. Since the voice-enabled interaction layer fits between the other components, it makes sense to stick with Void's current layered architecture. More specifically, the voice-enabled layer would be under the browser process layer of the LLM Pipeline, to be optionally called upon if the user wants to use any voice-related features, and entirely skipped otherwise [7].

Alternative Implementation 2

The second alternative is to implement through a **real-time processing** approach. Under this alternative, the user explicitly starts and stops each session, and partial transcriptions are shown while the user is speaking. In both designs, audio is strictly converted to text before it enters any LLM pipeline, and we only add new components around the existing Void subsystems rather than modifying the underlying conceptual or concrete architecture [1].

In the real-time design, three main components collaborate to implement voice input: VoiceInputUI, VoiceSessionController, and a Speech-To-Text service named SpeechService. VoiceInputUI is integrated into existing Void surfaces, including the chat sidebar, the Quick Edit panel, and the command surface. It displays the microphone state (idle, listening, processing) and shows partial text in the relevant input field while the user speaks. VoiceSessionController runs in the renderer process and manages the lifecycle of each voice session. It starts and stops audio capture, splits the audio into frames, streams those frames toward STT, and receives partial and final transcripts. Once a session completes, the final transcript is passed to the appropriate entry point: ChatThreadService and LLMMessageService for a Voice-to-Chat message, to EditCodeService for a Voice Quick Edit, or to a VoiceCommandInterpreter for voice-triggered commands [1].

The Speech-To-Text SpeechService exposes a streaming STT API to the controller while hiding provider details, such as Azure integration, credentials, and rate limiting. Process placement can be varied without changing the logical responsibilities. Throughout, the core invariant is preserved so only text flows into the LLM pipelines, audio handling is confined to new voice and STT components.

SAAM Analysis

ALTERNATIVE 1: Static Processing

Scenario	**Main Impact of Static Processing**	**Key NFRs Related**	**Scope**
Long chat dictation in a single-voice session.	Static processing waits for the entire audio message to be finished before processing it to text. Better performance in terms of resources used, but may increase response time.	Usability, performance.	Voice + STT.
Voice Quick Edit.	Processing starts after the user finishes speaking. While processing the audio is less resource-intensive than real-time, it may take longer overall.	Usability, reliability.	Voice + STT + EditCode.
Network delays during streaming STT.	Since static STT processing is less resource-intensive, network delays and latency should be less of an issue than real time, but will still slow down the process overall.	Reliability, UX, complexity.	Voice + STT.
Complex real-time Voice Commands in the IDE.	Many voice command prompts are short, so waiting for the user to stop talking before processing will not impact the response time of voice commands heavily.	Usability, complexity.	Voice + STT + Commands.

ALTERNATIVE 1: Real-Time Processing

Scenario	**Main impact of real-time processing**	**Key NFRs related**	**Scope**
Long chat dictation in a single-voice session.	Natural long dictation, partial text while speaking.	Usability, performance.	Voice + STT.
Voice Quick Edit.	Same Quick Edit flow, but prompts can be longer and spoken.	Usability, reliability.	Voice + EditCode.
Network delays during streaming STT.	Streaming makes stalls and lag visible to the user.	Reliability, UX, complexity.	Voice + STT.
Complex real-time Voice Commands in the IDE.	Enables richer voice-driven navigation and actions.	Usability, complexity.	Voice + commands.
A second STT provider for real-time use.	Easy to plug in new STT providers through SpeechService.	Evolvability, maintainability.	SpeechService.

Interactions Between Enhanced and Existing Components

Testing for Alternative 1

Testing the static implementation and its impact on the existing components is relatively straightforward since the voice-enabled interaction layer is isolated from the other layers. We must check that the text the static SpeechService passes to the rest of the LLM Messaging pipeline is accurate. So unit testing is necessary to test a variety of sounds and syllables to ensure they are properly accepted and transcribed to text. We must account for various demographics and people who may use Void, testing is needed for the static SpeechService to accurately transcribe spoken accents. Furthermore, there are some edge cases to consider, we must see how SpeechService reacts when it gets gibberish or silence. These must have dedicated fail cases to avoid any unintended consequences.

After seeing that SpeechService can transcribe text, we then move on to testing how the other components interact when fed the script from SpeechService. We verify that the LLMMessageService component understands and formats the script before sending it to the AI. After checking that this interaction is successful, two functions of the new voice commands, communicating with the AI and creating quick edit prompts with voice prompts, are both facilitated through the LLMMessageService component. So if the LLMMessageService can properly receive scripts from SpeechService, the whole system should work. The main thing we have to look out for when testing the LLMMessageService's interaction is ensuring there are proper error messages to handle incomprehensible gibberish from the SpeechService. Testing that SpeechService can relay the LLM's response to the user audibly is done mostly in isolation from other components. The only connection we must check is that the LLM Messaging Pipeline can successfully pass the LLM response to the SpeechService component. Outside of that, testing the SpeechService TTS component is simply verifying that it can successfully process and speak a large variety of written text. Testing of the VoiceSessionController is very similar to testing how the LLM Pipeline reacts to scripts from SpeechService. We must test that after receiving transcribed text from SpeechService, VoiceSessionController correctly identifies commands with error handling for receiving empty text and gibberish.

Testing for Alternative 2

Introducing real-time voice also requires careful testing of existing Void features. Chat and Agent Mode must behave identically to the current release when voice is disabled, and voice-dictated messages must integrate cleanly with the existing chat model, including the ability to edit transcripts before sending. Quick Edit and Apply need to handle voice prompts in the same way as typed prompts, while preserving diff visualization, apply, and undo behaviour [1].

Inline suggestions and other in-editor AI features must continue to work correctly when voice UI elements are present, without breaking focus or keyboard navigation. We must verify that large and multi-root workspaces remain responsive when voice sessions run concurrently with file operations and search, and that workspaces with voice disabled do not incur unnecessary overhead. Overall, this testing strategy is designed to ensure that the voice enhancement is additive rather than disruptive [7].

