PILOT STUDY PROTOCOL

Hearing the Architecture of Code: AI-Driven Semantic Restructuring of Abstract Syntax Trees for Accessible Code Navigation by Blind Programmers

A Pilot Study Protocol for AI-Driven AST Restructuring to Improve Code Accessibility for Visually Impaired Software Engineers

Abstract

Blind and visually impaired developers rely on screen readers that expose code as a linear audio stream, discarding the hierarchical structure sighted developers see at a glance. Prior tools such as StructJumper, Sodbeans, and WAD expose structure through manually engineered, language-specific parsers that do not generalize. Large language models (LLMs) can now parse and summarize code from Abstract Syntax Trees (ASTs), but this capability has been evaluated for engineering productivity, not accessibility. This paper specifies a two-phase pilot of an AI Accessibility Auditor: a prototype that parses code into an AST, uses an LLM to generate short descriptions of each node, and renders the result as a navigable, hierarchical audio interface layered on a participant's existing screen reader. Phase 1 validates summary accuracy against ground-truth AST structure and a predefined human-authored reference description before any benefit claim is tested; Phase 2 then tests, in a within-subjects study with twelve blind developers, whether validated navigation improves completion time, errors, comprehension, cognitive workload, and usability relative to standard screen-reader use. The pilot is scoped to Python and JavaScript, and framed as a focused, extensible pilot rather than a general-purpose claim.

1. Introduction

Opening an unfamiliar, thousand-line file with a screen reader exposes a structural asymmetry in how code is read: sighted developers glance at a file and absorb its shape in seconds — which function calls which, where nesting begins and ends — while blind developers, working through JAWS, NVDA, or VoiceOver, are limited to a linear, line-by-line stream with no equivalent overview. Albusays, Ludi, and Huenerfauth (2017), observing 28 blind professional programmers, found this sequential access was their single most consistent source of frustration, especially when jumping between a function definition and its use. Mealin and Murphy-Hill (2012) reported the same pattern independently: losing one's place in a codebase is a recurring, defining cost of the job, not a minor annoyance.

This paper specifies a testable research direction — using large language models (LLMs) to parse code into its Abstract Syntax Tree (AST) and re-render that tree as a structured, hierarchical audio experience — and scopes it deliberately: a two-phase pilot, first validating AI-generated summary accuracy and then testing its effect on navigation, centered on two widely used languages rather than a general-purpose claim across all languages and codebases.

1.1 The Scale of the Problem

According to the 2022 Stack Overflow Developer Survey, 1.7% of respondents identified as blind or low-vision — nearly 45% of all respondents reporting any disability (Stack Overflow, 2022; cited in Cha et al., 2026) — yet blind and low-vision adults, roughly 5% of the U.S. population, remain under 2% of the software workforce. The gap extends beyond software: the American Foundation for the Blind (2023) reports only 44% of blind or visually impaired adults are employed, versus 79% of people without disabilities, a gap that has barely moved in a decade (Figure 1). Programming's core output — text — is in principle already screen-reader accessible; that blind programmers still face a steep penalty suggests the barrier is not the code itself but the interface between code and ear.

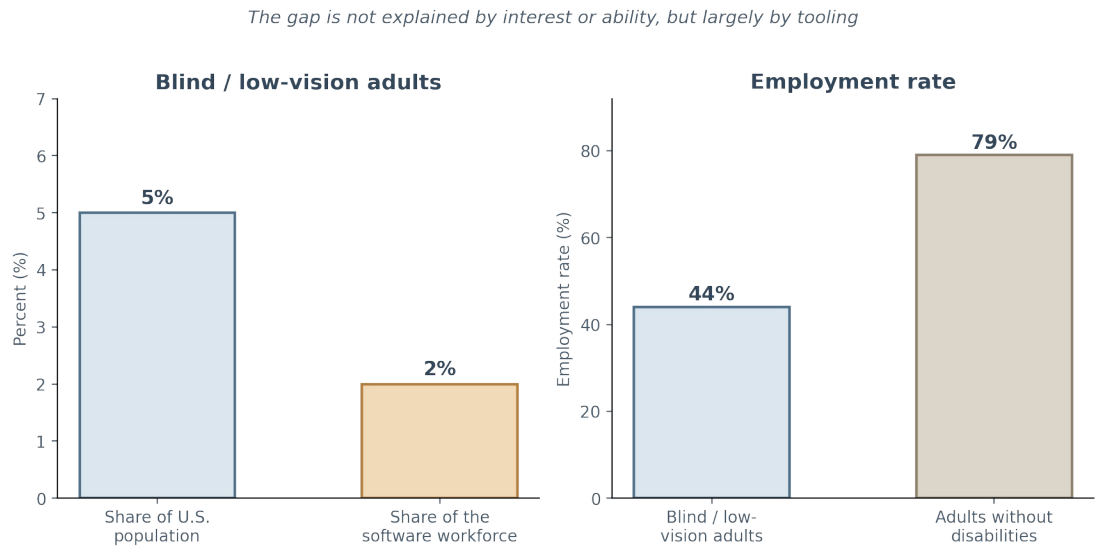


Figure 1. Blind and low-vision adults are underrepresented in the software workforce relative to the general population, and face a large employment-rate gap (Stack Overflow, 2022; American Foundation for the Blind, 2023).

