UCS 503 Software Engineering Project Report

End-Semester Evaluation

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Audio-Augmented Python Code Editor

for Blind Developers

Submitted by:

Arshnoor Singh (102317161)

Harshpreet (102317160)

Ansh Madaan (102317159)

BE Third Year, CSE

Group No: 7

Submitted to:
Dr. Deep Mann

November, 2025

Table of Contents

1. Project Selection Phase

1.1 Software Bid

Team Members

Name	Roll No	Project Experience	Programming Language Used	Signature
Harshpreet	102317160	Led the development of a real-time voice-to-code dictation engine integrated into Visual Studio Code. Designed microphone capture pipelines and converted speech to valid Python code using Whisper STT. Implemented quick-action shortcuts and enhanced voice command stability. Optimized responsiveness for blind users with real-time audio cues and minimal latency insertions.	Python, TypeScript	Harshpreet
Arshnoor Singh	102317161	Architected a fully accessible frontend interface using the VS Code WebView API. Focused on usability and assistive navigation through keyboard tabbing, sound cues, and dynamic status updates. Integrated the Web Speech API to deliver contextual spoken feedback. Developed dynamic UI panels for diagnostics, activity logging, and execution control tailored for non-visual workflows.	JavaScript, HTML/CSS	Arshnoor
Ansh Maadan	102317159	Developed and tested core backend logic using the VS Code Extension Host and Node.js. Implemented custom Python syntax validation with fallback diagnostics. Built the Insert Engine to interpret natural-language voice commands and apply safe edits. Contributed to LLM integration planning for future code summarization and query support. Managed file access and workspace automation.	Python, Node.js	Ansh

Tools & Technologies: TypeScript, Node.js, Python, Text-to-Speech Engine (say.js), VS Code Extension API.

Expected Outcome: A fully functional code editor extension that detects syntax errors, classifies them, and delivers real-time feedback through sound cues and TTS narration.

1.2 Project Overview

The system transforms traditional programming into an accessible experience by integrating sound and speech responses for all critical code interactions. The extension detects syntax issues, plays distinct earcons, and speaks error messages, allowing visually impaired users to understand and fix their code efficiently.

Key Modules: Error Detector, Audio Formatter, Speech Synthesizer, Training Mode, and Auto-Fix Engine.

2. Analysis Phase

2.1 Use Cases

Primary Actor: Blind Programmer

Supporting Actors: Language Server, File System

Use Cases: Detect Syntax Error, Read Current Line, Navigate Errors, Summarize Errors, Adjust Verbosity, Run and Speak Output, Start Training Mode.

2.1.1 Use Case Diagram

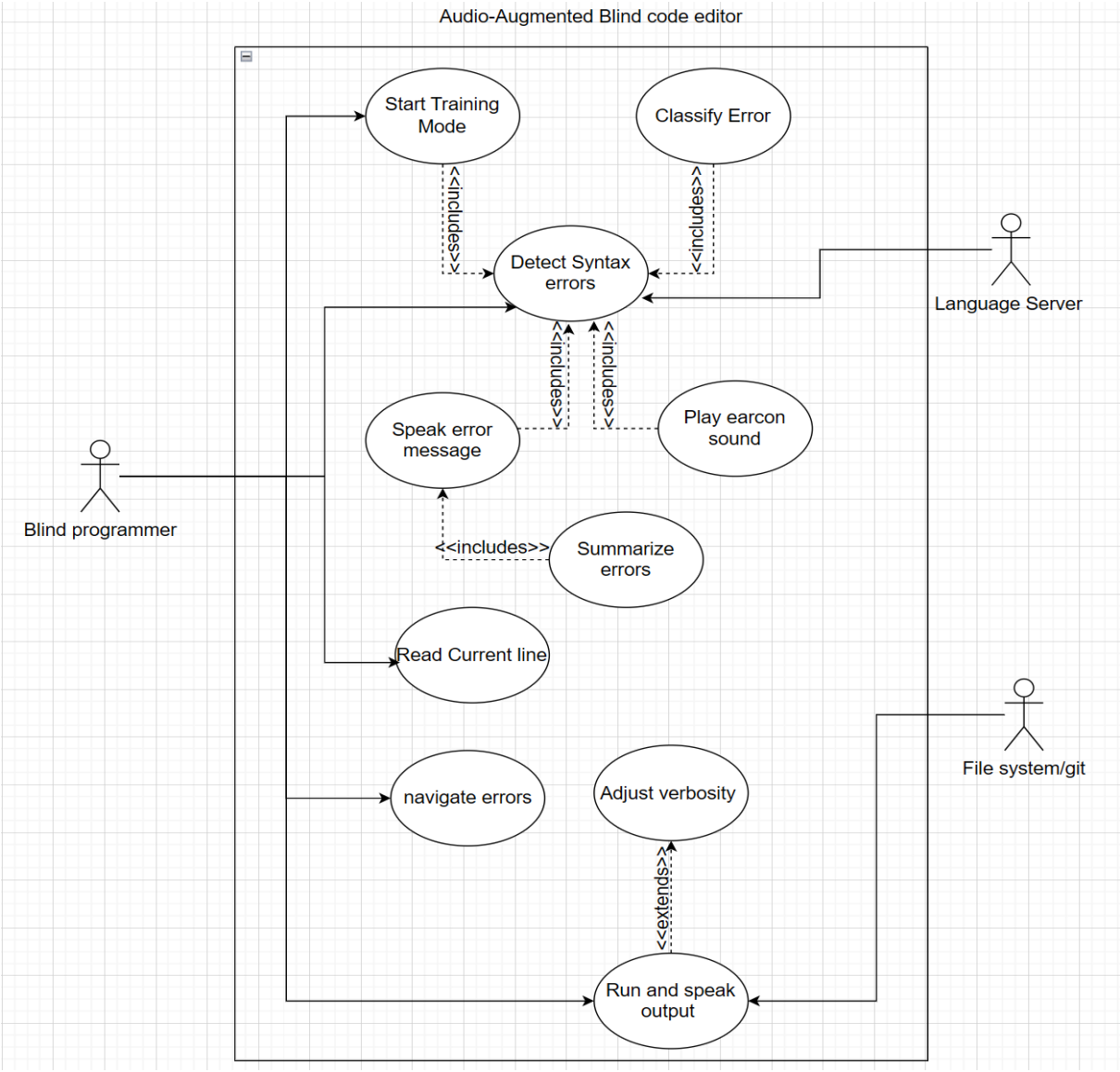


Figure 1: Use Case Diagram

2.1.2 Use Case Template Example

Use Case Specification	
1. Use case title	Audio-Augmented Blind Code Editor
2. Abbreviated title	AABCE
3. Use case ID	UC-1
4. Users	Primary actor: Blind Programmer Secondary actors: Language Server (LSP), File System / Git, Audio Output (TTS / Earcon Engine)
5. Description	The Audio-Augmented Blind Code Editor provides auditory assistance to visually impaired programmers. It detects and classifies syntax errors via a language server, plays earcon cues, reads error messages aloud, and offers navigation, verbosity control, and training features through keyboard or voice commands.
5.1 Preconditions	1. The extension is installed and running in the editor. 2. Language Server is connected and available for diagnostics. 3. Audio output (TTS / earcon) is enabled. 4. Accessibility mode in the editor is active.
5.2 Task sequence	1. Open or edit source file in the editor. 2. Editor Event Listener notifies the Language Server for diagnostics. 3. Language Server returns diagnostics; extension normalizes and classifies errors. 4. On detection, an earcon plays and an error message is spoken. 5. User reads current line or navigates errors via keyboard/voice. 6. User may request an error summary. 7. User can adjust verbosity (persisted in settings). 8. User runs code; runtime output is spoken. 9. (Optional) Start training mode for practice.
5.3 Postconditions	1. Timely auditory feedback for edits and errors. 2. Errors classified, summarized, and spoken. 3. Verbosity preference is saved for future sessions.
6. Alternative flows	Alt A – No errors: play a neutral confirmation tone. Alt B – File operations: confirm save/load via speech. Alt C – Verbosity change: switch minimal / normal / verbose.
7. Non-functional reqs	Accessibility (WCAG), near real-time feedback, reliable earcon mapping, clear TTS clarity, low latency.
8. Data / APIs	Data: diagnostics logs, TTS logs, user prefs. APIs: Language Server Protocol (LSP), File System APIs, TTS engine.
9. Modification history	07-Oct-2025 — Arshnoor Singh — Initial creation
10. Authors / Notes	Author: Arshnoor Singh, Harshpreet, Ansh Madaan Notes: - Consider future haptic integration. - Calibrate earcon set for clarity. - Support multiple IDEs via plugin APIs.