Comparison & Rationale for Choosing Static Processing

Criterion	**Static Processing**	**Real-Time Processing**	**Implication**
Architectural fit.	Single STT step plugged into the existing LLM pipeline.	New UI, controller, and streaming SpeechService.	Real-time touches more subsystems.
Usability.	Short delay after speaking; acceptable for small prompts.	Very responsive, streaming feedback. Natural dictation.	The difference is small for typical Void prompts.
Performance & resource usage.	Batch STT, no streaming, lower overhead.	Continuous capture and streaming, higher CPU/network.	Static is lighter-weight.

Reliability & testability.	Fewer moving parts, simpler tests.	More failure modes (lag, dropped frames, partial transcripts).	Static lowers risk of regressions.
Complexity & implementation risk.	Localized changes around STT to LLM/Quick Edit.	Cross-cutting UI, controller, and concurrency changes.	Static better matches our scope and team capacity.

We recommend Static Processing (Alternative 1) as the better choice for this enhancement. First, it fits Void's existing architecture more cleanly. The STT component is added as a simple pre-processing step in front of LLMMessageService and Quick Edit. Only text enters the current LLM pipelines, and audio handling stays inside the new voice layer. This preserves the current layered architecture and keeps the change localized [7]. Second, the static approach meets our main usability needs without adding much complexity. For chat, users can still dictate long messages; they just see the result after they finish speaking. For Quick Edit and commands, most prompts are short, so waiting until the user stops talking has little impact on perceived speed. This matches the SAAM scenarios where static processing slightly increases response time but still provides acceptable usability [1]. Third, static processing is lighter and more reliable. Because it does not stream audio, it uses fewer CPU and network resources, and has fewer moving parts than real-time streaming. Testing can focus on STT accuracy (including accents, noise, and gibberish) and on the integration with LLMMessageService and EditCodeService. Once those paths are validated, the rest of the system reuses existing, well-tested behaviour. In contrast, real-time streaming would introduce extra risks around lag, partial transcripts, cancellation, and UI focus [7].

Finally, this choice aligns with scope and future evolution. Static processing delivers clear value (voice to chat and Quick Edit) while keeping the enhancement additive and low-risk. The STT and voice layer designed here can be reused later if Void decides to add full real-time streaming. In other words, Alternative 1 gives us a solid, low-risk foundation without closing the door on richer voice features in the future.

Stakeholders and Their Desired NFRs:

To start, one of the most important stakeholders are the **users**. They are the ones who decide whether Void is a success at the end of the day by choosing to download and use Void as their IDE. The users mainly care about how easy it is to access and use at any given time. As such, they care about non-functional requirements (NFRs) like portability to ensure they can use Void regardless of what they are using, and performance, since if Void's IDE is slow, there are many other alternative IDEs they could go to.

Another key stakeholder that a new feature would affect is the **core developers and contributors** of Void. They care mainly about the future of Void and how easy it is to continue to develop and improve Void's features. Their most important NFRs are modifiability to be certain the software remains straightforward to build upon well into the future, and maintainability so they do not have to allocate too many of their resources to fix bugs and errors. While the **AI Providers** are less invested in development, they are still affected by changes in Void, as an increase in Void's popularity could increase the popularity of their LLMs. So the LLM service providers care about the same NFRs as the users because they care about the popularity of Void.

Finally, the lesser **platform stakeholders** like Microsoft with their VS Code/Azure license, and Y Combinator (YC), which is a primary partner of Void [5]. Much like the AI providers, they care directly about the success of Void as a product and less about the technical aspect of Void. They also care mainly about NFRs like portability and performance to make sure the product is the first choice of as many people as possible.

Static Processing Impact On Stakeholder NFRs (Alternative 1)

The static processing approach lets users of Void experience new features and interact with older ones in many ways. Not only that, but the static processing approach also appeases the **users'** most important NFRs. Compared to real-time processing, static voice processing uses much less computational power. This lets voice-related functions be implemented without worrying about whether the user will need higher-end hardware to access all the new features, successfully keeping in mind portability and letting lower-end computers use Void to its full capability. When it comes to performance, static processing will result in little to no latency when the user is speaking for their voice prompt since it does not have to process their voice in real time.

For the **core developers of Void**, the way the new components would be implemented makes modifiability and maintainability easy. Since it is within its own layer, accessing it for any needed testing, modifications, or bug fixes is very straightforward. For example, since the main connection to the other components is passing the text converted by the STT component to the pre-existing LLM communication pipeline, any new issues that arise can be clearly identified as issues stemming from the voice-enabled interaction layer. Thus streamlining bug fixing by removing the process of finding which layer a bug originates from. As stated, **AI Providers and the primary partners of Void**, such as YC, care about similar NFRs as the users. The AI providers want their AI LLM to be accessed by more people through many different means, such as Void, and the primary partners simply want Void to be as widespread and as popular as it can be. Therefore, since the static voice processing approach satisfies the user NFRs, it also satisfies the AI Provider and primary partner NFRs.

Real-Time Processing Impact On Stakeholder NFRs (Alternative 2)

For **end users and developers who use Void**, it clearly improves usability and accessibility for voice-heavy workflows: long, multi-sentence prompts and rich Voice Commands become easier and more natural, and partial transcripts that appear as they speak provide better responsiveness and feedback than a purely static "record then wait" model. At the same time, end-user reliability becomes more sensitive to network quality and timing issues because streaming STT can expose delays, jitter, or partial results that static processing hides [7].