1.2 Contributions

This study makes four contributions:

An AI-assisted semantic AST navigation framework, piloted on Python and JavaScript rather than claimed as general-purpose.

A connection between accessibility research and LLM-based, AST-aware code understanding — two literatures that have so far developed largely in parallel.

A working prototype architecture, the AI Accessibility Auditor, with an independent accuracy-validation step preceding any navigation-benefit claim.

A two-phase experimental design — validation, then a within-subjects navigation study — evaluating effects on navigation efficiency, comprehension, and cognitive workload.

2. Research Gap

Three findings, taken together, define what remains untested. First, existing accessibility tools such as StructJumper, Sodbeans, and WAD improve structural navigation but rely on manually engineered, language-specific parsers that do not scale across languages (Baker et al., 2015; Albusays et al., 2017). Second, recent LLM-based software engineering research shows strong AST reasoning and summarization ability (Hou et al., 2024; Zhang et al., 2025), but has been evaluated almost exclusively for general engineering productivity, not accessibility. Third, no published study has validated whether AI-generated summaries are accurate enough to be trusted by a user who cannot visually verify them before testing any resulting navigation benefit. Table 1 states the gap directly.

Prior work / line of research	Structural navigation	Language-agnostic	Validated with blind/low-vision users	Key limitation
StructJumper (Baker, Milne & Ladner, 2015)	Yes — navigable tree view of a class's nesting	No — Java only, hand-built Eclipse parser	Yes — 7 blind programmers, faster task completion	Fixed to one language's grammar; new engineering needed to generalize

Prior work / line of research	Structural navigation	Language-agnostic	Validated with blind/low-vision users	Key limitation
Sodbeans (cited in Albusays et al., 2017)	Yes — audible cues over a hierarchical tree	No — built for one teaching environment (NetBeans)	Used in blind-student programming education	Same manually engineered, single-toolchain limitation
Wicked Audio Debugger (WAD)	Partial — sonifies live program state, not static structure	No — Visual Studio only	No formal accessibility validation reported	Live-state sonification, not a navigable structural view
LLM–AST reasoning research (Hou et al., 2024; Zhang et al., 2025)	N/A — studies model reasoning, not a navigation UI	Yes, in principle — general-purpose LLM reasoning	No — evaluated for engineering productivity, not accessibility	Capability shown, but never pointed at the accessibility problem
LLM code-segmentation study (IEEE Access, 2025)	Partial — segments code into explained sections	Yes, in principle	Partial — 20 sighted, blindfolded participants on intro exercises	Not tested with blind professionals in real codebases; no navigable interface evaluated
AI coding assistants (Copilot, Claude, Gemini Code Assist, Cursor)	No — conversational explanation, not spatial navigation	Yes	No	Developer must still locate code by reading it linearly first

Table 1. What existing work has and has not done.

This paper addresses that gap with a two-phase design — validating summary accuracy first, then testing navigation benefit — evaluated as a focused pilot rather than a general-purpose solution.

3. Related Work

3.1 Accessibility Barriers in Programming

Screen readers are a genuine engineering achievement; most blind professional developers rely on them daily. Lock, Brown, and Kölling (2025) found blind programmers consistently prefer tools that stay compatible with their existing screen reader rather than a parallel interface, since switching paradigms itself imposes cognitive cost — a preference any proposed solution must respect. But reliability is not sufficiency: Mountapmbeme, Okafor, and Ludi's (2022) review documents specific mechanical limits — indentation is largely invisible to a screen reader, Python's semantically load-bearing whitespace is essentially unreadable without special configuration, and braces or brackets can consume an entire spoken line. These are not flaws in the screen reader; they reflect a structural mismatch between a technology built for prose and a medium — source code — that is fundamentally a graph rather than a sentence.

3.2 Existing Accessibility Tools

Baker, Milne, and Ladner (2015) built StructJumper, an Eclipse plugin generating a navigable tree view of a Java class's nesting structure; seven blind programmers completed tasks faster with it, though the sample was small. Sodbeans, layered on NetBeans, used audible cues in a hierarchical tree to teach programming to blind students (cited in Albusays et al., 2017). The Wicked Audio Debugger (WAD) sonified live program state for Visual Studio in the mid-2000s. Each tool shares a premise — the fix is structure, not more speech — and a limitation: each is generated by a parser fixed to one language's syntax, unable to say what a block of code is for, and unable to generalize without new engineering effort.

