NaturalEdit: Code Modification through Direct Interaction with Adaptive Natural Language Representation

Ningzhi Tang
University of Notre Dame
Notre Dame, IN, USA
ntang@nd.edu

David Meininger
University of Notre Dame
Notre Dame, IN, USA
dmeining@nd.edu

Gelei Xu
University of Notre Dame
Notre Dame, IN, USA
gxu4@nd.edu

Yiyu Shi
University of Notre Dame
Notre Dame, IN, USA
yshi4@nd.edu

Yu Huang
Vanderbilt University
Nashville, TN, USA
yu.huang@vanderbilt.edu

Collin McMillan
University of Notre Dame
Notre Dame, IN, USA
cmc@nd.edu

Toby Jia-Jun Li
University of Notre Dame
Notre Dame, IN, USA
toby.j.li@nd.edu

Abstract

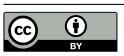
Code modification requires developers to comprehend code, plan changes, articulate intent, and validate outcomes, making it cognitively demanding. While natural language (NL) code summaries offer a promising external representation of this process, existing approaches remain limited. Systems grounded in exploratory data analysis are restricted to narrow domains, while general-purpose systems enforce fixed NL representations and assume that developers can directly translate vague intent into precise textual edits. We present NaturalEdit, which treats code summaries as interactive representations tightly linked to source code. Grounded in the *Cognitive Dimensions of Notations*, NaturalEdit introduces three key features: (1) adaptive, multi-faceted code summaries with a flexible *Abstraction Gradient*; (2) interactive mapping mechanisms between summaries and code that ensure tight, structurally stable *Closeness of Mapping*; and (3) intent-driven bidirectional synchronization that reduces *Viscosity* during editing while preserving *Visibility* and *Consistency* through incremental diffs. A technical evaluation confirms NaturalEdit's viability, and a user study with 20 developers shows that it improves comprehension, intent articulation, and validation while increasing developers' perceived confidence and control.

CCS Concepts

• **Human-centered computing** → **Interactive systems and tools**; • **Software and its engineering** → *Software notations and tools.*

Keywords

Natural Language Programming, Code Modification, Code Summarization



UIST '26, Detroit, MI, USA

ACM ISBN 979-8-4007-2856-3/2026/11
https://doi.org/10.1145/3830398.3830631

ACM Reference Format:
Ningzhi Tang, David Meininger, Gelei Xu, Yiyu Shi, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2026. NaturalEdit: Code Modification through Direct Interaction with Adaptive Natural Language Representation. In *The 39th Annual ACM Symposium on User Interface Software and Technology (UIST '26), November 02–05, 2026, Detroit, MI, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3830398.3830631

1 Introduction

Developers devote a substantial portion of their effort to code modification—altering existing programs to fit new tasks [39, 47]. This process is cognitively demanding: following Shneiderman and Mayers' model [47], developers must first construct an "internal semantics," a mental model of the program's logic; then form a modification plan, map it back to the program's formal syntax; and finally validate that the result reflects their intent. Crucially, this internal semantics remains implicit throughout: it is the cognitive target of comprehension, the basis of planning, and the standard against which validation is judged, yet it exists only in the developer's mind. Natural Language (NL) code summaries are especially promising as externalizations of this implicit model [32], which describes the functionality of the program in human-readable form. However, these summaries have traditionally served as static documentation for comprehension [25, 49, 58], leaving significant gaps in the later stages of expressing changes and validation.

While recent work has begun to make NL summaries interactive, existing approaches leave critical gaps. Early systems grounded in exploratory data analysis [28, 54, 55] rely on rule-based pipelines constrained to narrow, well-defined operation spaces (e.g., SQL query rewriting), limiting their applicability to general-purpose programming, where code semantics and developer intent are far more diverse. One of the most relevant prior work, NL outlines [46], envisions NL as an interactive editing surface for code maintenance, but explicitly leaves this workflow unevaluated. Moreover, its design also has two fundamental limitations: it generates a single fixed hierarchical outline, with no mechanism to adjust granularity or format across different cognitive tasks, and it requires users to directly

rewrite outline text to express changes without system mediation, sidestepping the "fuzzy abstraction matching" problem [45], namely the difficulty of translating vague intent into the precision required to modify a representation. More broadly, analyzing existing NL-mediated systems through the *Cognitive Dimensions of Notations* framework [16], grounded in empirical findings [28, 53], reveals four systemic gaps: a fixed ***abstraction gradient*** that cannot adapt across cognitive tasks; low ***closeness of mapping***, as implicit NL-code links burden working memory; high ***viscosity*** in translating intent into summary edits; and poor ***visibility*** and ***consistency*** when summaries are regenerated without surfacing inspectable diffs.

To address these challenges, we present NaturalEdit, a VS Code extension that treats a program's NL description as a first-class interactive representation, introducing three core features:

(1) *Adaptive Multi-Faceted Representation*: dynamically adapts NL summaries in structure and granularity, creating an adaptive *abstraction gradient* tailored to the developer's current task.
(2) *Interactive Cross-Representation Mapping*: provides fine-grained, structurally stable links between NL segments and code, ensuring explicit *closeness of mapping* throughout the workflow.
(3) *Intent-Driven Bidirectional Synchronization:* allows developers to express high-level goals as free-form instructions, which the system translates into concrete NL edits and code changes, reducing *viscosity* while preserving *visibility* and *consistency* through incremental diffs.