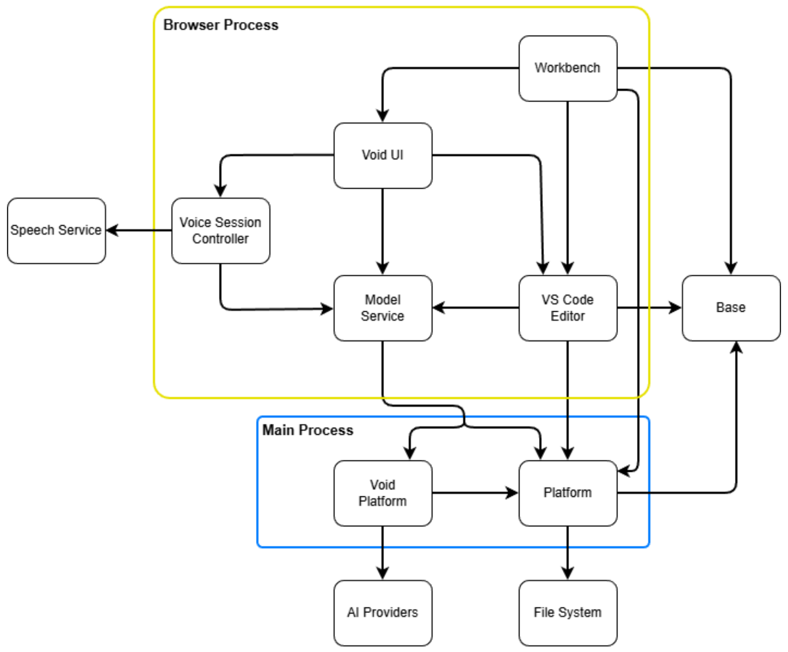
For the **core Void developers and contributors**, real-time processing is a trade-off between evolvability and maintainability, so once the streaming pipeline (VoiceSessionController + SpeechToTextService) is in place, it is a strong platform for future features (additional Voice Commands, richer accessibility modes, new STT providers). However, that same streaming state machine will increase complexity and make correctness or testability more demanding than a simple one-shot STT call [1]. For AI providers and STT/LLM services, the enhancement tends to improve recognition quality. It opens the door to richer usage patterns. Still, streaming connections complicate quota management, rate limiting, and observability, because calls are longer-lived and have variable bitrates rather than short, predictable bursts.

Finally, for **platform stakeholders** such as the VS Code ecosystem, Microsoft, and YC partners, real-time enhancement has a larger performance and security/privacy footprint than static processing. Continuous audio frames and partial transcripts put more load on the main/renderer processes and introduce more states to secure, but if SpeechService remains concentrated in Platform, the impact on overall architectural alignment and security can still be controlled and reasoned [7].

Enhancement Effects

High and Low Level Conceptual Architecture

With the addition of the voice-enabled interaction layer, we would see architectural changes in the input that Void allows. Without altering the core architecture surrounding the AI functionality, this new addition would require new components to handle speech-to-text (STT), text-to-speech (TTS), a new UI, and the audio itself. At the high level, this feature would be most prominent in the existing Void UI component. This component would contain the new UI, including a speech button, waveform indicators, speech settings, and taking the audio input from the user. Outside of the Void UI component, we would require two new components: VoiceSessionController and SpeechService. The current system operates based on text input for the AI models, for this reason, the two new components are required to format the audio into something valid that can be given to the AI. The SpeechService component will be external, providing STT and TTS services so that the AI can understand the user and the user can listen to the AI. VoiceSessionController will be the border between SpeechService and other components, such as VoidUI and ModelService. VoiceSessionController will handle audio and text data coming from all three of these components, as well as parsing through text input for custom speech commands to be given to the AI.



At the low level, no new components are required within Model Service or Void Platform because of the TTS and STT functionality added to allow the same program flow previously used. The main differences would appear at the boundaries of these components, specifically where they would communicate with other components. Within the VoiceSessionController, we would have modules that would take audio or textual data from SpeechService, textual data from ModelService, and audio data from VoidUI. VoidUI would contain new functions to receive and play audio data from VoiceSessionController, while ModelService would require new functionality to update speech commands based on VoidUI. With the help of STT, the new functionality is able to insert itself without much interference with other modules by converting the new input into the format that Void was already built for.

Maintainability

Due to the separation between our new enhancement and the core LLM pipeline, maintainability does not differ much from prior versions of Void. This isolation of components helps in maintenance and tracking any issues that may arise. The only affected areas would be the new Voice Session Controller component, maintenance of external STT and TTS providers within the Speech Service component, and possible maintenance in the Void UI component related to the microphone as an input method. Out of these three, the Voice Session Controller could become the largest issue due to it being an extra layer in the pipeline that would have to be debugged if there is an issue.

Evolvability

Our enhancement has great potential for future evolvability. Since the new components are isolated and SpeechService is external, there are many possibilities to upgrade only this feature without affecting other components. With this enhancement, users are able to use any external speech provider that they wish, similarly to the AI models, and this system allows for additions such as local models for SST and TTS in the future. Having access to the latest models keeps this feature evolving with those models, even if there is no active development going on with it.

Testability

The testability of our new feature brings some complications. With the use of hardware and external services it can become difficult to pin down the locations of any bugs, though there are some workarounds. In isolated tests, developers can use pre-made audio clips and mocks for Speech Service to isolate tests to the newly added code without relying on volatile components. Speech Service and microphone support can then be tested separately to ensure that every component works individually before doing full end-to-end tests. After the Voice Session Controller component, every other component should be fully tested based on text input, so there should be no changes there.

Performance

A slight increase in latency can be expected when using voice interaction. The addition of the external SpeechService can cause a noticeable increase in lag for the model to complete SST and TTS, depending on the audio length and the model being used. This performance hit will not impact any other components and will not be noticed by the user if they choose to use normal text-based interaction over voice interaction. All other additions should have little to no impact on performance and should function just as well as previous versions.