3.3 AI for Code Understanding

AST-structured input for code summarization predates the current wave of LLMs: encoder-decoder models that incorporate AST structure, rather than a flat token sequence, have consistently outperformed those that do not (Hu, Li, Xia, Lo, and Jin, 2018). Dong, Zhao, and Harvey's (2026) empirical study, run on Llama 3 (Dubey et al., 2024), benchmarks AST serialization directly against plain code as an LLM input format. Zhang et al.'s GALLa (2025, ACL) shows that training an LLM to reason over AST, control-flow, and data-flow graphs improves its downstream understanding of what code does, not only what it says. A 2025 IEEE Access study went further, using an LLM to segment code into semantically coherent, explained sections for accessibility — but tested it with twenty sighted, blindfolded participants on introductory exercises, not blind professionals working in real codebases, and did not evaluate a navigable hierarchical interface.

3.4 Comparison with Existing AI Coding Assistants

Existing AI coding assistants — GitHub Copilot, Claude, Gemini Code Assist, Cursor — generate or explain code conversationally, but do not restructure architecture into a hierarchical auditory representation. They assist code generation and incidental explanation rather than spatial exploration; a developer still has to locate relevant code by reading it linearly first. The navigation problem itself remains largely unaddressed by this class of tool.

4. Research Questions and Hypotheses

4.1 Research Questions

RQ1: Can AI-generated semantic restructuring of Abstract Syntax Trees improve code navigation efficiency and comprehension for blind programmers, relative to standard screen-reader navigation?

RQ2: Does hierarchical auditory presentation of code structure reduce perceived cognitive workload and improve usability compared with traditional, linear screen-reader navigation?

RQ3: How accurately do LLM-generated structural summaries represent the actual logical architecture of source code, judged against ground-truth structure?

4.2 Hypotheses

H1: AI-assisted AST restructuring significantly reduces navigation time relative to standard screen-reader navigation.

H2: Semantic audio navigation significantly improves code comprehension accuracy relative to standard screen-reader navigation.

H3: Hierarchical navigation significantly lowers perceived cognitive workload (NASA-TLX) relative to standard screen-reader navigation.

H4: Participants report higher usability (SUS) for AI-assisted AST navigation than for standard screen-reader navigation.

Table 2 states, for each question, exactly where in the design its answer comes from.

Research question	Hypothesis(es)	How it is answered	Where in the design
RQ1: Does AI-generated semantic restructuring of ASTs improve navigation efficiency and comprehension for blind programmers, relative to standard screen-reader navigation?	H1 (navigation time), H2 (comprehension accuracy)	Phase 2 within-subjects comparison of screen-reader-only vs. AI-assisted AST navigation, timed and scored per task	Sections 7.2–7.5

Research question	Hypothesis(es)	How it is answered	Where in the design
RQ2: Does hierarchical auditory presentation of code structure reduce perceived cognitive workload and improve usability, relative to linear screen-reader navigation?	H3 (NASA-TLX workload), H4 (SUS usability)	Post-task and post-session NASA-TLX and SUS questionnaires in the same within-subjects comparison	Sections 7.5–7.6
RQ3: How accurately do LLM-generated structural summaries represent the actual logical architecture of source code, judged against ground-truth structure?	Gate, not a directional hypothesis	Phase 1 blinded-rater accuracy validation against the source code and a human-authored reference description	Section 7.1

Table 2. How each research question is operationalized and answered.

5. Conceptual Framework

Figure 2 summarizes the causal chain from source code to outcome measures: source code → AST → LLM-generated hierarchical semantic tree → audio navigation interface → four outcome constructs (navigation efficiency, comprehension, cognitive load, satisfaction), corresponding directly to the dependent variables in Section 7.5. Because the pilot is scoped to Python and JavaScript, the pipeline should be read as instantiated for these two languages, not as a language-agnostic guarantee.

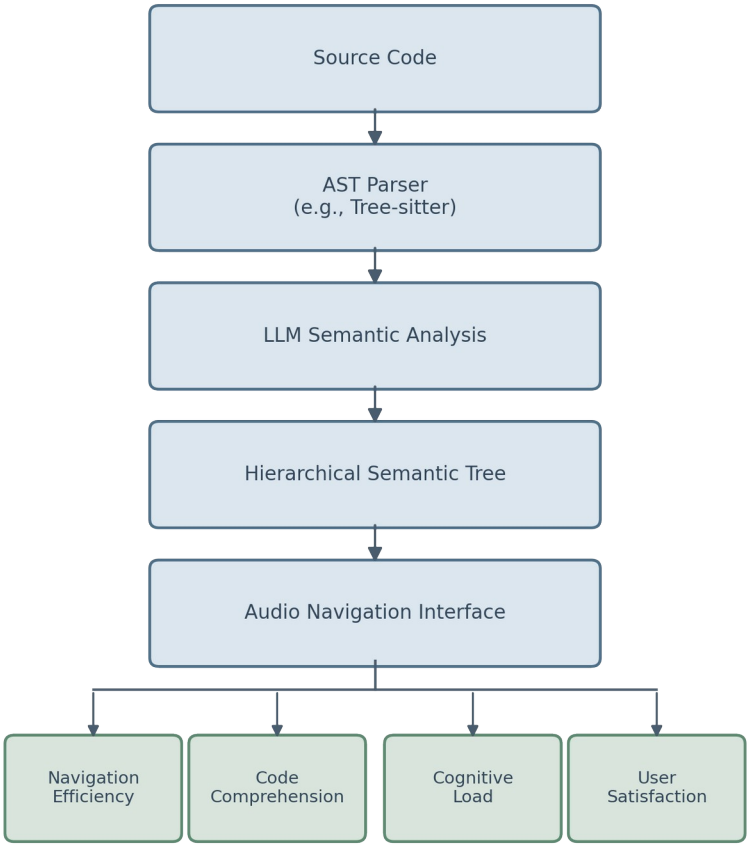


Figure 2. Conceptual framework linking the proposed AST-to-audio pipeline to the outcome measures evaluated in this study.