Figure 2: Use Case Template

2.2 Activity Diagram and Swimlane Diagram

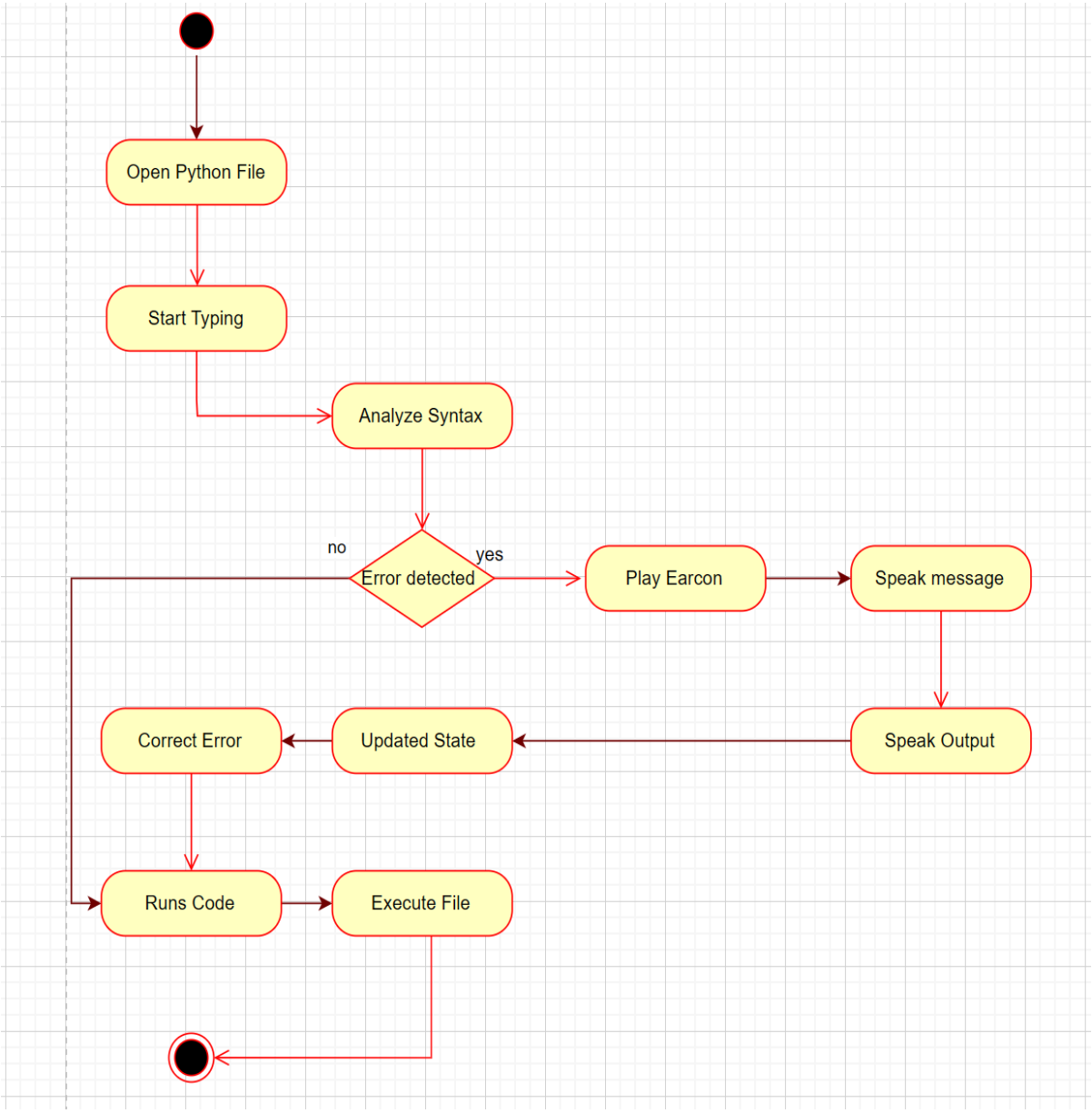


Figure 3: Activity Diagram

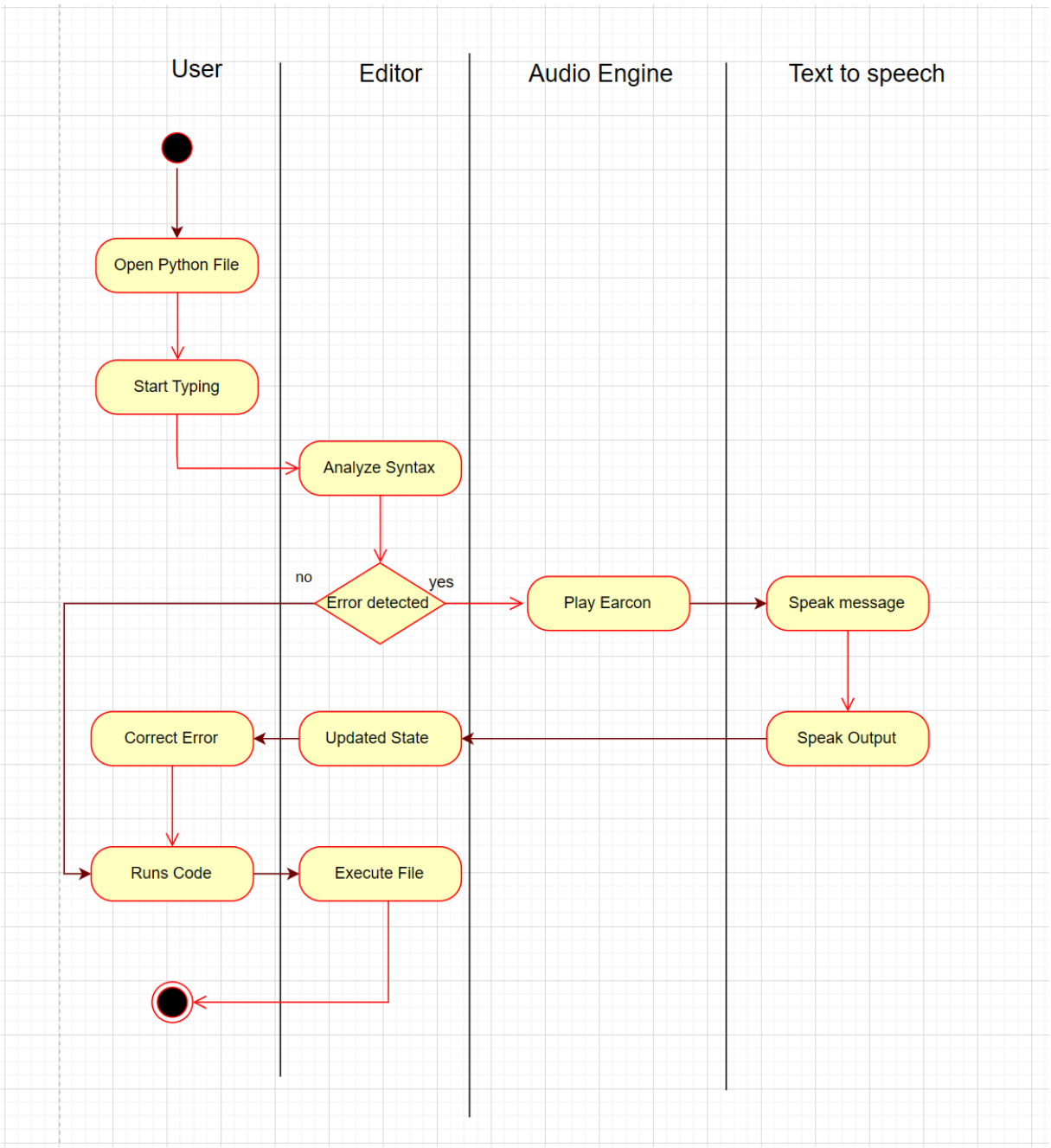


Figure 4: Swimlane Diagram

2.3 Data Flow Diagrams (DFDs)

LEVEL 0:

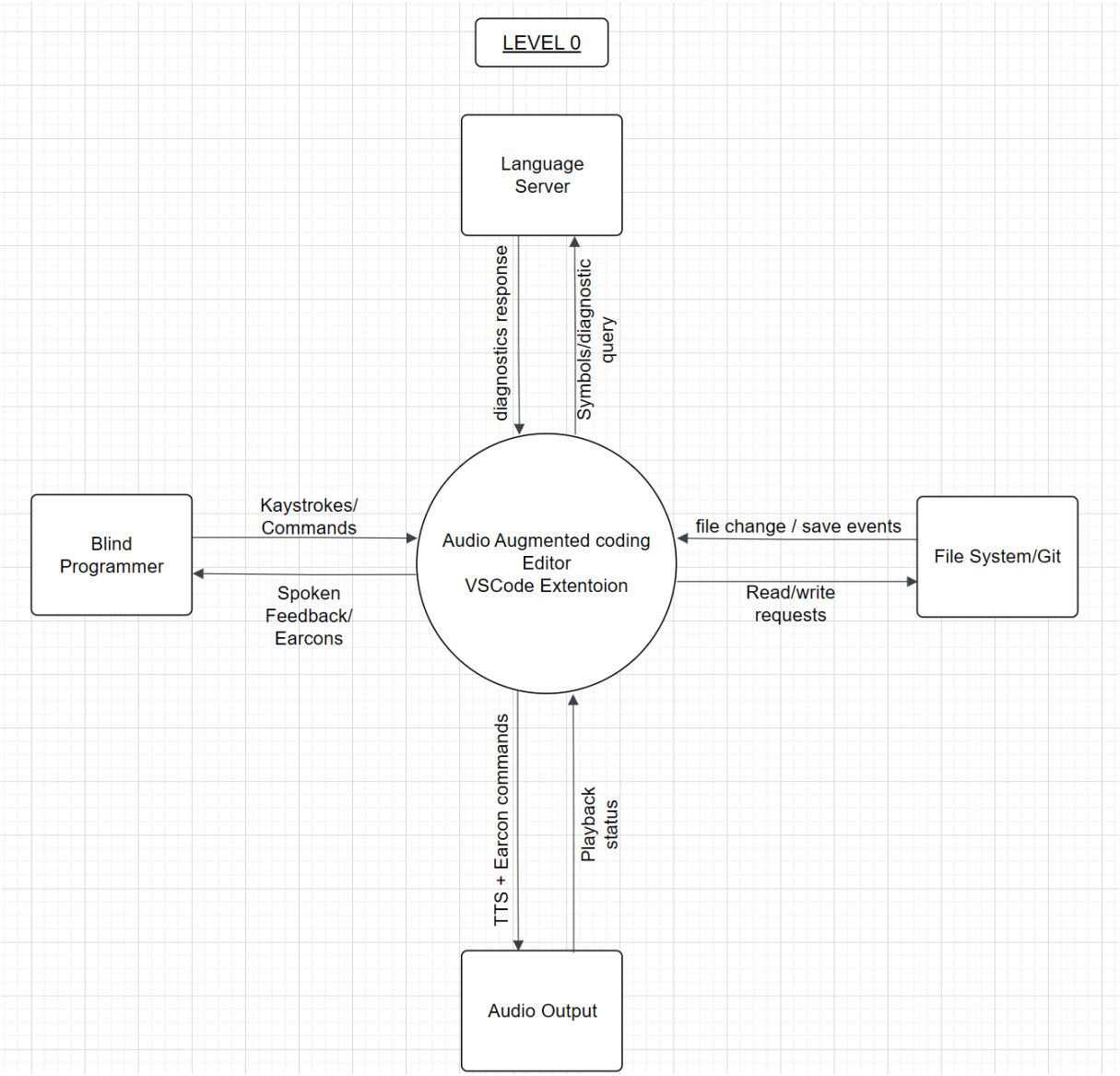


Figure 5: Level 0 DFD

LEVEL 1:

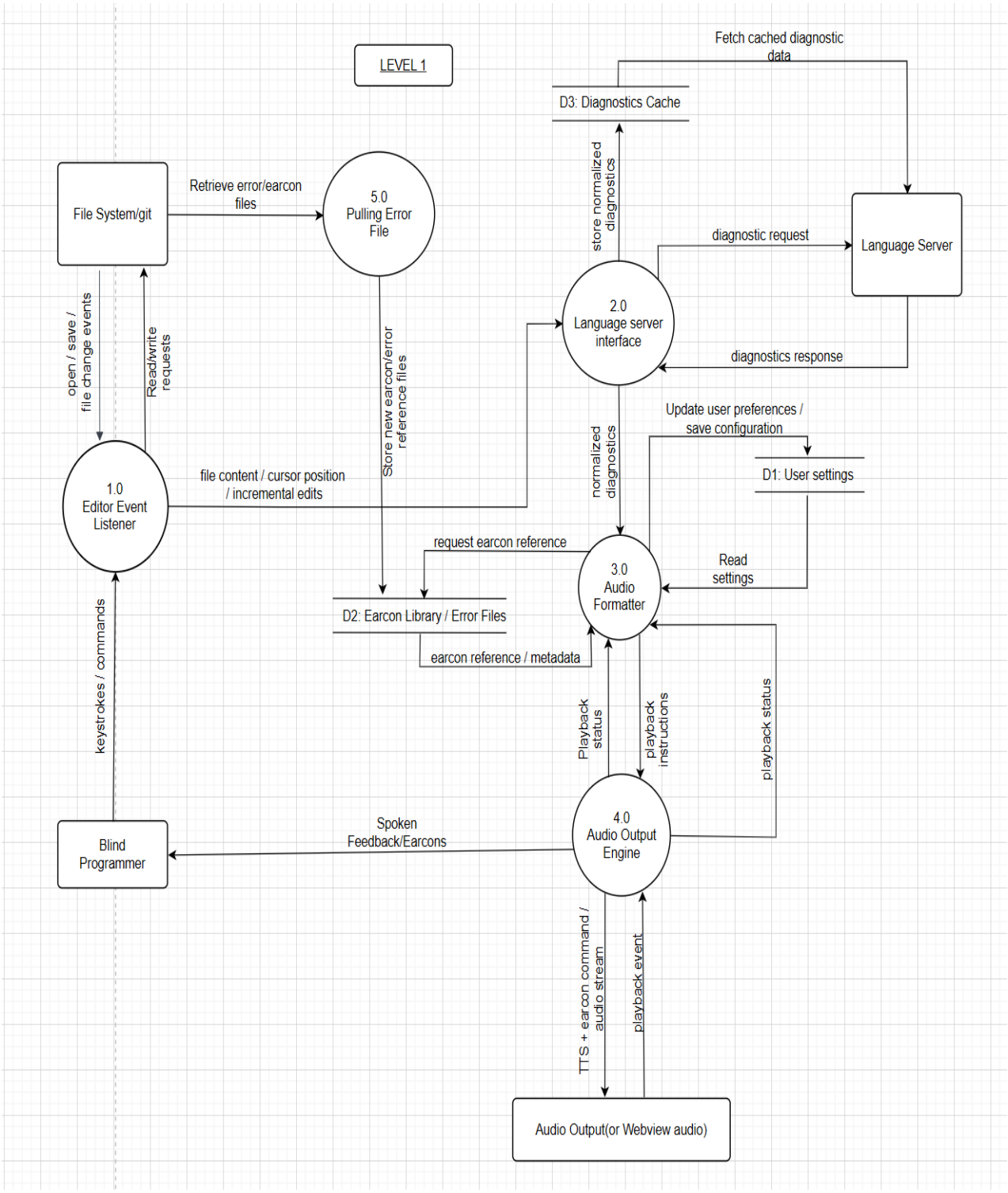


Figure 6: Level 1 DFD

LEVEL 2:

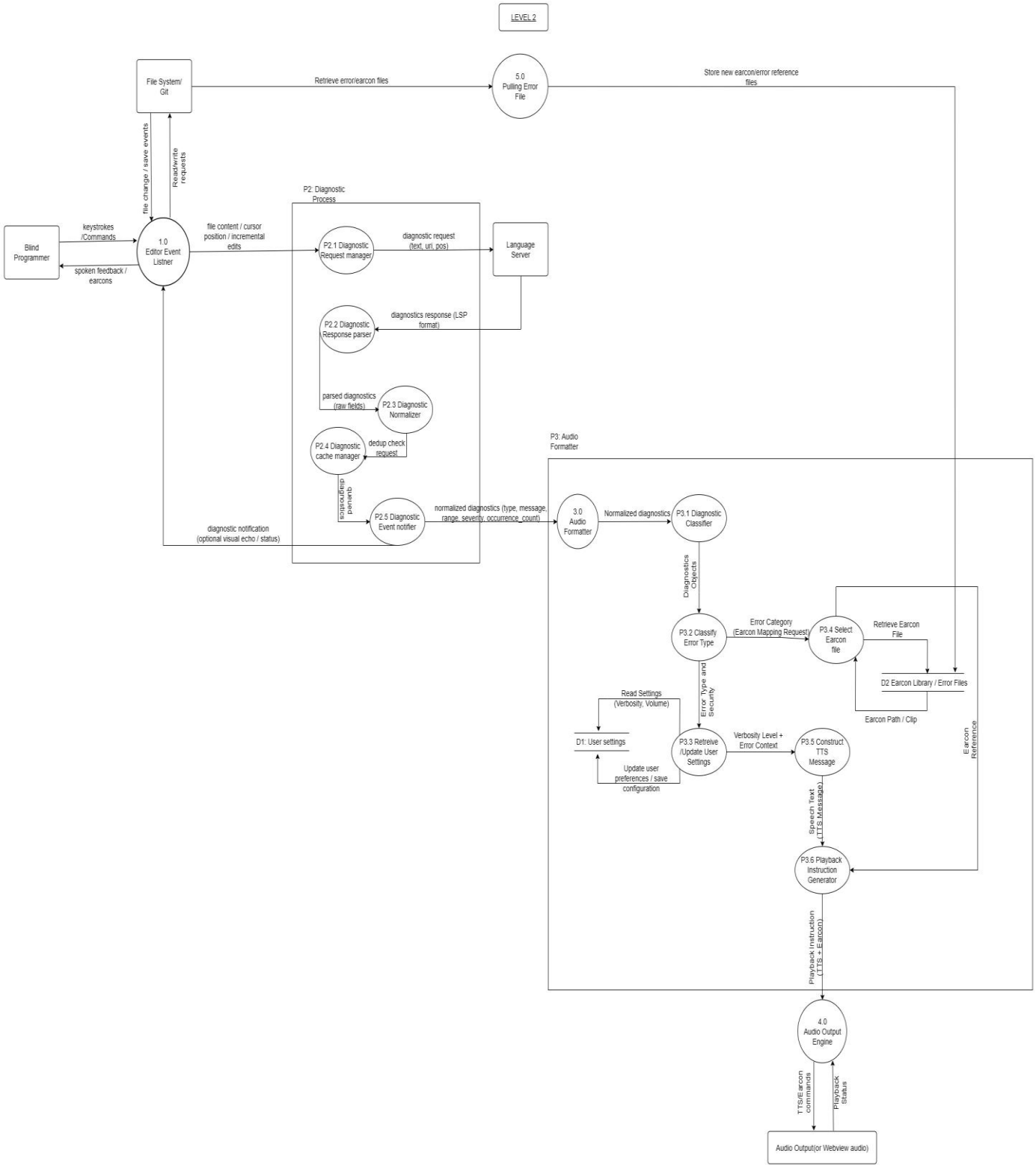


Figure 7: Level 2 DFD

2.4 Software Requirement Specification (IEEE Format)

https://drive.google.com/file/d/1iLN4E5Dgv6urd9NvVBbpzuyXM0JOXb1Y/view

2.5 User Stories and Story Cards

User Story 1: As a blind programmer, I want to hear a sound when I make a syntax error so that I can fix it quickly.

User Story 2: As a user, I want to listen to the current line of code so that I can understand my position.

User Story 3: As a user, I want to toggle verbosity to control the level of speech detail.

User Story 4: As a learner, I want a training mode so I can practice identifying and correcting common errors.