Use Cases Overview

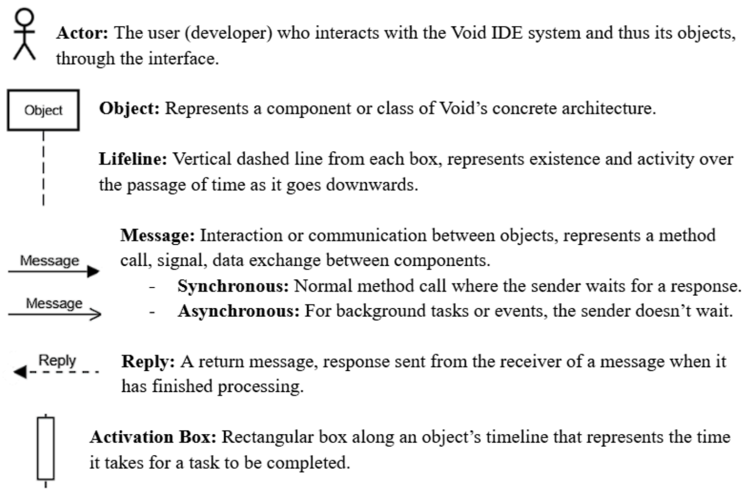
Discussed throughout the report, there are three main use cases that our enhanced feature introduces. All use cases begin where the user opens up Void's sidebar where the original messaging component lies, before clicking a button to activate the voice-enabled interaction layer component.

1. **Voice-to-Chat Message:** Users record audio through their microphone, which is transcribed by SpeechService to be passed through ChatThreadService and LLMMessageService, which produces an AI chat response displayed in the Void UI.
2. **Voice Quick Edit:** Users highlight code and speak an edit request, the resulting speech-to-text transcript is fed into EditCodeService. Like Void's original Quick Edit feature, it generates Diff Zones highlighted in red or green regions that the user can preview and apply [8].
3. **Voice Command Execution (Bindings):** The user speaks a supported voice command such as switching models, opening files, or triggering Agent Mode. VoiceSessionController then routes to the appropriate existing Void feature without involving the LLM pipeline or spoken AI responses. The IDE only reacts to components if it recognizes one that was already configured.

Above mentioned use cases build from existing Void features for maximum modifiability and integration. Improvements are also cost effective and reusable as nothing new is truly developed from the ground-up, instead borrowing finished code that handles the LLM messaging pipeline, normal chat interface, and apply system [7]. Like actual pipes, our feature can be inserted at the front of existing use case flows with minimal adjustments to the existing architecture. Because it relies on either converting user speech to text or AI text to speech, these new components are reusable for all other features and use cases which may require it in the distant future of Void's development. In the following subsections, we expand Voice-to-Chat Message and Voice Quick Edit in detail and with their corresponding sequence diagrams. Whereas Voice Command is a rather short explanation, these two focused use cases involve multiple interacting components which is more ideal for demonstrating the biggest architectural impacts of our enhancement.

Sequence Diagram Legend

Note: Asynchronous reply arrows are dashed asynchronous message arrows.



Actor: The user (developer) who interacts with the Void IDE system and thus its objects, through the interface.

Object: Represents a component or class of Void's concrete architecture.

Lifeline: Vertical dashed line from each box, represents existence and activity over the passage of time as it goes downwards.

Message: Interaction or communication between objects, represents a method call, signal, data exchange between components.

- **Synchronous:** Normal method call where the sender waits for a response.
- **Asynchronous:** For background tasks or events, the sender doesn't wait.

Reply: A return message, response sent from the receiver of a message when it has finished processing.

Activation Box: Rectangular box along an object's timeline that represents the time it takes for a task to be completed.