6. Proposed System: The AI Accessibility Auditor

The proposed system, the AI Accessibility Auditor, would take the shape of a VS Code extension — chosen because blind developers prefer tools inside their existing, trusted screen-reader workflow rather than a new parallel interface (Lock, Brown, and Kölling, 2025).

The pipeline has three layers, shown in Figure 3. A parsing layer converts source code (Python or JavaScript) into its AST using a mature, incremental parser such as Tree-sitter, with nodes for functions, classes, and control structures. A language layer prompts an LLM with the parsed tree to generate short natural-language descriptions of each node and its relationships (Section 3.3). An audio layer renders these not as one monologue but as a navigable, spatial hierarchy — letting a user move up and down tree levels the way StructJumper's view allowed for Java, but generated dynamically rather than hardcoded to one grammar.

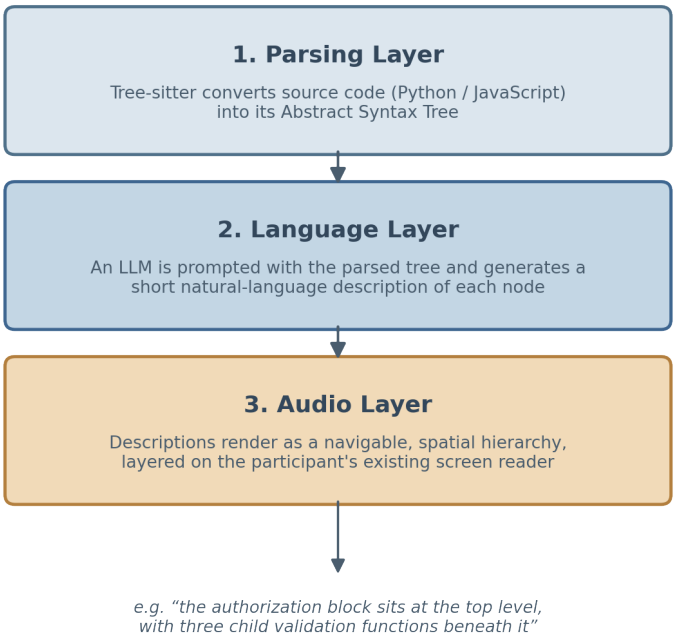


Figure 3. Three-layer architecture of the AI Accessibility Auditor: parsing, language, and audio layers. Figure 5, at the end of this section, shows the same pipeline as an exploded product view.

The contribution worth defending is not the plugin — plugins are engineering — but the empirical question in Section 4: whether an AI-generated semantic tree measurably reduces the cognitive burden of code comprehension for blind developers relative to linear screen-reader reading. Before any such claim, Section 7.1 validates the accuracy of this third layer's output against ground-truth AST structure.

This research does not claim AI-generated audio maps replace VoiceOver or NVDA; both do enormous, hard-won work reading interface elements and general text reliably and are treated as the baseline being extended, not replaced. The claim under test is narrower: for understanding the architecture of a large or unfamiliar codebase, an AI-generated hierarchical representation outperforms a purely linear reading of the same text, while the rest of the developer's screen-reader workflow remains untouched.

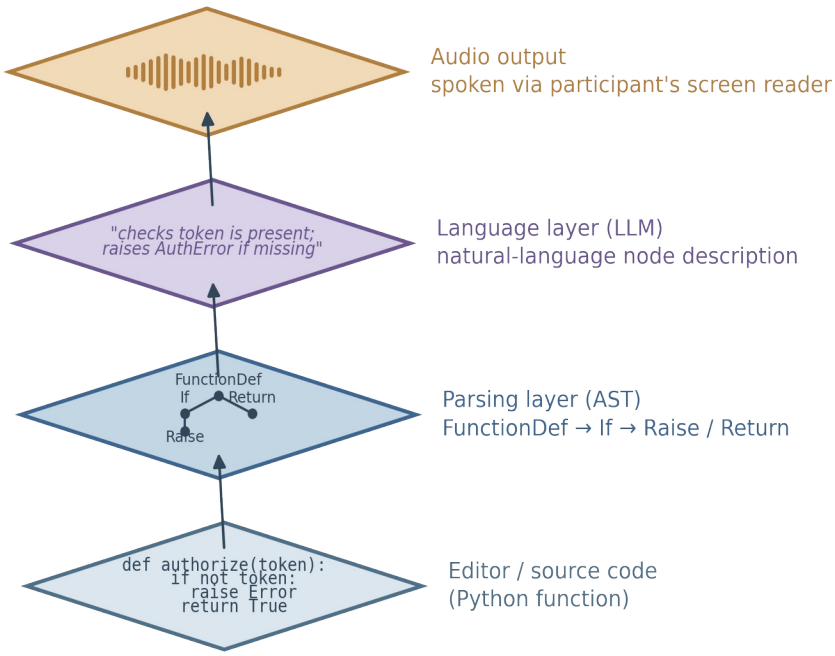


Figure 5. Exploded isometric view of the AI Accessibility Auditor pipeline, illustrated with a representative Python function: source code enters at the base, is parsed into an AST, summarized by the language layer, and rendered as expanding audio output at the top.