We evaluated NaturalEdit through a two-part study. First, a technical evaluation established the soundness of the approach: a benchmark confirmed the viability of our NL-mediated workflow, with performance comparable to that of direct-instruction baselines, while an expert-driven evaluation showed that NaturalEdit consistently generates high-quality artifacts. Second, a controlled lab study with 20 experienced developers provided strong evidence of the usability and cognitive benefits of NaturalEdit.

In summary, we make two contributions: (1) NaturalEdit, a VS Code extension that operationalizes NL-centric code modification by integrating three interaction features into a coherent comprehension-intent-validation workflow, each mechanism addressing a usability obstacle in prior systems and grounded in *Cognitive Dimensions* analysis; and (2) a mixed-methods evaluation showing that the approach is technically viable and improves developers' perceived comprehension, intent specification, sense of control over AI-generated edits and overall modification workflow.

2 Background and Related Work

2.1 Cognitive Demands of Code Modification

Modifying existing code is central to software engineering, yet it remains costly [10, 47]: constructing an internal semantics of existing code alone can consume up to 58% of working time [62]. Developers must then bridge the *Gulf of Execution* [18, 35] by translating intended changes into precise formal syntax, requiring substantial planning and syntax mapping effort [37, 42, 61]. Finally, they must cross the *Gulf of Evaluation* by verifying that modifications reflect their intent without introducing regressions or unintended side effects [3, 5, 33]. Large language models (LLMs) have transformed but not eliminated these bottlenecks. The rise of "vibe coding" [44], where developers delegate implementation through conversational prompts, often bypasses deep comprehension, compressing intent into a prompt and amplifying the "fuzzy abstraction matching" problem [45]. Additionally, validating AI-generated edits overwhelms cognitive load [30, 52], pushing developers toward surface-level execution checks rather than semantic understanding [53]. Together, these pressures disrupt the continuity across comprehension, planning, and validation, underscoring the need for NL interactions that reintegrate rather than further fragment these cognitive workflows.

2.2 NL-Mediated Code Modification

Early NL-mediated systems for exploratory data analysis, including GAM [28], Steps [55], and SQLucid [54], show that translating AI-generated code back into NL can increase developer confidence and reduce ambiguity, but remain restricted to semantically constrained operation spaces. In general-purpose programming, Shi *et al.* [46] introduced NL outlines, which summarize code as structured NL statements at a single fixed granularity. Their *Finish Changes* prototype allows developers to drive code updates by directly rewriting outline text to express desired changes, but this editing workflow was explicitly left out of their evaluation. Beyond these, it provides only coarse navigation between NL and code, and synchronizes changes via full regeneration rather than incremental diffs that isolate what has semantically changed. PInG [11] adopts a similar premise in a more constrained setting, anchoring NL to fixed statement-level inline comments, which further limits the range of cognitive tasks it can support. Both systems treat NL as a direct editing surface: to express a change, the developer must already translate a vague goal into precise summary text, confronting the fuzzy abstraction matching problem [45] without system support.

Recent empirical work by Tang *et al.* [53] documents the concrete usability gaps in this interaction. In a study of 15 developers using Pasta, a minimal summary-mediated prototype, they showed that although modifying summaries can prompt code changes, the workflow exhibits specific shortcomings, including overly rigid representations, implicit NL-code links, and disproportionate editing effort. NaturalEdit addresses these issues by treating NL as the primary artifact through which developers externalize and refine their modification intent before any code is changed, while also mapping code changes back into the NL summary for validation. The interaction design principles behind this approach are grounded in the *Cognitive Dimensions of Notations* (Section 3.1).

3 NaturalEdit

3.1 Design Goals

To derive our design goals, we use the *Cognitive Dimensions of Notations* framework [16] as an analytical lens, scoped specifically to the NL-mediated code modification workflow. Rather than applying the framework exhaustively, we focus on dimensions that surface consistently as key constraints across prior studies of summary-mediated interaction [28, 46, 53], as characterized in Section 2.2. We first synthesize these empirically recurring failures, then use the framework's vocabulary to explain *why* they arise at the level of cognitive mechanism, and to derive principled design goals that address them. Because the framework serves analysis rather than generation, the mapping between dimensions and features is not

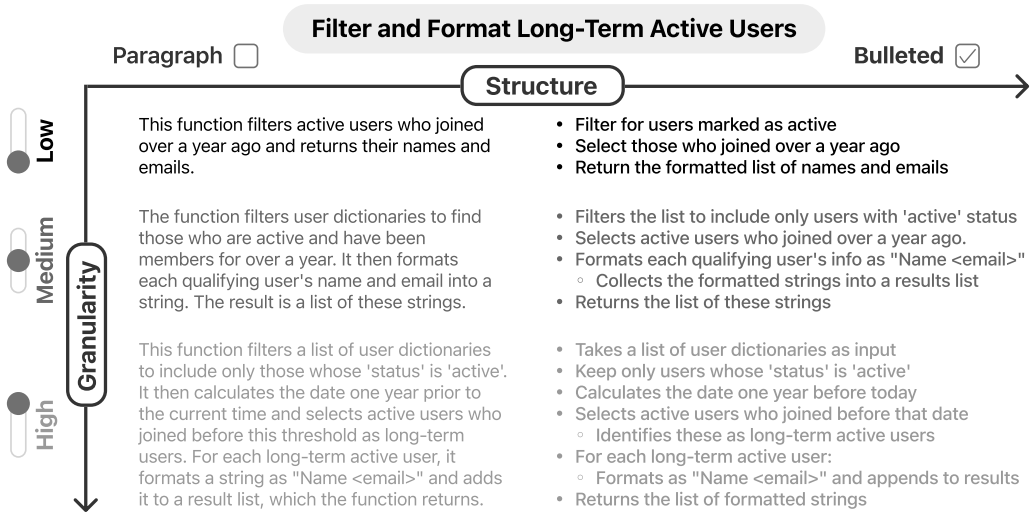


Figure 1: NaturalEdit generates a concise Title and an NL representation adjustable along two dimensions: Structure (paragraph / bulleted), and Granularity (overview, procedural, line-by-line).