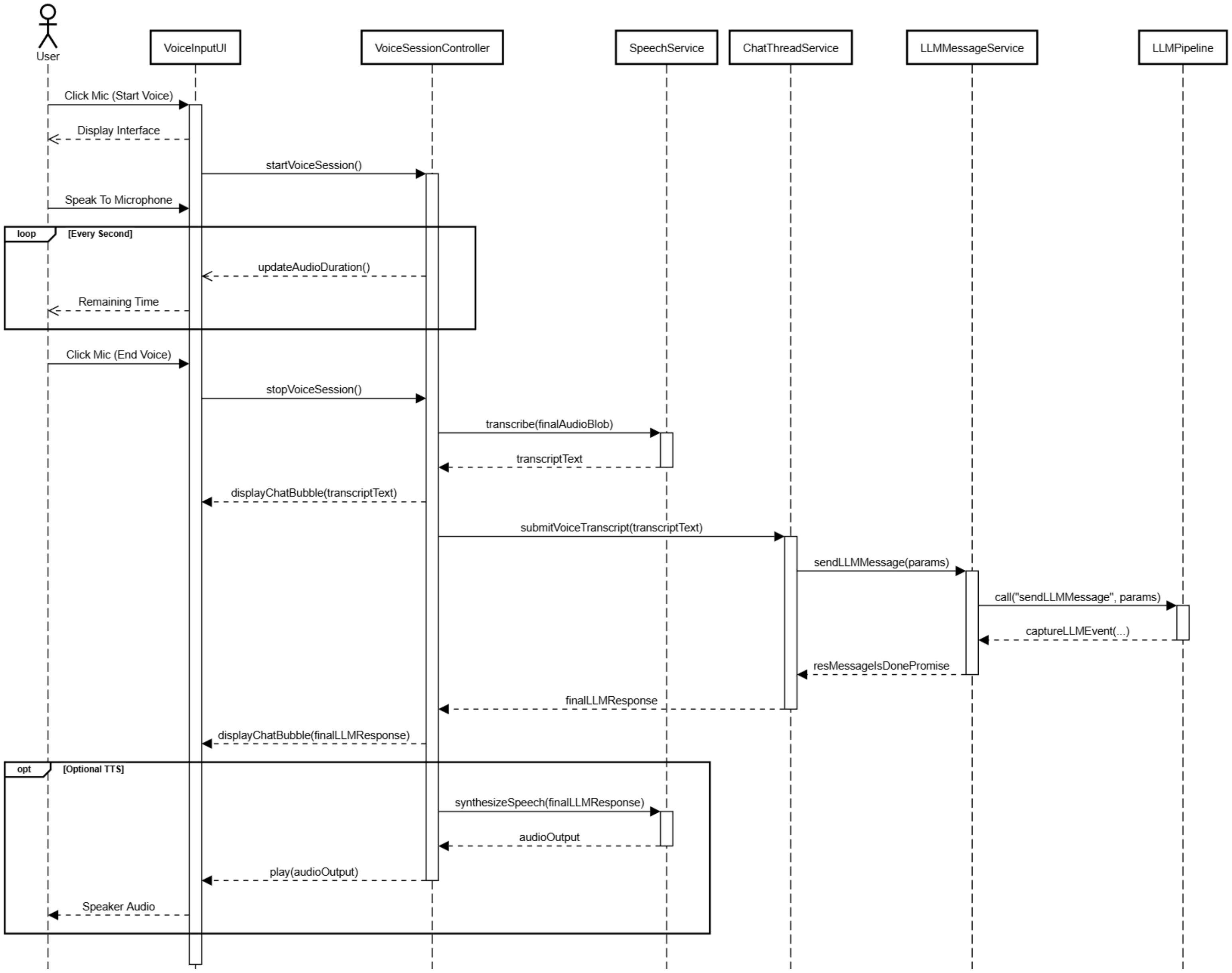
Use Case 1: Voice-to-Chat Message

The goal of this use case is to convert recorded audio into text so it can be processed by the existing LLM messaging pipeline which responds with a normal chat message and optional audio. By default, Void would choose Microsoft's Azure to act as the SpeechService component since it provides a reliable multi-accent support and end-to-end services [3]. This aligns well with key non-functional requirements such as performance and portability while being architecturally consistent with VS Code which is also by Microsoft. The diagram builds directly on an existing LLM Normal Chat mode mentioned in Assignment 2 with a new voice layer.

With the chat sidebar already open [8], the user first clicks a button to activate Voice-to-Chat messaging. It will display a small recording interface where the user speaks into their computer microphone for the component VoiceInputUI to forward captured audio to VoiceSessionController. A new listening session is started by this component through `startVoiceSession()`, ended when done speaking, and the voice interaction state is managed here. It sends the final audio blob to the SpeechService component with `transcribe(finalAudioBlob)` which performs Speech-to-Text conversion using an external provider such as Azure. The resulting transcript is passed to ChatThreadService using `submitVoiceTranscript(transcriptText)` which forwards it to the existing chat pipeline as if typed by the user. After the LLM generates its response, transcript text is displayed in the UI and synthesized back into audio by the SpeechService Text-to-Speech capability. This optional step, if enabled, calls `synthesizeSpeech(finalLLMResponse)` and allows users to hear the model's reply.

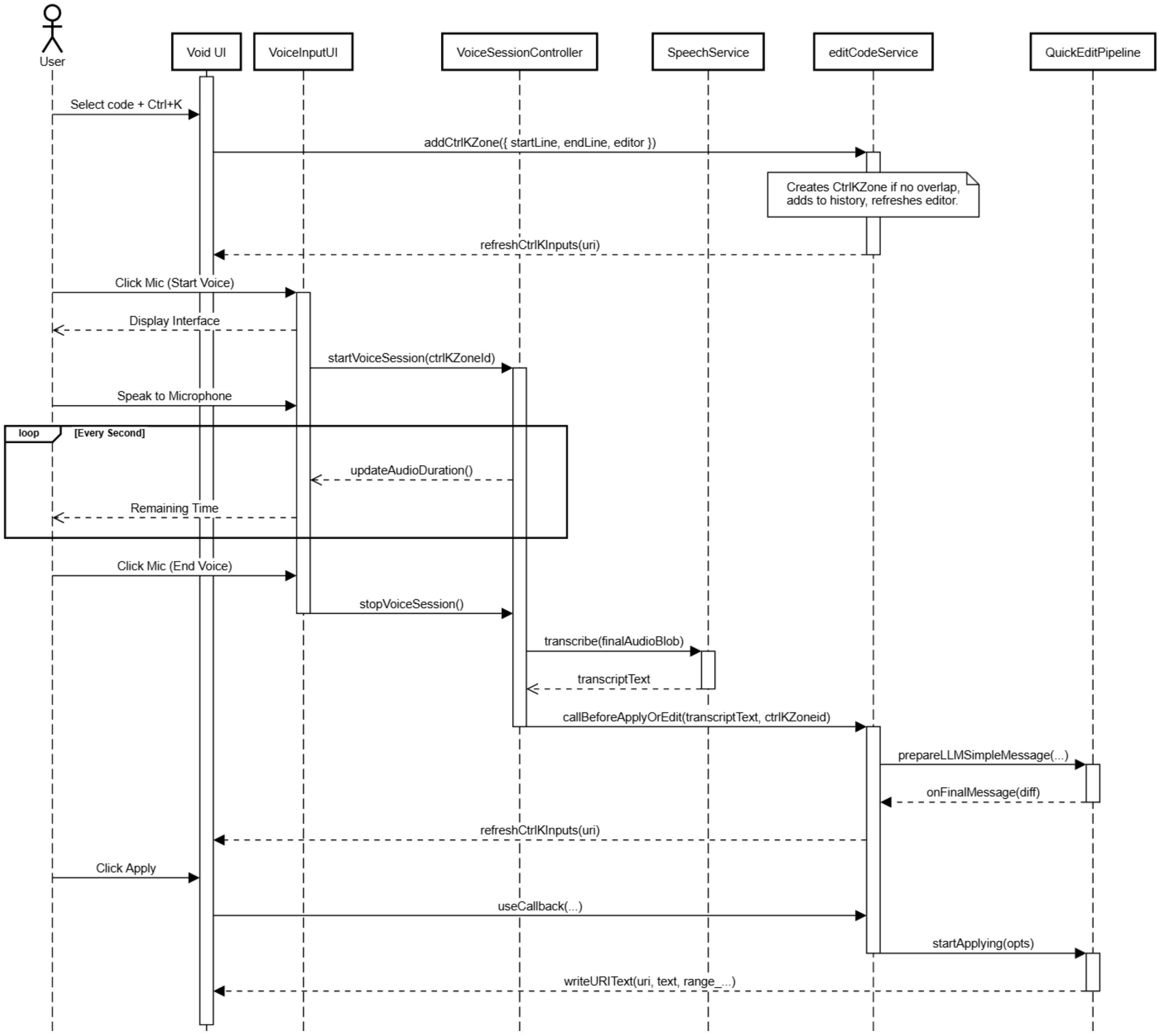
Users are made aware of this feature's maximum audio recording duration upon its release. Because the UI continuously displays remaining recording time using `updateAudioDuration()`, users always know how much audio can still be captured per second. If this time limit is reached, VoiceSessionController automatically terminates the current recording session, notifies the UI to display a message, and proceeds with SpeechService STT processing using the captured audio so far.