Figure 6 traces this pipeline through a participant's actual use of the tool: putting on the headset used with their screen reader, then stepping through the resulting audio navigation while working in an unfamiliar file.

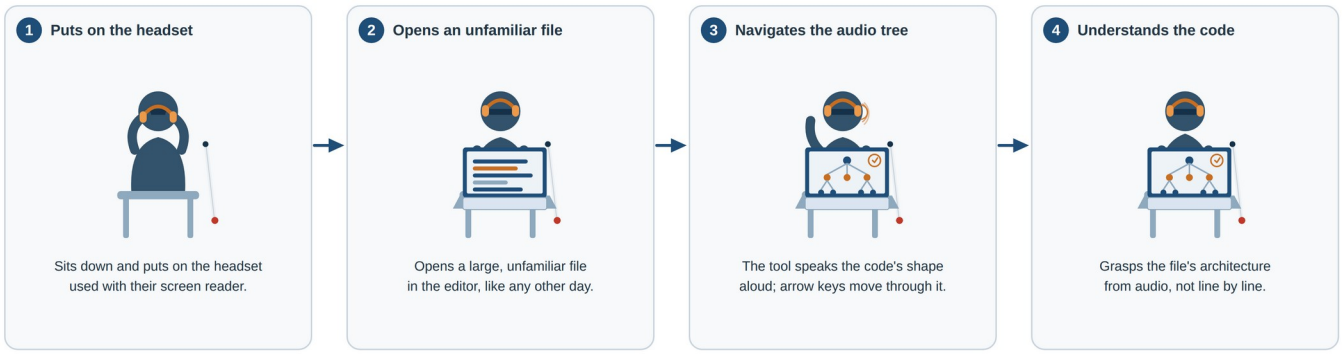


Figure 6. Day-in-the-life illustration of the AI Accessibility Auditor: a blind developer puts on the headset used with their screen reader, opens an unfamiliar file, navigates the tool's hierarchical audio tree, and understands the file's architecture without reading it line by line.

7. Method

The study proceeds in two phases, summarized in Figure 4. Phase 1 establishes, independently of any user study, whether the LLM-generated node descriptions are accurate enough to serve as a basis for navigation. Only if a pre-specified accuracy threshold is met does Phase 2 test whether validated navigation changes efficiency, comprehension, or cognitive load relative to standard screen-reader use — directly addressing the risk (Section 8) that a fluent but inaccurate summary could mislead a user who cannot visually cross-check it.

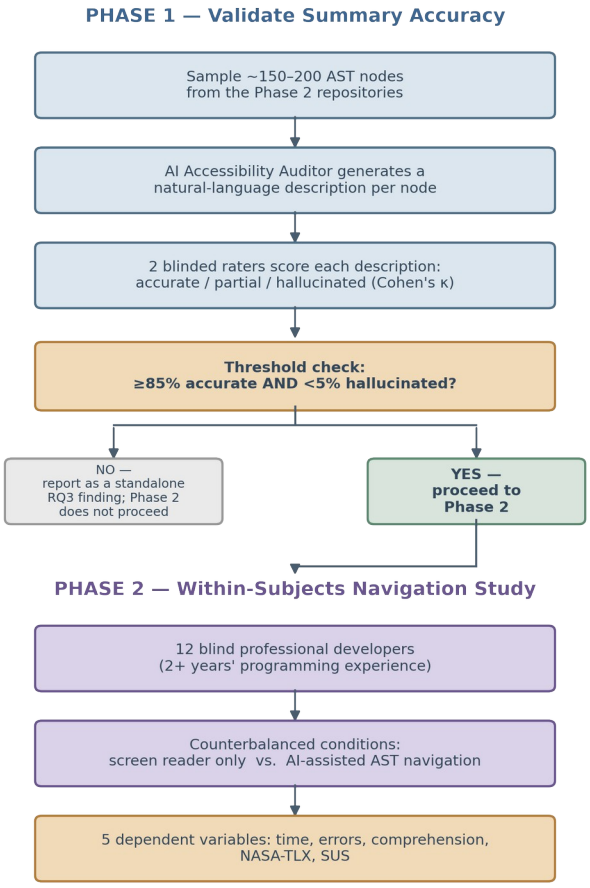


Figure 4. Two-phase study design: Phase 1 validates summary accuracy before Phase 2 tests navigation benefit.