one-to-one: e.g., intent-driven synchronization addresses both viscosity and visibility. The analysis was iteratively refined through discussions with two external experts with experience in software engineering research and developer tool design.

DG1. Enable an Adaptive Abstraction Gradient. Effective code modification requires developers to move fluidly across levels of abstraction [57]: from high-level orientation during comprehension, to fine-grained inspection during validation. A system enforcing a fixed *abstraction gradient* cannot support this range of cognitive tasks, and prior studies confirm this as a systemic failure in NL-mediated workflows [53]. Prior systems enforce such rigidity in both NL representation (e.g., PING [11] is anchored to statement-level comments; NL outlines [46] remain static once generated) and code scope selection, which offers no awareness of the surrounding structural hierarchy. NaturalEdit must therefore support dynamic adjustment of the NL representation, with complementary support for fluidly navigating the code scope.

DG2. Ensure Tight and Interactive Mapping. Mentally tracing relationships between NL concepts and code logic places a high load on working memory [20], yet prior systems leave this burden entirely on the developer [53]: most either hide code entirely [28] or provide only coarse, non-interactive links [46]. Furthermore, any such links must remain valid as code evolves through edits, requiring mapping mechanisms grounded in the code's underlying structure. NaturalEdit must therefore externalize NL-code relationships as explicit, fine-grained interactive links that remain structurally stable throughout the modification workflow.

DG3. Minimize Viscosity in Intent Specification. We define *viscosity* as the cognitive and physical effort required to translate high-level intent into concrete summary edits. This effort is structural, not incidental. Prior systems assume that developers can directly edit a fixed NL representation to express desired changes [11, 28, 46], thereby sidestepping the fuzzy abstraction matching problem that makes intent articulation cognitively demanding. Consequently, the NL layer does not mediate between a developer's vague goal and the precise textual edits required to implement it, resulting in high viscosity for non-trivial modifications. NaturalEdit therefore treats intent specification as a system-mediated process, allowing developers to express high-level goals naturally, while the system translates them into concrete NL edits.

DG4. Maintain High Visibility and Consistency. Opaque or inconsistent updates disrupt developers' mental models and undermine trust in AI-generated changes [1, 45, 53]. Prior systems either discard NL after each interaction [11] or preserve it without exposing inspectable diffs [46], leaving developers without a reliable basis for validation. Full regeneration worsens this problem: when the entire summary is rewritten, developers cannot distinguish intended changes from incidental rewording. NaturalEdit therefore presents NL updates as explicit, incremental diffs that isolate only what changed, allowing the summary to remain a persistent basis for validation throughout the modification workflow.

3.2 Key Features

NaturalEdit introduces three key features, organized by role: the NL representation design (Section 3.2.1), the mapping between representations (Section 3.2.2), and the end-to-end modification workflow (Section 3.2.3).

3.2.1 Adaptive Multi-Faceted Representation. To meet **DG1**, NaturalEdit enables NL representation dynamically adjustable along two orthogonal dimensions: *Structure* and *Granularity* (Figure 1).

Structure toggles between prose paragraphs and bulleted lists. Prior work shows that list-style presentations better support procedural understanding [19]. Tang *et al.* [53] also found that developers preferred bullet lists to reason about code behavior and task decomposition, as they mirror code's logic. By contrast, paragraphs support a more holistic, narrative view. Offering both allows developers to match the format to their current cognitive task.

Granularity controls the level of detail, ranging from a high-level overview (what a function does), to a procedural summary (how it

works), to a line-by-line explanation. This progression is analogous to semantic zooming [4, 36], reflecting developers' preference for adjustable detail to balance efficiency and thoroughness [53].

Developers interact with these through simple UI controls: a toggle switch ☑ for Structure and a three-stop slider for Granularity. Internally, NaturalEdit generates a concise **Title** and all six representations (2 structures × 3 granularities) on demand from a single structured LLM prompt, enabling instantaneous, consistent switching.

Fluid Code Scope Navigation. DG1 also requires fluid navigation across code scopes. Instead of requiring repeated manual selection, NaturalEdit supports two complementary interactions: developers can move upward to enclosing scopes by clicking a code region and selecting from its AST path (e.g., method, class), or move downward by clicking a summary segment to select the corresponding code region (Section 3.2.2). Both interactions are enabled by the Structural Alignment Engine (Section 3.4.1).

3.2.2 Interactive Cross-Representation Mapping. To address **DG2**, NaturalEdit replaces the implicit relationship between summary and code with explicit, dynamic links. As shown in Figure 2, these links are fine-grained and responsive. NaturalEdit segments each NL representation into semantic units. Hovering over a segment immediately highlights the corresponding code block; clicking navigates to and selects that scope, decoupling passive comprehension from active navigation. Interactive mappings are generated in parallel for each summary variant. The model receives source code annotated with 1-based line numbers and returns a JSON array of (`summaryComponent`, `codeSegments`) pairs. Each `codeSegments` holds an array of objects, each specifying a code fragment and its line number in the annotated source. Line numbers are essential for precise alignment, especially when identical code fragments appear multiple times.