The VoiceSessionController component may temporarily hold final transcripts before routing it to the appropriate subsystem but long-term storage and chat history will remain the responsibility of ChatThreadService [2]. Recorded audio would be limited in duration to avoid excessive STT processing time and hitting provider usage limits. Furthermore, Void UI already handles all visual interactions like the chat sidebar and editor interface so embedding voice controls within it keeps the architecture consistent [6]. VoiceInputUI is a small subcomponent, reusing Void UI's rendering and event-handling behaviour but adding the microphone button, timer, and recording indicators. This way, our new feature feels like a natural extension of the existing interface instead of a new layer entirely.



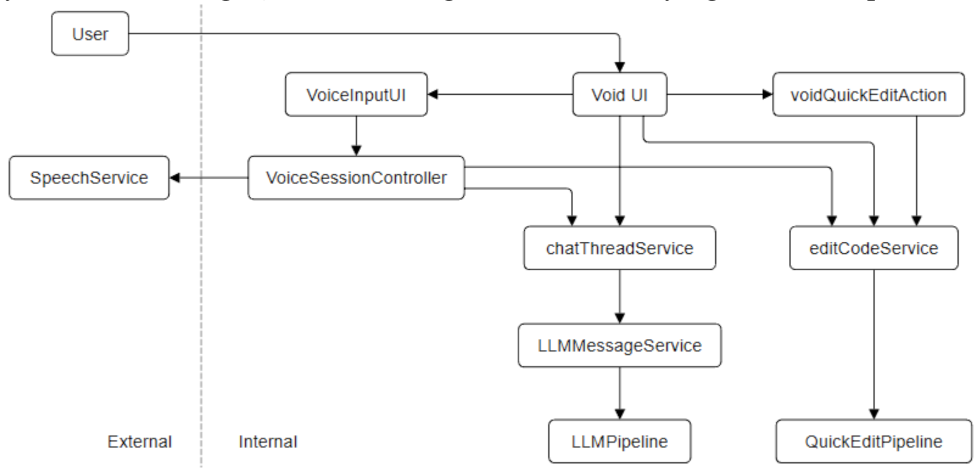
Note: The LLMPipeline "component" is an abstraction of a deeper message-handling flow already described by our Assignment 2 report. In the full architecture, ChatThreadService sends and receives messages through components convertToLLMMessageService, LLMMessageService, and Void Platform's IPC, ultimately to the external AI Provider. Representing all of these individually would create a sequence diagram too long to fit meaningfully on a single page. Additionally, its interactions were already extensively analyzed in our previous assignment sequence diagram. The goal here is to focus on the flow of our new feature while emphasizing that it fits cleanly into the existing architecture without changes to the established LLM messaging pipeline. Therefore, LLMPipeline groups these lower-level components into a single participant to demonstrate where our new voice layer connects. This abstraction proves developmental reuse while keeping the diagram focused on the integration introduced by our voice-enabled feature.

Use Case 2: Voice Quick Edit



This time, the user's goal is to select code and perform edits by speaking a prompt request into the microphone, transcribed into text and executed through the existing Quick Edit pipeline. The use case begins by selecting code and pressing Ctrl+K, which triggers Void UI to call editCodeService's function `addCtrlKZone()`. Like in Assignment 2 but abstracted, this creates a diff zone for the selected lines and refreshes the editor UI with `refreshCtrlKInputs()`. Here, voice controls also become available through a clickable microphone icon which is handled by the previously mentioned VoiceInputUI component. It instructs the VoiceSessionController to start a new voice session for the active CtrlKZone so while the user is speaking, the UI is periodically updated with audio duration. When clicking the mic a second time, VoiceInputUI sends the final audio to VoiceSessionController, which calls `transcribe(finalAudioBlob)` for SpeechService to convert into text.

Upon return, VoiceSessionController invokes `callBeforeApplyOrEdit()` with the generated quick edit transcript prompt and the associated CtrlKZone for editCodeService to process. From this point onward, our diagram intentionally abstracts internal LLM processing by naming a new "component" QuickEditPipeline. It represents the entire Quick Edit flow from Assignment 2 including message preparation, provider communication, diff generation, and UI updates. Therefore, the diagram demonstrates architectural reuse while remaining manageable and clear to read. Finally, the user confirms the changes which triggers Void's existing Apply System application path. editCodeService instructs QuickEditPipeline to either use fast or slow apply for the diffs. Once completed, the updated code can be viewed in the editor. Altogether, our new voice feature introduces only three new components (VoiceInputUI, VoiceSessionController, SpeechService) and this diagram emphasises that they fit cleanly into the existing Quick Edit design without modifying its core responsibilities.



In this box-and-arrow from the sequence diagrams, LLMPipeline and QuickEditPipeline are abstractions of the lower level message processing and apply mechanisms. As explained in each use case, representing every internal subcomponent would clutter this diagram, so they are combined into single conceptual blocks to highlight how the new voice layer integrates with existing functionality.

Impacted Directories

Our enhancement only requires a small number of file-level changes, which are all localized to the existing Void workbench and platform layers. To align with Void's current folder organization, new UI and workflow logic are added under the main browser folder while SpeechService is introduced as an external platform service following the same separation already used for LLM providers. Just one subfolder is required for both VoiceInputUI and VoiceSessionController code files. Referring to the conceptual and concrete architecture, required modifications are listed below.