7.1 Phase 1: Validation of Summary Accuracy

A stratified sample of approximately 150–200 AST nodes (functions, classes, control-flow blocks) will be drawn from the Phase 2 repositories. Two sighted software engineers with at least three years' experience, blinded to which descriptions are AI-generated, will independently rate each against the source code using a three-point rubric: accurate (the description's claims about the node's behavior, inputs, and relationships are all correct and complete enough to navigate by), partially accurate (the description is directionally correct but omits or slightly mischaracterizes a material detail, such as a side effect or an exception path), and hallucinated (the description asserts behavior, parameters, or relationships not present in the source).

An AST captures syntactic and structural ground truth — what nodes exist and how they nest — but not fully a function's intent, side effects, or architectural role. To keep this distinction explicit, the rubric will be built from two reference points rather than one: the source code itself, and a predefined, human-authored reference description written by the study team before rating begins. Raters will record syntax/structure accuracy (does the description correctly reflect the AST's nodes and relationships) separately from semantic/intent accuracy (does it correctly reflect what the code is for, including side effects and exception paths), so a description can be flagged as structurally correct but semantically misleading, or vice versa, rather than collapsed into one score.

Two independent raters, rather than one, are used because a single rater's judgment cannot be distinguished from that rater's individual bias or fatigue; a second, blinded rater makes systematic over- or under-crediting of the tool detectable rather than invisible, and Cohen's kappa quantifies how much the two agree beyond chance. Disagreements are resolved by a third, independent rater rather than by averaging the two scores or by discussion between the original raters, since averaging can silently launder a real error into a passing “partially accurate” score, and discussion risks the more senior or more confident rater anchoring the other's judgment

rather than genuinely resolving it.

Pre-specified gate: *Phase 2 proceeds only if ≥85% of rated descriptions are accurate AND under 5% are hallucinated. The threshold is asymmetric because the two error types carry different costs to a blind user who cannot visually cross-check a summary — a partially accurate description degrades navigation quality, while a hallucinated one actively misleads — so hallucination is held to a tighter bound than overall accuracy. If the threshold is unmet, the result is reported as a standalone RQ3 finding and Phase 2 does not proceed with the unmodified tool. This rule is fixed in advance, before any node is rated, to prevent quietly lowering the evidentiary bar after seeing the data.*

7.2 Phase 2: Participants

Twelve blind professional developers with at least two years' programming experience will be recruited through accessibility organizations and developer communities (for example, AccessComputing, the American Foundation for the Blind). This matches prior within-subjects evaluations in the subfield — Mealin and Murphy-Hill (2012) interviewed eight developers, Baker et al. (2015) evaluated StructJumper with seven — reflecting the real difficulty of recruiting blind professional developers rather than an arbitrary target. Per the power analysis in Section 7.6, this size is sufficient to detect only large effects.

7.3 Study Design and Procedure

The study uses a within-subjects design across three to five GitHub repositories of varying size and complexity. Each repository is paired with either a naturally occurring, previously filed issue with a known fix, or a single-function defect from a fixed injection protocol (for example, an off-by-one bound, an inverted conditional, a missing null check), stratified so each participant sees a comparable mix in both conditions. In one condition, participants navigate with their normal screen reader while a researcher records completion time and navigation errors; in the other, the same participants use the validated AST-to-audio tool on the same repositories. Condition order is counterbalanced and repositories rotated to control for learning effects.

Comparator fairness. “Normal screen-reader use” otherwise varies across editors, keyboard shortcuts, screen readers, and experience levels in ways that could confound the comparison. The code editor version and task environment will therefore be standardized across both conditions, while participants use their own configured screen reader to preserve ecological validity. Screen reader and version, operating system, years of programming experience, and prior familiarity with the study's languages will be recorded for each participant and treated as covariates in the analysis.

Confounding. The treatment bundles two changes at once — a different navigation structure and an added semantic explanation — so a better score in the AI-assisted condition would not by itself show which feature drove the improvement. This pilot is scoped to evaluate the combined intervention rather than isolate the LLM's individual contribution; a third condition (deterministic AST navigation without LLM summaries) is a natural extension for a follow-up study and is noted here as future work rather than folded into the current design, which is already constrained by the sample size in Section 7.2.

7.4 Independent Variable

The independent variable is navigation method, with two levels:

Screen reader only (baseline condition).

AI-assisted AST navigation (treatment condition).

7.5 Dependent Variables

Dependent variable	Operational definition	How it is measured
Navigation time	Time taken to locate the code relevant to a given task in a repository	Researcher-recorded per task during the session
Navigation errors	Count of incorrect navigation actions or false starts per task	Researcher-recorded per task during the session
Comprehension accuracy	Correctness of participants' answers about the code's structure and behavior after navigating it	Post-task comprehension questions, scored against a rubric
Cognitive workload	Perceived mental demand and effort of completing the navigation task	NASA-TLX questionnaire, administered after each task
Usability	Perceived ease of use and satisfaction with the navigation method	System Usability Scale (SUS) questionnaire, administered after each condition

Table 3. Dependent variables, their operational definitions, and how each is measured.