Critically, these links must remain valid as the code evolves from manual edits, which simple text-based anchoring cannot guarantee. NaturalEdit addresses this limitation through its Structural Alignment Engine (Section 3.4.1), which converts volatile line-based references into persistent AST anchors tied to structural lineage rather than physical position. As a result, tracing between summary and code becomes a low-effort perceptual operation that remains stable throughout the modification lifecycle.

3.2.3 Intent-Driven Bidirectional Synchronization. The final key feature integrates adaptive representation and interactive mapping into a cohesive end-to-end modification workflow (Figure 3).

Intent Articulation and NL Refinement. ❶ The developer issues a high-level instruction. ❷ The system proposes a summary diff (`insertions` in green and ~~`deletions`~~ in red) as an editable intermediate artifact. ❸ The developer validates or refines this NL diff to ensure it precisely captures their intent before any code is modified. This directly addresses **DG3**: a single high-level command suffices for conceptually simple changes, with the intermediate diff making the system's interpretation explicit and correctable.

Code Modification and Bidirectional Synchronization. ❹ Upon approval, the system generates the code change by providing the LLM with the original code alongside both summary versions, directing attention to the developer-approved NL diff. ❺ NaturalEdit then automatically regenerates the NL summary as **Incremental Diffs** (targeted edits that isolate the semantic impact), keeping the summary synchronized with code. When the developer instead edits the code directly, the code is treated as the source of truth, and background validity checking (Section 3.4.1) re-resolves the mapping accordingly.

Multi-Level Validation. ❻ Developers perform *global validation* by inspecting the summary and code diffs side by side. ❼ *Local validation* then leverages interactive mapping directly on highlighted diff regions (e.g., purple highlights in Figure 3), enabling fine-grained verification of NL-code correspondence. This dual-diff loop addresses **DG4** by ensuring visibility and consistency, giving developers confidence that the final program reflects their intent.

3.3 Example Workflow

To illustrate how NaturalEdit integrates its features, we follow Grace as she performs the refactoring task in Figure 4. Her goal is to modify the function `process_user_data`, which *filters active users who joined over a year ago and returns their names and emails*, so that it groups the resulting users by email domain. Ⓐ Grace begins by selecting `process_user_data` in the editor and clicking Summarize Selected Code, which opens a new interactive session.

Ⓑ Grace then switches to a structured, low-granularity view, which presents the function's logic as three bullet points for quick comprehension. She hovers over each point to highlight the corresponding code, building a mental model before deciding an edit scope. She then clicks the edit button to load the summary into the **Modifiable Code Summary** area. Ⓒ Rather than editing the summary directly, Grace types `Group the results by the users' email domain` into the **Edit Instruction** box and clicks ↓ Apply to Summary. The system generates a proposed change, inserting the phrase `grouped by email domain` into the summary. Ⓓ Reviewing the proposed diff, she refines it to additionally specify the return type as ~~`the`~~ `a dictionary of formatted lists`, then approves by clicking ➤. Ⓔ NaturalEdit applies the change and presents a synchronized code diff, while generating a new session with updated NL representations and **Incremental Diffs**. Grace confirms that `collections.defaultdict` is correctly imported and that the grouping logic matches her intent. Ⓕ If the system introduces incorrect logic, she can revert specific lines in the diff view or continue iterating in a new summary session. All sessions remain live while the IDE is open, so Grace can later revisit `process_user_data` without re-summarizing.

3.4 Implementation

NaturalEdit is implemented as a VS Code extension[1], with its main UI rendered in a side panel. This architecture ensures compatibility with all forks of VS Code, e.g., Cursor. The front end is built with React and Vite inside a VS Code Webview and styled with `@vscode/webview-ui-toolkit` for seamless integration into different themes. All NL processing is powered by OpenAI's GPT-4.1[2]; prompt templates are provided in Appendix B. NL summary diffs are computed client-side using `diff-match-patch` library[3];

[1] https://code.visualstudio.com/api
[2] https://openai.com/index/gpt-4-1/
[3] https://github.com/google/diff-match-patch

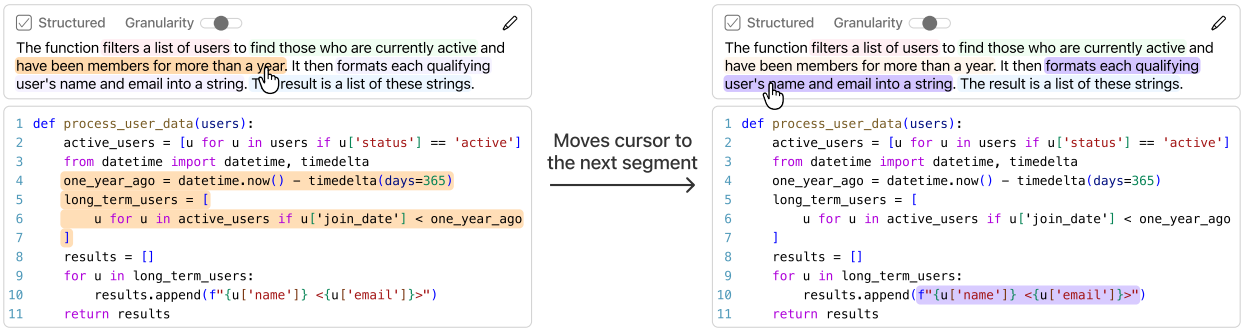


Figure 2: Interactive cross-representation mapping in NaturalEdit. As the user's cursor moves from one semantic segment (e.g., have been members...) to another (e.g., formats each...), the system dynamically updates the corresponding code highlight.