User Story 1 Card:

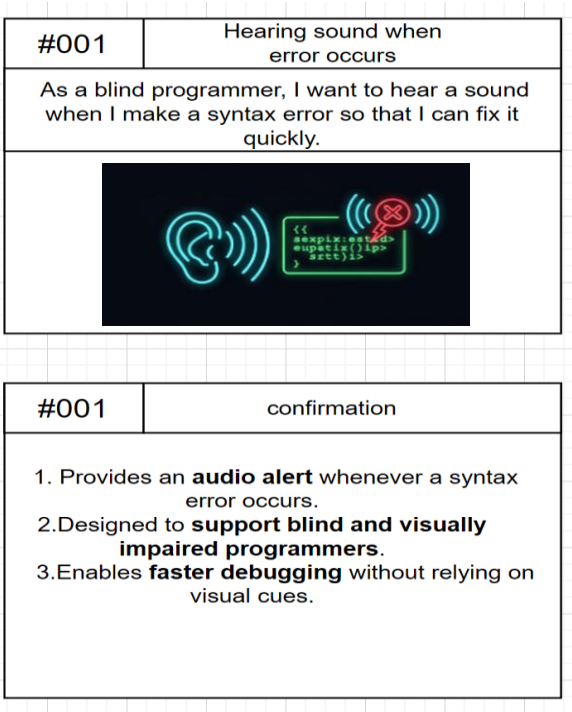


Figure 8: User Story Card 1

User Story 2 Card:

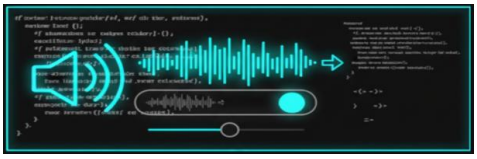
#002	Listen to the current line of code when error occurs
As a user, I want to listen to the current line of code so that I can understand my position.	

#002	confirmation
1.Adds a **voice feature** to read aloud the **current line of code**. 2.Helps users **track their position** within the code editor. 3.Provides **context awareness** without relying on visual indicators	

Figure 9: User Story Card 2

User Story 3 Card:

#003	toggling verbosity to control the level of speech
As a user, I want to toggle verbosity to control the level of speech detail.	



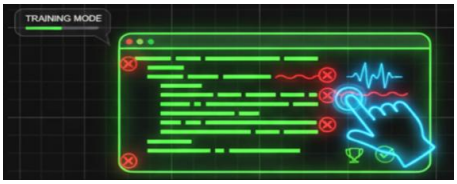
#003	confirmation
1. Allows users to **toggle verbosity levels** for spoken feedback. 2.Provides **control over speech detail**, from brief summaries to full code descriptions. 3.Enhances **personalized accessibility** based on user preference.	

Figure 10: User Story Card 3

User Story 4 Card:

#004	toggling verbosity to control the level of speech

As a user, I want to toggle verbosity to control the level of speech detail.



#004	confirmation

1. Introduces a **Training mode** for practicing error detection and correction.
2. Simulates **common coding mistakes** in a safe, guided environment.
3. Provides instant audio feedback to reinforce learning.