7.6 Statistical Analysis

The primary analysis will use linear mixed-effects models, with participant — and repository or task, where the data permit — specified as random effects; this tolerates the missing sessions and unbalanced repository exposure a small field study is likely to produce, and models individual variability more directly than a strict repeated-measures design. Repeated-measures ANOVA, or Wilcoxon signed-rank tests where normality or sphericity assumptions are violated, will be reported alongside as a secondary sensitivity check rather than as the primary test, with effect sizes reported using Cohen's d. Because five dependent variables are each tested across the same two conditions, p-values across the family of comparisons will be adjusted using the Holm-Bonferroni correction to control the family-wise error rate, and this correction will be specified in the preregistration rather than applied post hoc.

An a priori power analysis (G*Power, paired-samples design, $\alpha = .05$, 80% power) shows that $n = 12$ is powered only for large within-subject effects ($d \geq 0.8$); the study is framed accordingly as a pilot for identifying large, practically meaningful effects, not a definitive test of small ones. Effect sizes and confidence intervals are emphasized alongside significance testing, each participant's paired outcomes will be reported rather than group averages alone, and the full analysis plan — including the correction procedure, model specification, and the Phase 1 stopping rule from Section 7.1 — will be preregistered (for example, via OSF) before Phase 2 data collection begins. A 20–40% reduction in completion time or navigation-error rate — comparable to StructJumper's reported gains — is the threshold for a practically meaningful finding.

7.7 Reproducibility and Implementation Transparency

To meet the reporting bar expected of an empirical journal submission rather than a protocol alone, the prototype's fixed model version, prompt templates, and non-sensitive benchmark stimuli will be archived and made available where permitted, alongside the pipeline settings used to produce every rated description in Section 7.1. Implementation failures — prompts that had to be revised, node types the parser mishandled, latency issues encountered during piloting — will be reported alongside successes, so that the published record reflects what was actually built and tested rather than only its final, working form.

8. Anticipated Limitations of LLM-Based Semantic Restructuring

Phase 1 (Section 7.1) is designed to catch the most serious of these failure modes before Phase 2 begins; the rest are addressed through the study-design choices below. Table 4 summarizes each risk and how it is addressed. RQ3 and the comprehension variable are designed specifically to surface these failure modes empirically, rather than assume the tool's summaries are reliable by default.

Anticipated risk	Why it matters	How the design addresses it
Hallucinated node description	Asserts behavior, parameters, or relationships not present in the source — actively misleads a user who cannot visually cross-check it	Phase 1 gate requires under 5% hallucinated ratings before Phase 2 may proceed (Section 7.1)
Partially accurate description	Directionally correct but omits or mischaracterizes a material detail, such as a side effect or exception path	Phase 1 requires at least 85% accurate ratings; the rubric records this separately from hallucination (Section 7.1)
Structural accuracy conflated with semantic accuracy	A description can be correct about AST shape while misleading about intent, or vice versa	Raters score syntax/structure accuracy and semantic/intent accuracy as two separate judgments (Section 7.1)
Fluent but wrong output invites over-trust	A confidently worded, incorrect summary is harder to catch than an obviously broken one, especially without visual cross-checking	RQ3 and the comprehension variable are designed to surface this empirically rather than assume the tool is reliable by default (Section 8)
Node types the parser or LLM handles poorly	Coverage gaps could produce silently low-quality descriptions for less common constructs	Implementation failures, including mishandled node types, are logged and reported alongside successes (Section 7.7)
Findings may not generalize beyond the pilot's scope	Two languages and three to five repositories cannot represent the full diversity of real-world code	Explicitly scoped as a pilot; language and repository limits are stated up front (Sections 1, 10)

Table 4. Anticipated risks of LLM-based semantic restructuring and how the study design addresses each.

9. Ethical Considerations

This study involves human participants from a population that is both protected and, in research contexts, historically under-consulted about tools built ostensibly for their benefit. The study protocol will include:

- Informed consent obtained in an accessible format — screen-reader-compatible digital consent forms, with a verbal walkthrough option.
- Accessibility accommodations for the study sessions themselves, including participant choice of remote or in-person setting and use of the participant's own configured screen reader and hardware.
- Anonymization of session recordings and transcripts prior to analysis, with identifying details removed from any quoted material.
- Institutional Review Board (IRB) approval prior to recruitment, including review of the informed-consent materials and data-handling plan.

Recruitment will proceed only after a faculty research supervisor is secured and full IRB approval is obtained, including review of recruitment partnerships with accessibility organizations, a participant compensation plan, and a data-management plan; no participant will be recruited and no research data will be collected before that approval is in place.

10. Limitations

Beyond the LLM-specific risks in Section 8, four broader constraints bear on interpretation:

- Sample size. Twelve participants, consistent with prior work in this subfield, constrain statistical power to large effects and limit generalizability; findings indicate feasibility and direction, not a population-level claim. Recruitment through accessibility organizations may also select for comparatively well-resourced, experienced developers.