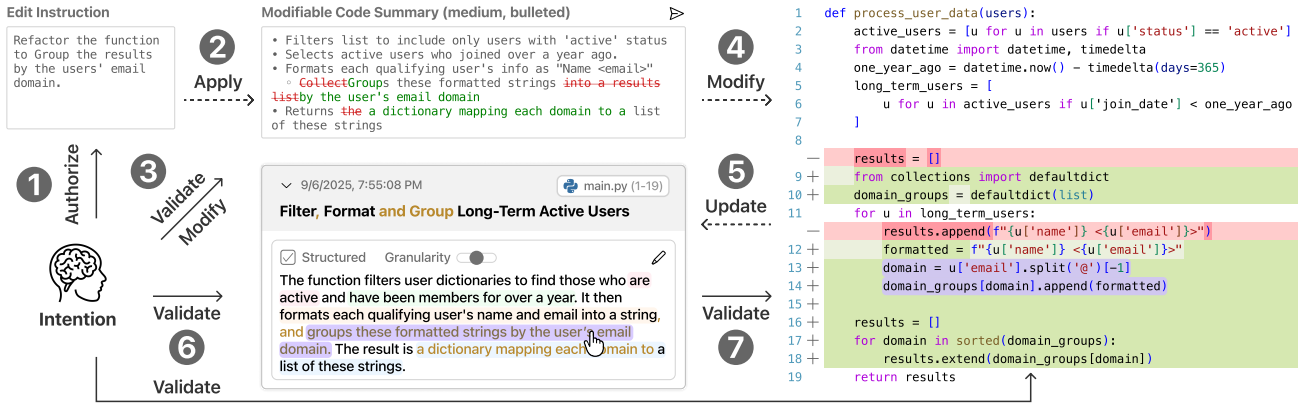


Figure 3: Intent-driven bidirectional synchronization in NaturalEdit: developer actions ⟶ (solid arrows) and system actions ⇢ (dashed arrows) form a cohesive loop connecting intent articulation/refinement, code modification, and validation.

code diffs use VS Code's native `vscode.diff` engine, and editor highlights are rendered with `TextEditor.setDecorations`.

3.4.1 AST-Driven Structural Alignment Engine. To maintain alignment robustness at both the session and mapping levels as code evolves, NaturalEdit replaces volatile line-number and text-based references with persistent ***AST Anchors***: structural descriptors that identify where each summary segment maps to in the code, parsed via Tree-sitter[4]. Each anchor encodes a location through two components: (1) a *structural path* from the file root to the target node, recording the AST type, name, and sibling index at each level; and (2) *target identity*, capturing the AST type, name, and content hash of the minimal AST node enclosing the target code scope (e.g., a function named `process_user_data`). When code is edited, the system re-resolves each anchor by scoring candidates as $s_{\text{path}} \cdot s_{\text{target}}$, traversing the path top-down and matching each level by type, then name, then position, so refactored nodes can still be located and moves among siblings resolve correctly, as position serves only as the final tie-breaker. This keeps interactions focused on code semantics: minor formatting changes, such as added whitespace or blank lines, do not affect the structural path and therefore leave all anchors intact, while meaningful edits that move or rename a code scope are gracefully re-resolved rather than silently misaligned. A summary session remains *Valid* as long as its anchors can be successfully re-resolved; otherwise, it is flagged as *Outdated* in the UI, prompting the developer to refresh. This mechanism supports all Tree-sitter languages, including Python and JavaScript; unsupported languages fall back to fuzzy text matching.

[4]https://tree-sitter.github.io/tree-sitter/

4 Technical Evaluation

We conducted a two-phase technical evaluation: a benchmark assessment verifying that NL-mediated modification preserves accuracy comparable to direct-instruction baselines, and an expert evaluation of the intermediate NL artifacts (summaries, mappings, and diffs) central to the user experience.

4.1 Code Editing Benchmark Evaluation

4.1.1 Experimental Setup. We evaluated on two standard Python code editing benchmarks, both providing NL instructions and unit tests for correctness validation. *CanItEdit* [7] provides 105 hand-crafted tasks, each in two instruction styles: *Lazy* instructions are brief and potentially ambiguous, resembling human-authored requests, while *Descriptive* instructions specify the intended change in detail. *EditEval* [27] provides 194 tasks from GitHub commits, HumanEval [9], and MBPP [2]. We compared *Direct Instruction*