1. **Workbench Layer (UI and Workflow)** in `src/vs/workbench/contrib/void/` contains Void's chat UI, quick edit interfaces, and LLM triggers [6]. Besides modifying existing files in `/browser/`, we introduce a new subfolder `/voice/` to store voice interaction logic.
   - `.../browser/sidebarPane.ts` holds the rendering so it is modified to add a microphone button entry point inside the existing sidebar UI.
   - `.../browser/sidebarActions.ts` and `sidebarChat.tsx` is modified to register the "Start Voice Input" action and forward UI events into the voice layer.
   - `.../browser/voice/voiceInputUI.ts` will be a new file that displays a recording indicator, timers, and the transcript preview. It is a small UI helper that corresponds directly to our added VoiceInputUI component.
   - `.../browser/voice/voiceSessionController.ts` will be a new file that coordinates starting and stopping in a sessions lifecycle. Maps to the

VoiceSessionController component and calls SpeechService to route transcripts either to ChatThreadService or EditCodeService.
  - `.../browser/editCodeService.ts` is slightly modified to accept transcript text prompts via callBeforeApplyOrEdit() so the existing Quick Edit pipeline can be reused without introducing new responsibilities.
  - `.../browser/quickEditActions.ts` is modified to add voice activation logic to Ctrl+K zones. Its related file in `.../react/quickEditChat.tsx` is modified to provide microphone button logic.
2. **Platform Layer: Speech (External Boundary)** like Void's other current external integrations, the folder `src/vs/platform/` would introduce a new subfolder `/speech/` to contain all Text-to-Speech and Speech-to-Text logic. The most popular companies provide both features as one API so a file can be dedicated to each.
  - `.../common/speechService.ts` new file defines the SpeechService STT and TTS interface used by VoiceSessionController. This isolates provider-specific details from the workbench layer and preserves the same layered separation seen in the concrete architecture.
  - `.../node/azureSpeechService.ts` is where the new default STT and TTS implementation using Azure AI Speech is located. It mirrors how Void already organizes LLM provider binding components (`src/vs/platform/llm/`) and keeps external service logic outside the UI.
3. **Regarding LLMPipeline and QuickEditPipeline**, both remain unchanged and are reused. Their concrete implementations span existing files `convertToLLMMessageServices.ts`, `sendLLMMessageChannels.ts`, and `chatThreadService.ts`, all of which require negligible or no edits at all for our enhancement. As mentioned before, the new voice layer connects before these pipelines to supply transcripts without altering their internal behaviour.

Potential Risks

Due to the isolation from other components, our enhancement can reduce risks affecting NFRs, such as performance and maintainability. On the other hand, it does increase notable risks in reliability and security. Text-based interaction with AI is generally very reliable, with the only bottleneck being the model itself. With our addition, two new factors are introduced: hardware interaction and a second external component. Depending on usage environments, accents, hardware used, and the model being used, the reliability of the technology can vary. In software like Void, using STT could cause accidental usage of things like quick edit, voice commands, or agent mode, which could frustrate users or cause serious damage to their projects.

The second risk stems from the sensitivity of passing audio files to an external provider. This can bring up issues such as interception of audio that might contain sensitive information, for example, something in the background. This could bring complications for implementing this feature in the safest way possible, with the number one goal of protecting user data. There is also a small performance risk because STT requires passing audio through Platform and external providers, which is another step in the pipeline. Architecturally, introducing a new subsystem increases the number of components that must be maintained and tested, which leads to more failure points over time.

Conclusion

Our voice interaction layer takes Void to another level, it keeps its architecture but significantly enhances usability. By passing all audio inputs through STT and then converting them into plain texts to re-enter the LLM pathways, the enhancement integrates seamlessly with the existing design without changing the core message processing subsystems. The two possible implementations, static and real-time, exhibit contrasting performance, complexity, and user experience trade-offs, but both can be hosted by the proposed architecture with only a few discrete component changes. Throughout the studies, we observed that the impact on maintainability, evolvability, and testability of voice enhancements is contained since new voice components constitute a new layer with explicit bounds. In fact, risks such as reliability and privacy mainly originate from audio processing and third-party STT/TTS services, but are principally manageable with careful design and isolation. Ultimately, the proposed feature enhances Void's usability and elasticity while dictating no change to its structure, offering a simple yet realistic way for extensions to expand its capabilities.

Lessons Learned

This report gave our team a perspective on how new features get added to existing architecture without compromising its core. Constructing the voice interaction layer as a separate set of components that first turns audio to plain text before entering any LLM pipeline not only helps us avoid adding unnecessary layers of complexity, but also guarantees our Message path and Quick Edit paths remain fairly immune to stability issues. We also discovered that while a feature may seem straightforward at the user level, its architectural implications can be surprisingly profound. Routing through audio, the different STT/TTS processing, and the new session life cycle raised practical issues regarding reliability, privacy, and performance. These challenges highlight the importance of testing non-functional requirements early, particularly when external services or hardware are involved.

Another significant takeaway was the benefit of contrasting multiple architectural choices. It was also apparent to us that the static vs. runtime computation approach lets you see how trade-offs can be about maintainability, testability, and evolvability. Understanding how various methods of implementation affect a feature made us realize the value of architectural decision-making outside of code-level concerns. This demonstrated the need for clear team communication. Aligning diagrams, use cases, and subsystem descriptions pushed the team to refine terms and keep terminologies in sync. As we observed in A2, the architecture becomes significantly easier to analyze when diagrams, modules, and descriptions of what's going on align neatly. In total, this project helped us learn to think about improvements systematically, balancing design imperatives and technical limitations.

AI Collaboration Report

During Assignment 3, our team continued using OpenAI's ChatGPT-5 as an AI partner for planning, clarification, and quality control. ChatGPT performed well in previous assignments, especially when clarifying architectural ideas and refining explanations. It made sense to use the same AI teammate with an existing chat history that everyone was familiar with. The resulting report, in contrast to A2, consists of suggesting a new architectural improvement, comparing two alternatives, and evaluating their impact on NFRs and stakeholders. Given this broader context, the role of AI changes from correcting grammar and structuring to serving deep architectural reasoning, finding blind spots, and guiding the building of exhaustively detailed SAAMs. Following old procedures, all AI suggestions were carefully reviewed and validated against documentation before being included.