Figure 11: User Story Card 4

3. Design Phase

3.1 Class Diagram

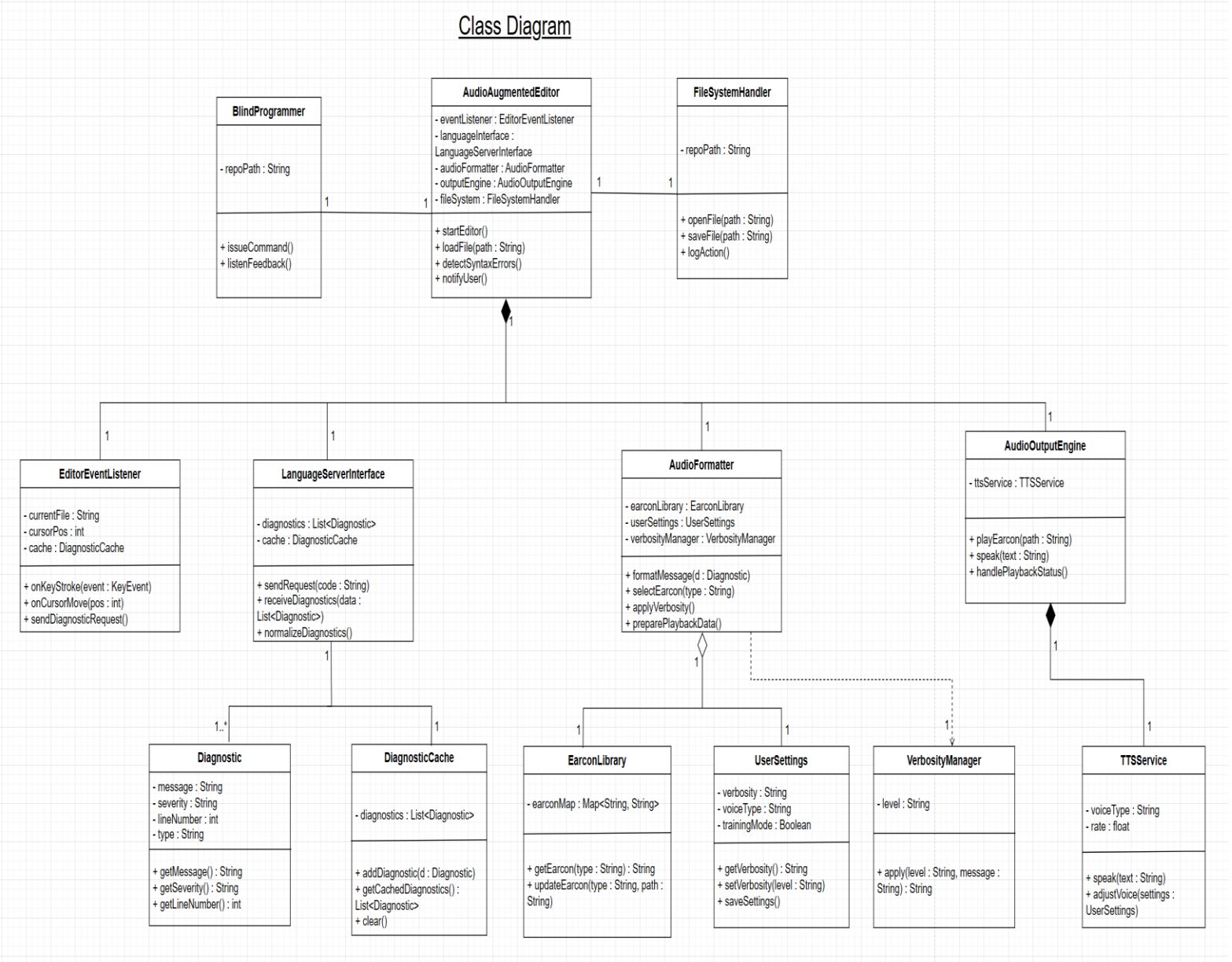


Figure 12: Class Diagram

3.2 Sequence Diagram

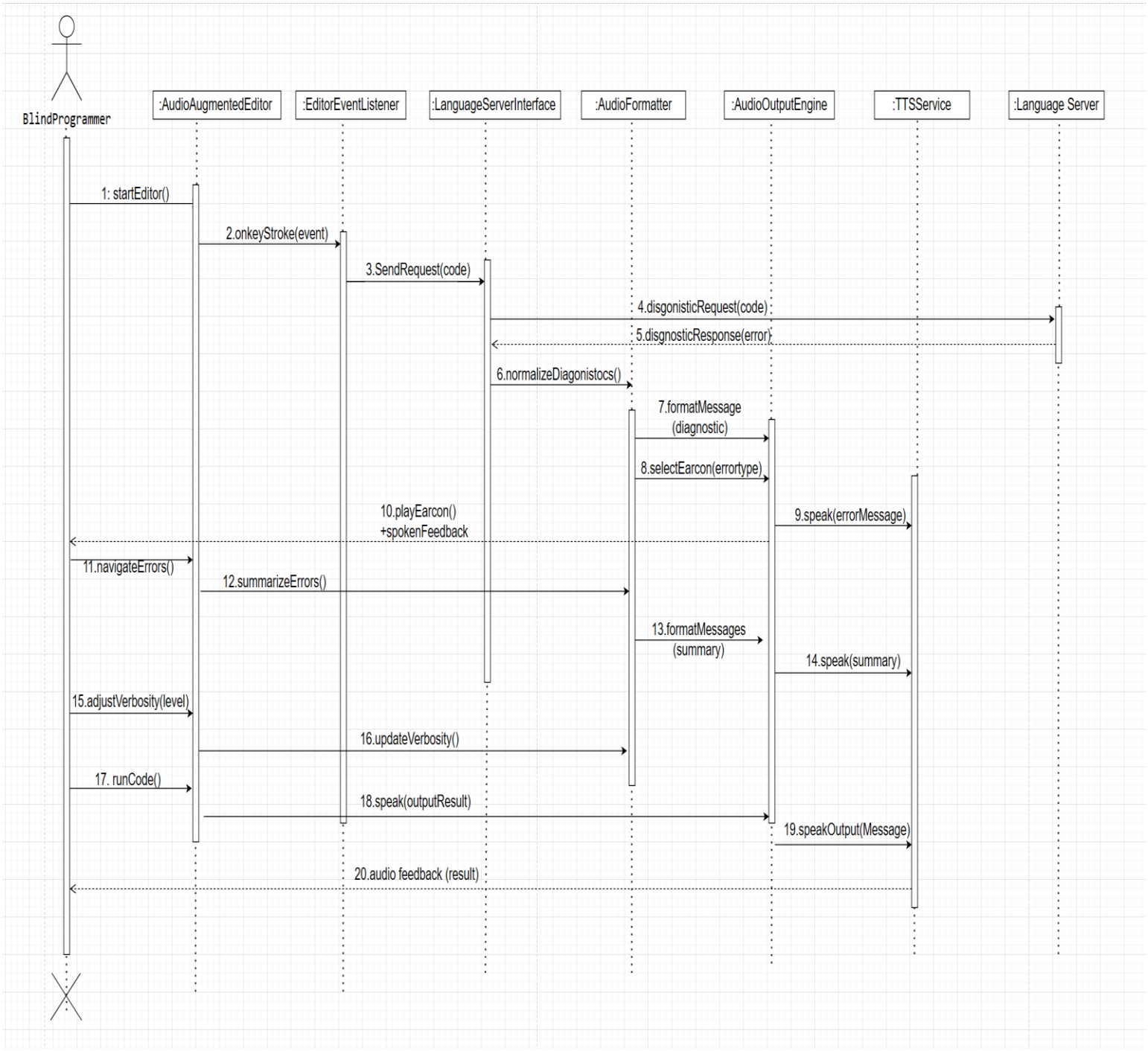


Figure 13: Sequence Diagram

3.3 Collaboration Diagram

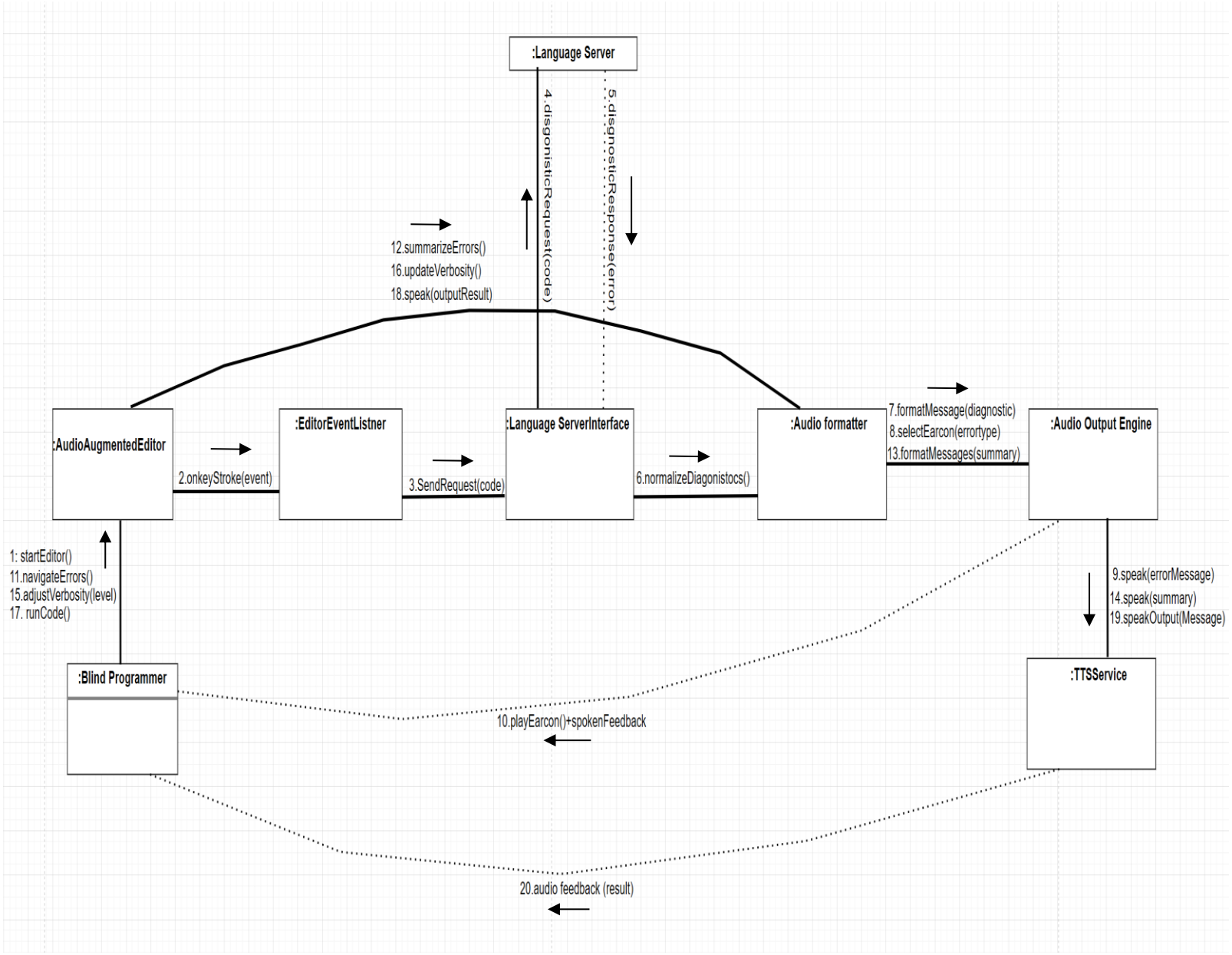


Figure 14: Collaboration Diagram

3.4 State Chart Diagram

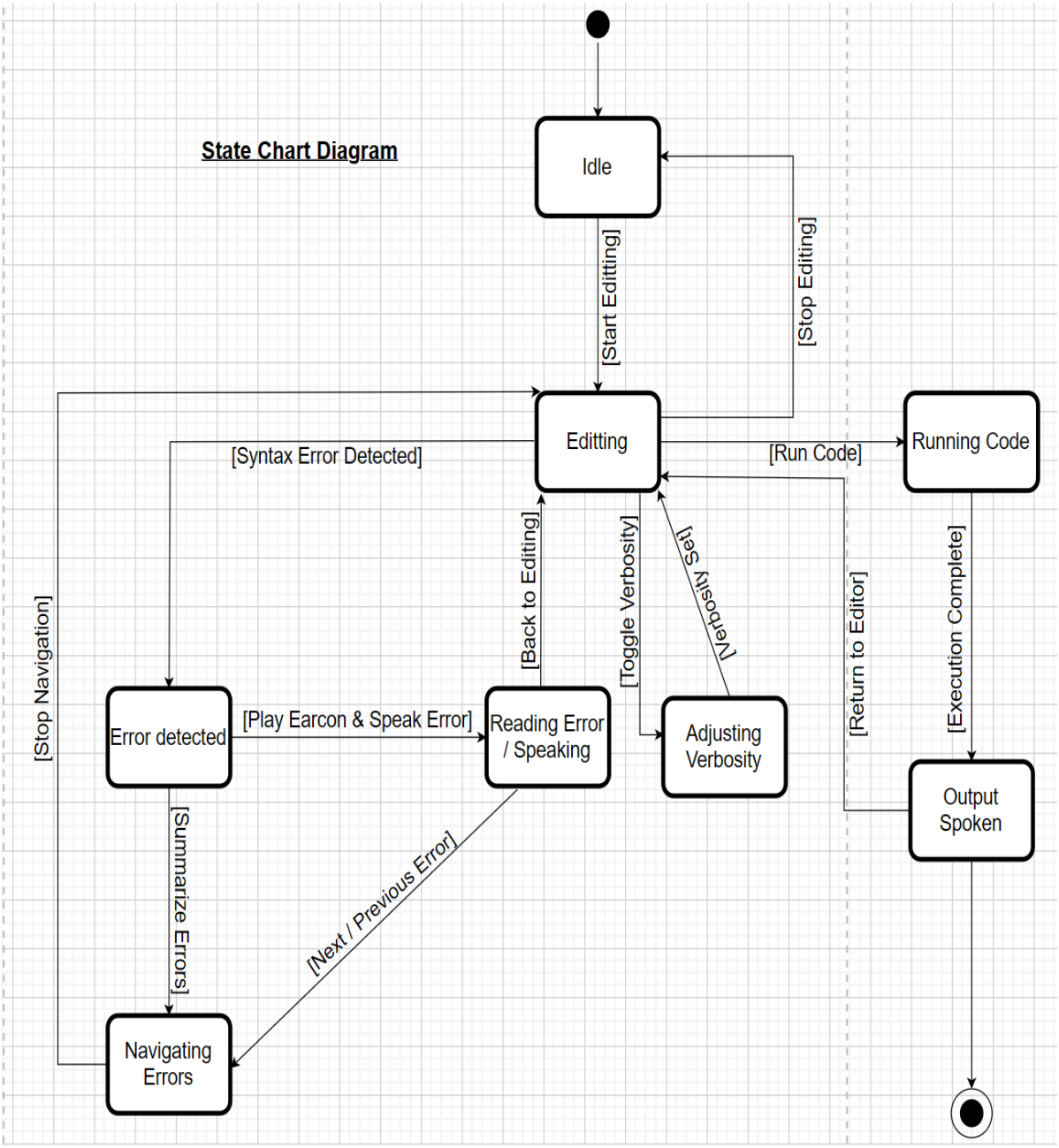


Figure 15: State Chart Diagram

4. Implementation

4.1 Component Diagrams

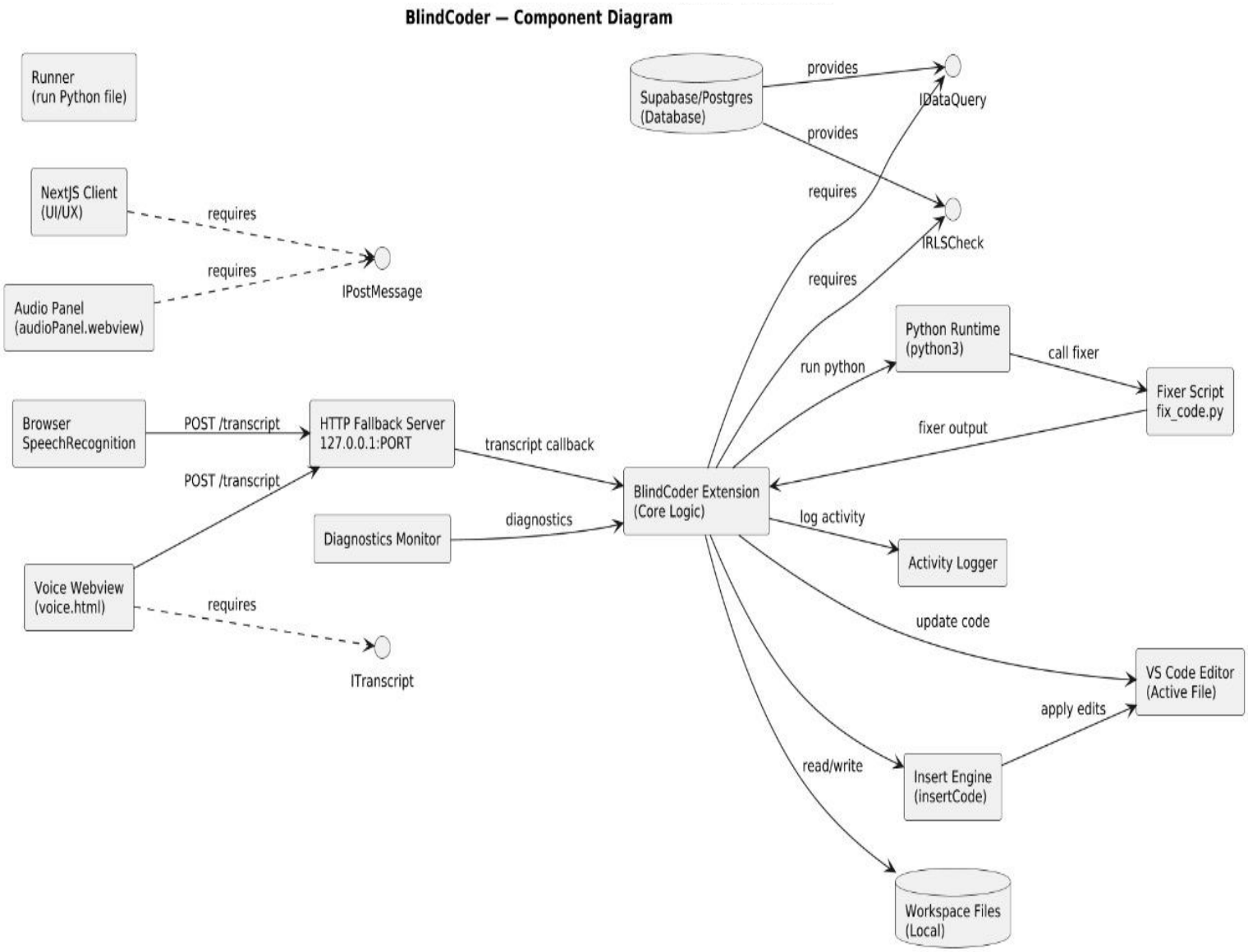


Figure 16: Component Diagram

4.2 Deployment Diagrams

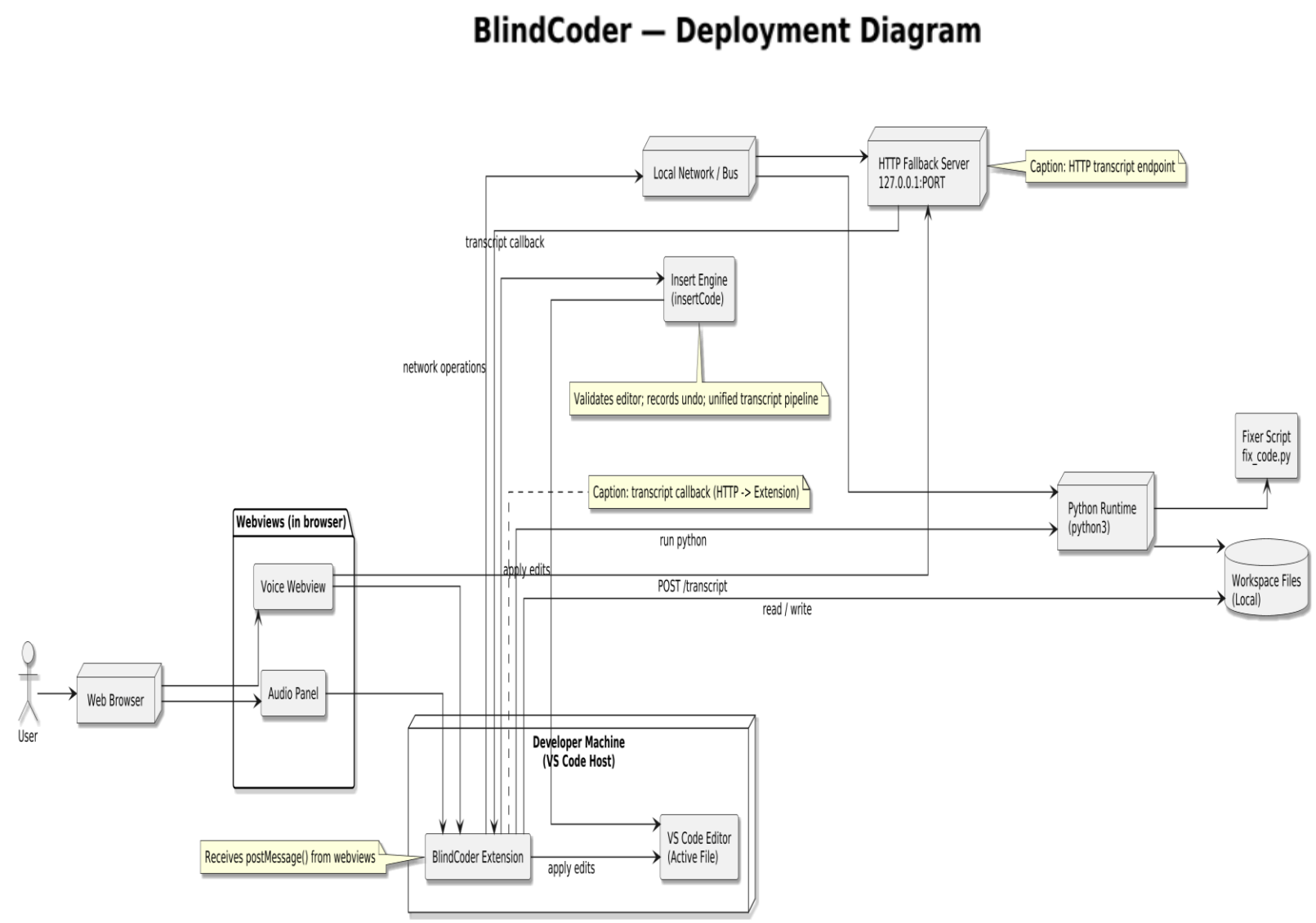


Figure 17: Deployment Diagram

4.3 Screenshots

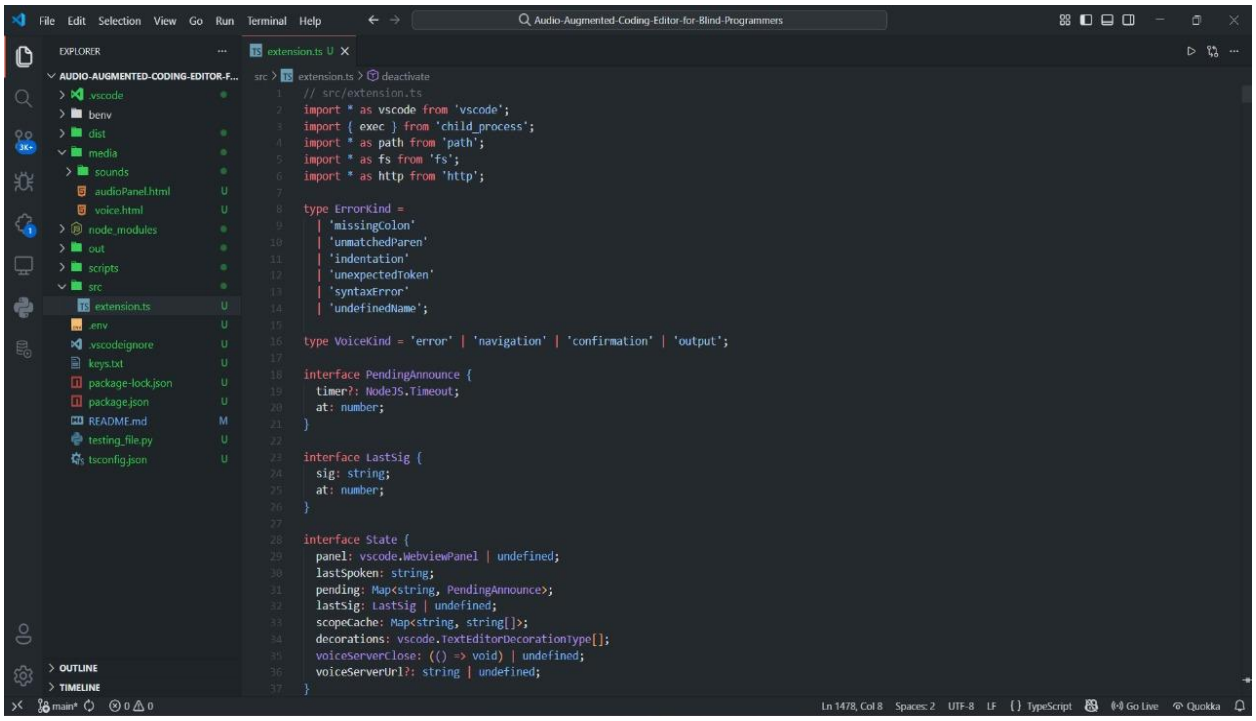


Figure 18: Screenshot 1

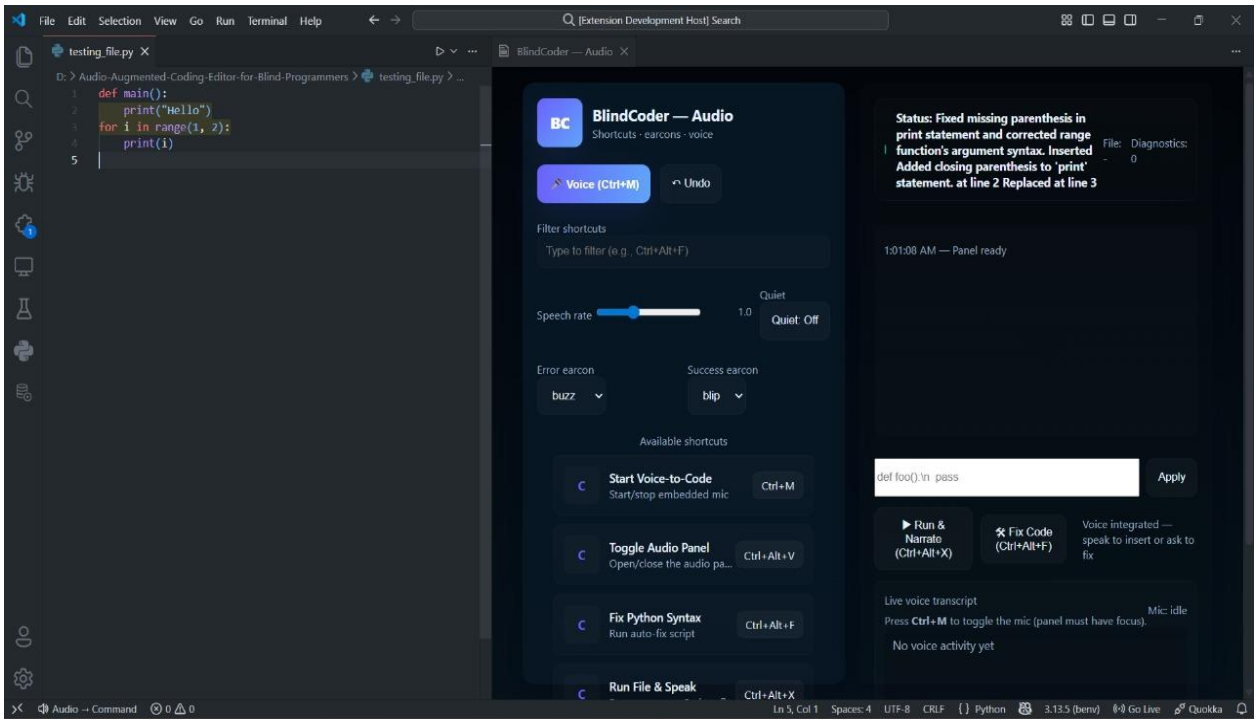


Figure 19: Screenshot 2

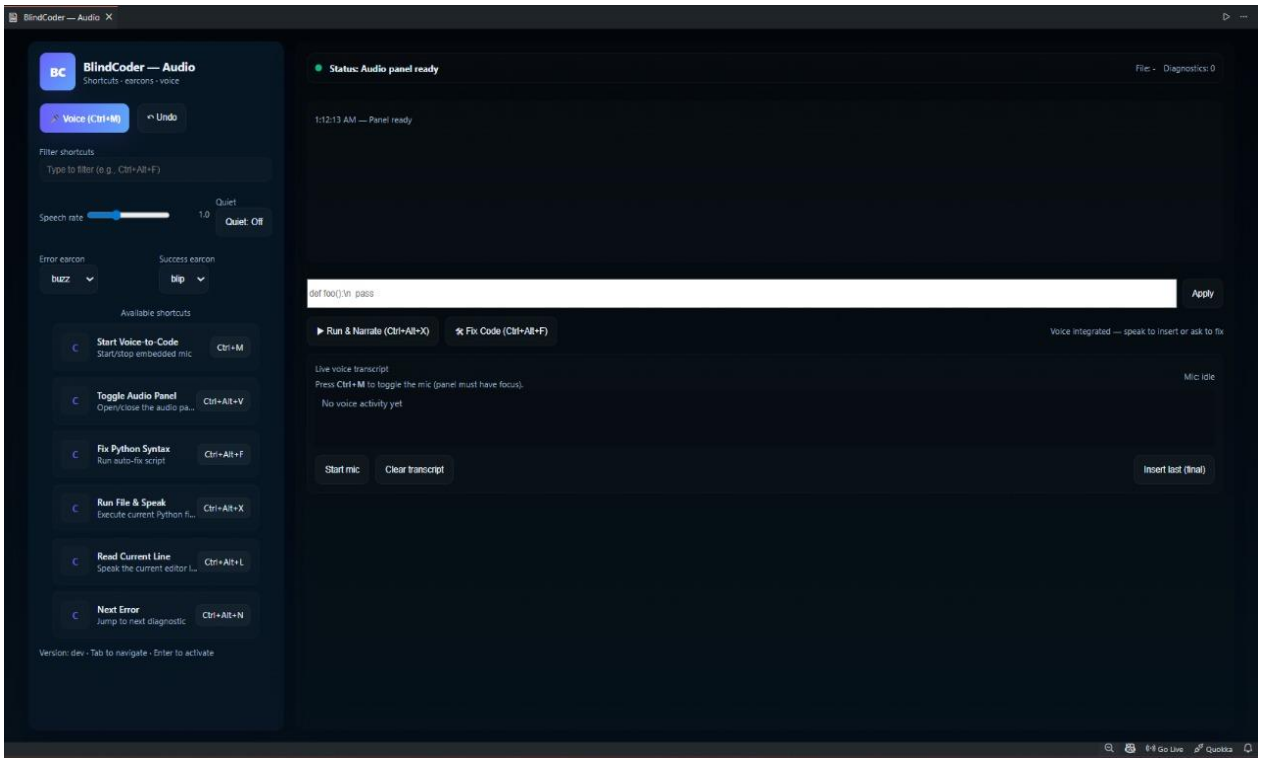


Figure 20: Screenshot 3