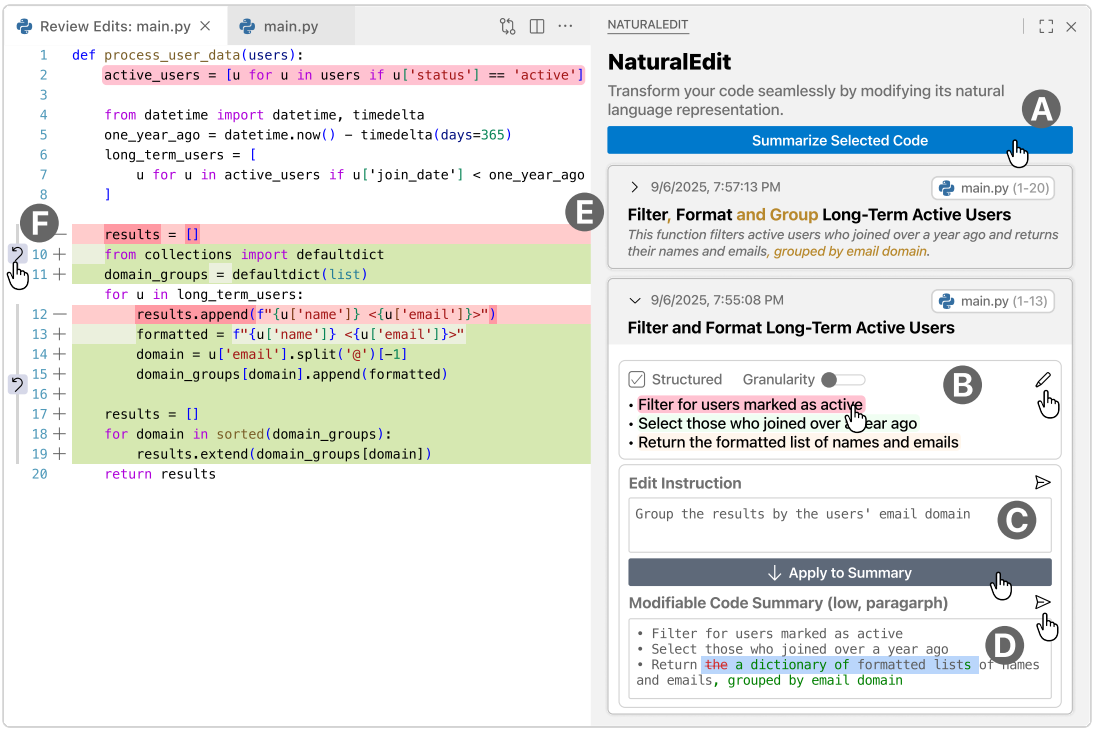


Figure 4: The NaturalEdit interface during the example workflow, showing key steps: summarizing code (A), interacting with the NL representation (B), issuing an instruction (C), validating the NL diff (D), reviewing the synchronized code diff with Incremental Diffs (E), and reverting unintended changes (F).

(single-step code generation from instruction) against *NL Mediation* (NaturalEdit's two-step workflow across six representation variants). Both conditions used GPT-4.1 with prompts based on NaturalEdit (Appendix B), with minor adjustments for the benchmark format (e.g., removing unavailable file context). We report *Pass@1*, the proportion of tasks where the first generated code passes all of the benchmark's ground-truth unit tests.

4.1.2 Results. Figure 9 (Appendix C.1) reports Pass@1 for all six representation variants on each benchmark; here we summarize the aggregate patterns. **The NL-mediated workflow preserves technical soundness.** NaturalEdit achieves performance broadly comparable to the Direct Instruction baseline across all benchmarks (average −1.92%, maximum −8.98%), suggesting that an intermediate NL layer does not fundamentally compromise code correctness. We interpret the modest decrease as the cost of the intermediate NL representation, which requires the model to summarize and reinterpret both the code and the instruction. In most cases, errors arose when this representation omitted or simplified key semantics, such as boundary conditions or constraints needed for correct edits. This suggests a trade-off between NL readability and the semantic precision required for accurate code modification.

NL mediation is more robust to vague instructions. Compared with the baseline, NL mediation performs better on *Lazy* instructions (+0.48%) than on *Descriptive* ones (-4.45%). This suggests that grounding vague intent in an explicit NL representation reduces the risk of underspecification.

4.2 Expert Ratings of Intermediate Artifacts

4.2.1 Evaluation Protocol. We used the six code modification tasks from our user study (Section 5.1.3) as the corpus, as they involve multi-file edits and better reflect the real-world scenarios NaturalEdit is designed to support. For each task, we generated six representation variants for both the initial code and the ground-truth final solution; final-solution summaries were produced as incremental diffs against the original. This yielded a corpus of 72 summaries (36 with diffs) and 487 summary-code mappings.

Drawing on prior work in code summarization [56, 58], the two authors collaboratively defined criteria for each artifact: *Accuracy* (correctness of description) and *Clarity* (readability) for summaries; *Segmentation Granularity* (appropriateness of segment size), *Accuracy* (correctness of NL-code links), and *Coverage* (completeness of links) for mappings; and *Faithfulness* (accuracy of diff representation), *Completeness* (capture of all semantic changes), and *Salience* (prominence of important changes) for diffs. The full rating guide is provided in Appendix C.2.

Two computer science researchers, one specializing in AI model evaluation and the other in developer tools, conducted the assessment. They independently scored all items on a 5-point scale (1 = Very Poor, 5 = Excellent). We assessed inter-rater reliability using percent agreement, which is better suited than Cohen's kappa for highly skewed data [13], e.g., when the majority of ratings are 5. We report both strict (exact-match) and relaxed (within ±1 point) agreement: 70.4% (96.3%) for Summary Quality, 74.1% (99.1%) for Diff

Quality, and 94.3% (98.0%) for Mapping Quality. Final scores were determined by averaging ratings that differed by one point and resolving larger disagreements through consensus. These discussions also informed a qualitative analysis of recurring error patterns.