Single session. Effects of sustained use — learning curves, novelty effects that fade with repetition — fall outside this design and would require a longer-term follow-up study.

Repository and language scope. Three to five repositories cannot represent the full diversity of real-world code; the pilot is scoped to Python and JavaScript, and results may not transfer to languages with substantially different structural idioms.

Observer-dependent measurement. Completion time and errors are recorded by a researcher during the session; inter-rater checks on the comprehension rubric mitigate, but do not eliminate, this concern.

Combined intervention. Because navigation structure and semantic explanation change together in the treatment condition (Section 7.3), the study evaluates their combined effect and cannot, on its own, attribute the outcome to either component individually.

11. Discussion: Broader Significance

Software has become one of the more visually saturated professions that still nominally runs on text — modern frameworks, architecture diagrams, and design tools such as Figma pull programming further from a purely textual discipline, quietly raising the barrier for blind engineers even though the underlying source code remains, in principle, just as readable as ever. Tools that translate structural information back into something a screen-reader user can navigate are, on this view, closer to necessary infrastructure for keeping programming text-accessible — a motivation for this pilot rather than a claim it sets out to prove directly.

The proposal also sits at an intersection rarely worked directly: accessibility researchers (ACM ASSETS, CHI) have spent two decades documenting where blind programmers struggle (Armaly, Rodeghero, and McMillan, 2017; Lock, Brown, and Kölling, 2025), while LLM software-engineering researchers (ICSE, arXiv) have rapidly improved how models understand code structure (Hou et al., 2024). Cha et al.'s (2026) ICSE study of blind and low-vision professionals' actual generative-AI use found real productivity gains alongside real frustration when models hallucinated confidently wrong descriptions users could not visually double-check — a caution this proposal takes seriously in Sections 7.1 and 8, not as a footnote.

12. Conclusion

This paper specifies, rather than gestures at, a two-phase pilot connecting AST-aware LLM summarization to accessible, hierarchical audio navigation for blind programmers: first validating whether AI-generated summaries are accurate enough to trust, then testing whether validated navigation measurably helps. It sits where two unresolved problems meet — reliable LLM reasoning about code structure, and a persistent equity gap in the technology workforce — and does not claim to solve either at the scale of arbitrary languages or codebases. With a validation step, a working prototype scoped to two languages, and a modest but real sample of participants, it is tractable enough to make measurable, honestly bounded progress on both.

References

Albusays, K., Ludi, S., and Huenerfauth, M. (2017). Interviews and Observation of Blind Software Developers at Work to Understand Code Navigation Challenges. Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '17), 91–100.

American Foundation for the Blind. (2023). Disability Employment Research: Key Takeaways. Retrieved from afb.org/research-and-initiatives/employment.

Armaly, A., Rodeghero, P., and McMillan, C. (2017). A Comparison of Program Comprehension Strategies by Blind and Sighted Programmers. IEEE Transactions on Software Engineering, 44(8), 712–724.

Baker, C. M., Milne, L. R., and Ladner, R. E. (2015). StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15), 3043–3052.

Cha, Y., Jackson, V., Shu, L., Branham, S. M., and van der Hoek, A. (2026). “Game Changer” or “Overenthusiastic Drunk Acquaintance”? Generative AI Use by Blind and Low Vision Software Professionals in the Workplace. Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26).

Dong, S., Zhao, H., and Harvey, P. (2026). Code vs Serialized AST Inputs for LLM-Based Code Summarization: An Empirical Study. Proceedings of the 3rd International Workshop on Large Language Models for Code (LLM4Code 2026), co-located with ICSE 2026.

Dubey, A. et al. (2024). The Llama 3 Herd of Models. Meta AI technical report. arXiv:2407.21783.

Enhancing Algorithm Comprehension for Visually Impaired Individuals with LLM-Based Programming Code Segmentation. (2025). IEEE Access.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H. (2024). Large Language Models for Software Engineering: A Systematic Literature Review. ACM Transactions on Software Engineering and Methodology.

Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. (2018). Deep Code Comment Generation. Proceedings of the 26th IEEE/ACM International Conference on Program Comprehension (ICPC '18), 200–210.

Lock, J. G., Brown, N. C. C., and Kölling, M. (2025). Programming Education for Blind and Low Vision Users: Beyond Reading the Screen. Proceedings of the 25th Koli Calling International Conference on Computing Education Research (Koli Calling '25).

Mealin, S., and Murphy-Hill, E. (2012). An Exploratory Study of Blind Software Developers. Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 71–74.

Mountapmbeme, A., Okafor, O., and Ludi, S. (2022). Addressing Accessibility Barriers in Programming for People with Visual Impairments: A Literature Review. ACM Transactions on Accessible Computing, 15(1).

Stack Overflow. (2022). 2022 Developer Survey: Disability and Accessibility Demographics.

Zhang, Z., Yu, H., Li, S., Di, P., Li, J., and Wang, R. (2025). GALLa: Graph Aligned Large Language Models for Improved Source Code Understanding. Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL 2025). arXiv:2409.04183.