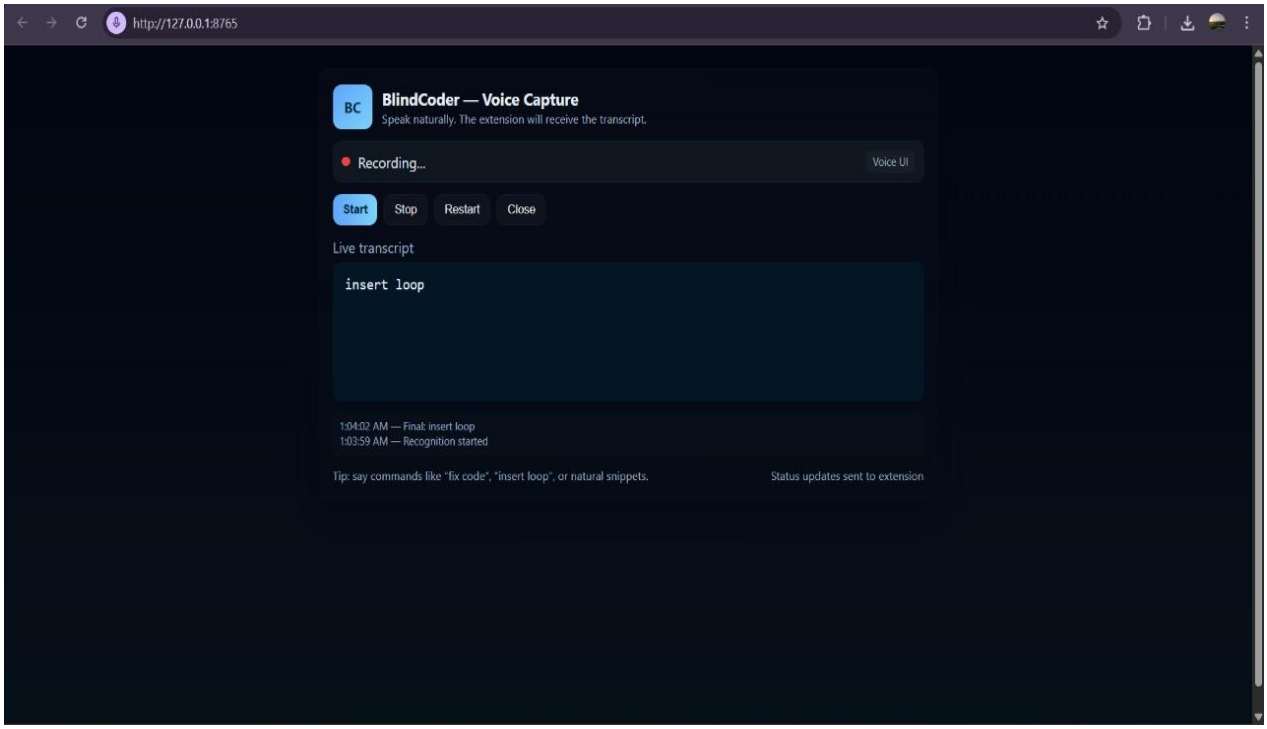


Figure 21: Screenshot 4

5. Testing

5.1 Test Plan

The test plan defines the purpose and scope of testing for the BlindCoder VS Code extension. In this plan we identify objectives, features to be tested, and testing constraints. The purpose is to verify that BlindCoder's audio-augmented features work correctly and reliably[1]. We outline an executive summary of what the testing will accomplish and the limits of the effort (e.g. environments, resources)[2].

- **Purpose & Scope:** Ensure the extension's voice-driven coding and navigation features meet design requirements. The scope includes functional testing of all major workflows (voice insertion, execution, fixing, navigation) and basic non-functional checks (performance, responsiveness under typical conditions). We will also verify compatibility with assistive technologies (screen readers, TTS)[1].

- **Features/Workflows to Test:**

- *Voice-to-Code:* Capturing spoken commands and inserting code snippets (e.g. "print hello" → print("hello")). Voice recognition uses the Web Speech API, allowing natural-speech coding (as demonstrated by tools like Serenade[3]).

- *Code Execution:* Running code from the editor and narrating the output or errors back to the user using text-to-speech.

- *Syntax Fixing:* Automatic detection and correction of syntax errors. For example, missing punctuation or misspelled keywords should be flagged and fixed, with changes announced via audio.

- *Navigation/Debug:* Moving through code (jump to definitions, find references, next/previous function), and reading contextual information (e.g. reading the current line or error messages). This includes debugger integration (announcing breakpoints, debug state changes).