4.2.2 Results. Figure 10 (Appendix C.3) shows ratings for all eight criteria across the six variants. **The generated NL artifacts were consistently rated as high quality.** The majority (42/48) of mean scores fell between 4.5 and 5.0 on a 5-point scale, confirming the viability of our pipeline for producing usable NL representations.

However, segmentation utility decreased at high granularity in unstructured summaries. Raters scored *Segmentation Granularity* significantly lower for the high-granularity unstructured variant ($M = 4.08$, $SD = 0.74$; $W = 2.5$, $p = .038$). Raters observed that, at this level of detail, consecutive highlighted blocks often corresponded to nearly identical code regions, suggesting that coarser semantic segmentation would be more effective.

A critical trade-off emerged between diff detail and salience. While *Faithfulness* and *Completeness* remained highly rated, *Salience* declined with increasing granularity and was lowest in the high-granularity unstructured variant ($M = 3.75$, $SD = 1.08$). Raters attributed this to two factors: (1) the large volume of insertions, deletions, and rewordings made important changes harder to distinguish from minor ones, and (2) action-oriented phrasing (e.g., "*Current price... added to the response*") occasionally broke the summary's "objective description" metaphor.

5 User Study

We conducted a controlled lab study to examine NaturalEdit's usability and cognitive impact on code modification, focusing on how its three core features shape developers' comprehension, intent specification, and validation workflows[5].

5.1 Study Design

5.1.1 Participants. We recruited 20 participants (12 male, 8 female; ages 21–52, $M = 27.15$, $SD = 8.25$) through purposive sampling [12]. All had substantial experience with Python and JavaScript. The sample included two undergraduates, 11 graduate students, and seven industry professionals, with an average of 9.55 years of programming experience ($SD = 9.16$). All 13 student participants majored in Computer Science or Engineering, and eight reported prior internship experience. Each participant received a $66 Amazon gift card. Participants reported using LLM-powered programming tools either "frequently" (7) or "almost always" (13), including ChatGPT (18), GitHub Copilot (15), Cursor (11), and Claude Code (9). Most also had relevant domain experience in the study scenarios: web development (20/20) and machine learning (18/20). A detailed demographic summary is provided in Table 1 (Appendix D.1).

5.1.2 Baseline. To examine whether NaturalEdit's key features improve on existing NL-mediated systems, we built a controlled baseline in which the three core interaction designs were disabled: *Adaptive Multi-Faceted Representation* was replaced with a fixed, medium-level paragraph summary; *Interactive Cross-Representation Mapping* was removed; and *Intent-Driven Synchronization* was reduced to manual summary editing. This ablated design is functionally comparable to prior systems such as Pasta [53] and GAM [28], and it was implemented using the same VS Code framework and UI style as NaturalEdit (screenshot in Figure 11, Appendix D.2).

5.1.3 Programming Tasks. We designed two programming tasks in full-stack web development and machine learning, implemented in JavaScript and Python, respectively. Tasks are summarized here and described in full in Appendix D.3:

- **Task 1.** *Finance Dashboard*: A Node.js/React stock price visualization application. The subtasks included (a) implementing a helper function to format the tick labels on the x-axis, (b) adding the current price to the backend API, and (c) displaying it as a reference line on the chart.
- **Task 2.** *MVP Predictor*: A Python pipeline for scraping NBA player data and predicting MVP rankings. The subtasks included (a) extending the scraper with advanced statistics, (b) adjusting the `n_estimators` hyperparameter of the `XGBRanker` model, and (c) improving the type and color scheme of the visualization chart.

Task realism was validated through participant ratings, with all 20 participants rating the tasks at least 4 on a 7-point Likert scale. Across tasks, participants worked with code at varying scopes, ranging from sub-function regions to multiple related code segments across files. To minimize bias in how participants instructed the system, we designed the task descriptions following the Natural Programming Elicitation method [31]. Each task was presented with annotated target visuals and brief textual descriptions, preventing the direct reuse of the task descriptions as instructions.

5.1.4 Study Protocol. We conducted a within-subjects user study comparing NaturalEdit against *Baseline*. Both conditions were deployed in CodeSandbox[6], a web-based fork of VS Code where researchers can preconfigure Node.js and Python environments, eliminating local setup issues and screen-sharing latency. The IDE theme was set to Default Light+ for visual consistency. Participants accessed the environment through provided links, with task instructions shown on a separate device to minimize context switching.

Each session lasted approximately 90 minutes. Participants first signed a consent form and completed a pre-study questionnaire on demographics and programming experience. To reduce performance anxiety and encourage authentic behavior [43], we told participants that the study evaluated the systems rather than their performance. Task and condition order were counterbalanced to mitigate learning and order effects [34]. Before each condition, participants completed a guided onboarding example, and then worked on the task for 25 minutes.

We used a mixed-methods approach for data collection. The environment automatically logged all interactions to Firebase, including timestamps, feature usage events, and LLM response contexts; sessions were also screen- and audio-recorded. After each condition, participants completed the NASA-TLX [17] cognitive workload measure and self-reported their understanding of both the original code and the system-generated modifications.

After both tasks, participants completed a final questionnaire with parallel sections for each condition, including the UMUX-Lite

[5]Approved by the Institutional Review Board (IRB) at our institution.

[6]https://codesandbox.io/

usability scale [26] and custom items on key modification stages and NaturalEdit's interactive features. The session concluded with a 20-minute semi-structured interview covering participants' typical modification workflows, their experiences with each of NaturalEdit's core features, and their perceived control over and trust in the system's output, along with follow-up questions grounded in observed behaviors.. Full questionnaire items and the interview protocol are provided in Appendices D.4 and D.5.