ChatGPT helped define our role of Voice Session Controller (Question and Answer):

When describing the Voice Session Controller, we needed a modular, clear approach to explain what the component is, where it exists in the architecture, and how it fulfills its role with subsystems within Void. We asked ChatGPT how we should refer to such a controller and how to position it to integrate naturally with existing components like LLMMessageService and chatThreadService. Its guidance helped refine our terminology and keep explanations consistent with the vocabulary used throughout the report. This was a suitable task for the AI because naming and boundary definitions benefit from comparing multiple architectural patterns, which ChatGPT was well-suited for. It also helped us confirm that introducing this controller would not conflict with existing responsibilities in the message pipeline, which kept our final design as coherent as possible.

ChatGPT discussed architectural placement decisions for STT (Question and Answer):

A critical design choice in A3 is determining where the STT component should operate, at the UI level or the main (Void Platform) process. We had ChatGPT describe common architectural concerns relevant to this choice, like fault isolation, latency, IPC overhead, and security boundaries. Its advice helped us form comparisons and trade-offs between the two alternatives. We then cross-validated all claims with documentation and concepts from the course. This was particularly valuable because architecture placement often involves subtle interactions between processes that can be easy to overlook. ChatGPT also helped us identify which risks were specific to static versus real-time systems, so our analysis stayed within the proposed constraints.

We asked ChatGPT about common integration problems faced in an additional input modality to a system similar to an IDE (Question and Answer):

Adding voice as an input method entails modifying existing systems such as chat UI, Quick Edit pipeline, command handlers, and editor surfaces. To better understand risks in these interactions, we prompted ChatGPT to identify which areas of an IDE typically require extra caution after integrating voice support. The response helped us craft testing approaches and determine interaction points that might have otherwise gone unnoticed. Finally, it highlighted small edge cases such as conflicting triggers or ambiguous input contexts that we eventually incorporated into our evaluation of potential failure modes.

Interaction Protocol and Prompting Strategy

We consulted ChatGPT for clarification on concepts like the component boundaries, STT placement, and integration concerns of the two architecture alternatives that team members worked on when we felt challenged. We also posted the conversation link in a group chat for other members to see or reuse prompts. Furthermore, all prompts were refined iteratively, especially when we needed the AI to stay within static STT assumptions instead of drifting into real-time scenarios. This collaborative prompting approach meant that members could build on each other's insights rather than starting from scratch for every question.

Quality Assurance and Validation Procedures

Our team manually verified every proposal created by AI using both the class materials and Void Architecture of A1/A2, as well as the components presented for A3. If the ChatGPT responses were either too generic or inaccurate, we would enter the correct information into the document using the documentation and team discussion. Also, our team reviewed ChatGPT's grammar edits to ensure we maintained the same tone throughout. We compared some of its suggestions with alternative outputs from Gemini or DeepSeek to ensure our conclusions were not biased toward a single tool. Finally, for architectural claims, at least two team members cross-checked the AI reasoning to ensure consistency with our enhancement strategy.

Quantitative Contribution of AI to Final Output

Although ChatGPT did not directly write the text for this project, it contributed to accelerating the outline process, making wording suggestions, and supporting some of the architectural reasoning involved in the project. Most of the value came from using AI to quickly compare architectural alternatives, which reduced the time spent debating initial design directions. Based on these contributions, we estimate that the total contribution of ChatGPT to the final output was approximately 9%, primarily due to its effectiveness in supporting the initial planning stages and clarifying ideas prior to writing our report. Its impact was helpful but limited, since all architectural decisions, diagrams, and written sections were ultimately produced and verified by the team.

Reflection Upon Human + AI Team Dynamic

ChatGPT was more useful for the beginning of each task, primarily for filtering out potential ideas before writing. We also noticed that this AI was most effective when we provided strict architectural constraints to prevent overly general answers. ChatGPT's recommendations significantly improved our team's workflow, but it still required diligent review to ensure group members didn’t assume false premises. Throughout, the team has maintained a culture of open communication and collaboration through creating team-wide logs of chat communications. In summary, ChatGPT is a useful tool for planning and assisting human judgment, provided it is used appropriately. This helped us better understand the limits of AI support and reinforced the importance of human oversight in large architectural decision-making.

References

[1] Arık, Sercan Ö., et al. "Deep voice: Real-time neural text-to-speech." *International conference on machine learning*. PMLR, 2017. Deep Voice: Real-time Neural Text-to-Speech.

[2] A. Kumar, “Void IDE: The Comprehensive Guide to the Open-Source Cursor Alternative,” *Medium*, Mar. 24, 2025. Available: https://medium.com/@adityakumar2001/void-ide-the-comprehensive-guide-to-the-open-source-cursor-alternative-2a6b17cae235.

[3] Microsoft Azure, “Azure AI Speech,” *Azure AI Foundry*, 2025. Available: https://azure.microsoft.com/en-us/products/ai-foundry/tools/speech.

[4] Void Editor, “Void Official Website,” *Void Editor*. Available: voideditor.com.

[5] “Void: The Open Source Cursor Alternative.” *Y Combinator*, www.ycombinator.com/companies/void.

[6] voideditor, “void/VOID_CODEBASE_GUIDE.md at main · voideditor/void,” *GitHub*, 2024. https://github.com/voideditor/void/blob/main/VOID_CODEBASE_GUIDE.md.

[7] Zread.ai, “LLM Message Pipeline Architecture | Voideditor/Void,” *Zread*, Jun. 25, 2025. Available: https://zread.ai/voideditor/void/10-llm-message-pipeline-architecture.

[8] Zread.ai, “Quick Start | Voideditor/Void,” Zread, Jun. 25, 2025. Available: https://zread.ai/voideditor/void/2-quick-start.