- *UI/UX Accessibility:* Keyboard navigation and shortcuts, screen-reader optimized views, and any custom UI (menus, status bar, output panels) must be accessible. VS Code provides many accessibility features (e.g. Accessible View, screen-reader mode[4]), which we will test.

- **Technologies Involved:** The BlindCoder extension is a standard VS Code extension built with Node.js and TypeScript (the package.json contains Node.js fields such as scripts and devDependencies[5]). The extension logic runs on the Node.js runtime, a cross-platform JavaScript environment[6]. For voice features, it leverages the browser-based Web Speech API: the SpeechRecognition interface for capturing speech and the SpeechSynthesis interface for outputting speech[7][8]. Python scripts or libraries may also be used for auxiliary tasks (e.g. advanced text processing or AI assistance), so Python 3.x should be available.

- **Testing Environments & Tools:** Testing will be performed on multiple operating systems (Windows 10/11, macOS, Linux) since Node.js and VS Code are cross-platform[6]. VS Code should be the latest stable version. Key tools include: a screen reader on each OS (NVDA or JAWS on Windows, VoiceOver on macOS, Orca on Linux) to verify accessibility[4]; Node.js (latest LTS) and npm for building the extension; Python 3 for any scripts; code coverage tools (e.g. Jest, Mocha) for unit tests; and accessibility auditing tools (e.g. Axe for UI elements). Audio recording software can help capture spoken output during tests. We also use standard VS Code testing support (the Extension Test Runner) and manual exploratory testing. Regression tests can be automated via CI (GitHub Actions or Azure Pipelines) on each supported platform.

• Additional Considerations

- **Risk Assessment:**
  - Speech recognition may behave inconsistently due to accents or noise.
  - TTS performance may vary across systems.
  - Screen-reader behavior can differ across OS versions. These risks will be monitored during testing.
- **Constraints:**
  - Dependence on browser-based APIs may limit offline or air-gapped performance.
  - Variability in microphones and audio hardware may influence test outcomes.
  - Voice recognition is not working properly.
- **Entry & Exit Criteria:**
  - **Entry:** Build is stable, all primary features implemented.
  - **Exit:** All critical test cases pass, no major accessibility blockers remain.

5.2 Test Cases

The following tables list test cases for each major workflow. Each case has a unique ID, description, input conditions, expected output, space for actual results, and a Pass/Fail status. *Actual Result* and *Pass/Fail* will be filled in during testing.

Test Case 1:

BlindCoder — Test Case Document Page 1 of 5

Test Case: Voice-to-Code Insertion (BC_01)

Test Case #: BC_01	**Test Case Name:** Voice-to-Code Insertion	**Page:** 1 of 5
System: BlindCoder VS Code Extension	**Subsystem:** Speech Recognition / Editor Integration	
Designed by: Group 7	**Design Date:** 2025-11-25	

Short Description

Verify microphone input converts spoken code to text and inserts it at the cursor position in the active editor.

Pre-conditions

1) VS Code running with BlindCoder extension installed. 2) Webview panel open. 3) Active Python file in editor. 4) Microphone permission granted.

Step	Action	Expected System Response	Pass/Fail	Comment
1	Press the Voice button (Ctrl+M) to start recording	Microphone indicator shows active; status announces "Recording"	Pass	Indicator lit; speech synthesis announced recording.
2	Dictate: "def add(a, b): return a plus b"	Speech-to-text returns code-like text; extension inserts: `def add(a, b):\n return a + b`	Fail	Transcription correctly normalized to '+' and inserted at cursor.
3	Stop recording	Status returns to Ready; announcement "Snippet inserted"	Fail	Announcement played; log entry added "Snippet inserted".
4	Run basic syntax check on the file	No syntax errors reported for the inserted code	Fail	Inserted code passed linter; no diagnostics created.

Post-conditions

The new code is present in the editor at the cursor location and is syntactically correct or flagged if not.

Figure 22: Test Case 1

Test Case 2:

BlindCoder — Test Case Document Page 2 of 5

Test Case: Undo Voice Insertion (BC_02)

Test Case #: BC_02	**Test Case Name:** Undo Voice Insertion	**Page:** 2 of 5
System: BlindCoder Extension	**Subsystem:** Editor Commands	

Short Description

Ensure the "undo last voice insertion" action removes the last snippet inserted by voice.

Pre-conditions

An active editor with at least one voice-inserted snippet from previous test.

Step	Action	Expected System Response	Pass/Fail	Comment
1	Insert a voice snippet "print hello world"	Snippet inserted into editor: `print("hello world")`	Pass	Snippet inserted and visible in editor.
2	Click Undo (↶) or press Ctrl+Alt+U	Last voice insertion is removed; editor content reverts to previous state	Pass	Content reverted successfully; undo stack behaved as expected.
3	Attempt Undo again	Either previous insertion removed or graceful message if no more voice insertions to undo	Pass	Second undo removed earlier insertion; when none left, extension announced "No voice insertions to undo".

Post-conditions

Editor no longer contains the undone voice-inserted snippet.

Figure 23: Test Case 2

Test Case 3:

Test Case: Run & Narrate Output (BC_03)

Test Case #: BC_03	**Test Case Name:** Run & Narrate	**Page:** 3 of 5
System: BlindCoder	**Subsystem:** Execution & TTS	

Short Description

Verify that running the current Python file returns stdout/stderr that the extension narrates and plays earcons for success/error.

Pre-conditions

A runnable Python file is active in editor with a predictable output (e.g., prints "hello").

Step	Action	Expected System Response	Pass/Fail	Comment
1	Click Run & Narrate or press Ctrl+Alt+X	Extension executes file; status shows running; plays running earcon	Pass	Execution started; running earcon audible; status updated.
2	Observe stdout (program prints "hello")	Speech announces: "Program output: hello"; log contains stdout entry	Pass	Speech synthesis read output correctly; log contains the stdout text.
3	Introduce a runtime error (e.g., divide by zero) and run	Speech narrates stderr with error message; plays error earcon; diagnostics updated	Pass	Error narrated verbatim; error earcon played; diagnostic marker present in editor.

Post-conditions

Execution result recorded in activity log; audio narration matches output and errors.

Figure 24: Test Case 3

Test Case 4:

Test Case: Fix Code Syntax (BC_04)

Short Description

Validate the "Fix Code" action repairs common Python mistakes (missing colon, indentation) and announces changes.

Pre-conditions

An open Python file containing known simple syntax issues: e.g. "def foo()\nprint('hi')"

Step	Action	Expected System Response	Pass/Fail	Comment
1	Open the faulty file and click Fix Code (Ctrl+Alt+F)	Extension runs auto-fix; shows progress; announces "Fixes applied"	Pass	Auto-fix completed and summary announced.
2	Review changes in editor	Missing colon and indentation fixed; resulting code compiles without syntax error	Pass	Editor shows corrected code; linter passes.
3	Run syntax check	No syntax diagnostics remain for previously fixed issues	Pass	No diagnostics after fixes.

Post-conditions

File updated with fixes; a summary of changes recorded in log and announced.

Figure 25: Test Case 4

Test Case 5:

BlindCoder — Test Case Document Page 5 of 5

Test Case: Read Context & Where Am I (BC_05)

Test Case #: BC_05	**Test Case Name:** Read Context / Where Am I	**Page:** 5 of 5
System: BlindCoder	**Subsystem:** Navigation & TTS	

Short Description

Check that the extension can speak the current line, surrounding function/class name and overall scope when requested.

Pre-conditions

Open a Python file with multiple functions and classes; place cursor inside a function body.

Step	Action	Expected System Response	Pass/Fail	Comment
1	Invoke Read Current Line (Ctrl+Alt+L)	Speech reads the current line verbatim and highlights it in the log	Pass	Line read correctly; log entry created.
2	Invoke Read Context (Ctrl+Alt+C)	Speech announces function signature and brief surrounding lines	Pass	Function signature and two surrounding lines announced.
3	Invoke Where Am I (Ctrl+Alt+I)	Speech announces "Inside function <name> in file <filename>", including class if applicable	Pass	Correct scope announced including enclosing class when present.

Post-conditions

Navigation announcements correctly reflect cursor position and scope. No change to file contents.

Figure 26: Test Case 5

5.3 Test Reports by Peers

Peer testers can record their findings using a structured report format. Below is a template table and sample entries illustrating how to log what was tested, what worked, what failed, and overall impressions:

Tester	Date	Feature/Workflow Tested	What Worked	What Failed	Comments / Overall Impression
Harry	2025-11-15	Voice-to-Code (print)	Recognized “print x” correctly; code inserted	Misunderstood “loop i times”; inserted wrong syntax	Very responsive for basic commands; complex phrasing needs refinement.
Vivek	2025-11-16	Code Execution & Output	Simple outputs read correctly (“Hello” spoken)	Stopped speaking mid-output when code had two lines	Output narration is clear; should handle multi-line outputs fully.

Sources: Standard software testing documentation and accessibility guidelines were used to structure this plan[1][2]. The use of speech interfaces is based on Web Speech API documentation[7][8], and VS Code extension fundamentals[5]. Screen reader compatibility is informed by VS Code’s accessibility support[4][9]. Voice-coding examples draw on open-source tools (e.g. Serenade[3]) demonstrating natural-speech coding.

[1] [2] jmpovedar.files.wordpress.com

https://jmpovedar.files.wordpress.com/2014/03/ieee-829.pdf

[3] Serenade | Code with voice

https://serenade.ai/

[4] [9] Accessibility

https://code.visualstudio.com/docs/configure/accessibility/accessibility

[5] Extension Anatomy | Visual Studio Code Extension API

https://code.visualstudio.com/api/get-started/extension-anatomy

[6] Node.js — Run JavaScript Everywhere

https://nodejs.org/en

[7] [8] Web Speech API - Web APIs | MDN

https://developer.mozilla.org/en-US/docs/Web/API/Web_Speech_API