5.1.5 Data Analysis. For qualitative analysis, we transcribed the interviews and followed an iterative, discussion-based thematic analysis process [24, 60]. One author coded 50% of the data to develop a preliminary codebook, and a second author then coded the same subset, refining the codes through discussion. Using the unified codebook, the two authors collaboratively coded the remaining data and continuously discussed interpretations to maintain agreement. Because the analysis was discussion-based, we did not calculate a formal inter-coder reliability metric [29]. The complete codebook is provided in Appendix D.7.

For quantitative analysis, we used nonparametric tests appropriate for small samples. Within-subject comparisons between NaturalEdit and *Baseline* were conducted using the Wilcoxon signed-rank test ($\alpha = .05$). We report the median for each condition (Mdn_N for NaturalEdit and Mdn_B for *Baseline*), the test statistic W, and the p-value. To analyze interaction behaviors, we constructed transition graphs from user logs by preprocessing raw events into higher-level actions. Two filters were applied: (1) mapping hover events with dwell times under 500 ms were excluded, and consecutive hovers were merged into `Inspect Mapping` actions; (2) consecutive `Adapt Summary Level` events were collapsed into a single action reflecting the final state. Results are shown in Figure 7.

5.2 Study Results

NaturalEdit achieved higher task correctness than *Baseline* (95.0% vs. 81.7%), while completion times were comparable (*Baseline*: 19.30± 5.21 min; NaturalEdit: 20.85 ± 3.80 min; $W = 59.5$, $p = .256$). Despite similar time investment, behavioral data reveal systematic differences in interaction strategies (Figure 7). Participants also self-reported significantly higher satisfaction with NaturalEdit's code modifications (Q11: $Mdn_N = 7.0$, $Mdn_B = 5.0$; $W = 0.0$, $p = .003$).

5.2.1 Overall Usability & Experience (RQ1). **NaturalEdit demonstrated significantly higher usability, and participants reported they would prefer it for real-world development.** The overall SUS score for NaturalEdit ($Mdn_N = 77.07$) was significantly higher than *Baseline* ($Mdn_B = 66.23$, $W = 34.5$, $p = .026$). Participants reported that NaturalEdit better met their requirements ($Mdn_N = 6.0$, $Mdn_B = 4.5$, $W = 11.0$, $p = .004$) and was significantly more useful in real development work ($Mdn_N = 6.0$, $Mdn_B = 4.0$, $W = 8.0$, $p = .001$). Perceived ease of use and learnability were comparable across conditions ($p > .3$), suggesting that NaturalEdit preserved the simplicity of the bare-bones *Baseline* despite its additional features. Cognitive workload (NASA-TLX) showed no significant differences across all six dimensions ($p > .1$).

NaturalEdit shifted modification strategies toward an NL-centric workflow. As shown in Figure 7, 72/104 modifications were completed via the modified summary. In contrast, participants in *Baseline* avoided manual summary edits entirely (95/111 modifications via direct-instruction fallback). This shift reflects reduced viscosity: participants described manual summary editing in *Baseline* as "*cumbersome and taking significant effort*" (P5).

NaturalEdit enhanced developers' sense of control via enforced comprehension and shared grounding. (Q10: $Mdn_N = 6.0$, $Mdn_B = 5.0$, $W = 7.50$, $p = .008$). NaturalEdit encouraged a "comprehend-then-act" strategy [47], contrasting with vibe coding practices [15]. Participants also reported significantly better perceived understanding of AI-generated edits with NaturalEdit ($Mdn_N = 5.5$, $Mdn_B = 4.5$, $W = 15.0$, $p = .009$). The NL summary served as a shared ground for validating intent and system interpretation, making the "*black box more transparent*" (P11) and giving developers "*greater confidence it would work*" (P12).

NL mediation diverged between declarative and procedural tasks. Participants rated the NL representation in NaturalEdit as significantly more useful ($Mdn_N = 7.0$, $Mdn_B = 5.0$, $W = 0.0$, $p = .001$), with over 800 NL-mediated actions recorded. For declarative tasks (e.g., UI changes), participants preferred direct instructions, as visual feedback reduced the need for deep comprehension (P2: "*I care more about the final effect*"). For procedural tasks (e.g., backend algorithms), participants consistently used the summary-mediated path, where correctness required semantic validation beyond execution results (P12: "*strongly need additional representation to help understand changes*").

5.2.2 Abstraction Gradient (RQ2). **Adaptive representation supported top-down comprehension through on-demand granularity adjustment.** NaturalEdit significantly improved self-reported code comprehension (Q5: $Mdn_N = 6.5$, $Mdn_B = 4.5$, $W = 0.0$, $p = .0003$), with 19 participants rating the feature 6 or 7 (Figure 13). Participants appreciated starting from a "*coarser view for initial understanding*" (P3) before drilling into details (P1, P2, P3, P8, P15, P20). However, interaction logs show they rarely switched to the lowest granularity (Figure 8), suggesting the default medium level was usually sufficient as an entry point.

Developers generally preferred high-granularity summaries for modification, while using granularity opportunistically. As shown in Figure 8, participants often switched to and committed edits at the high-granularity level, valuing its comprehensiveness as a basis for editing (P3, P7, P9, P10, P12, P16, P19), and its support for incremental changes that felt less likely to "*affect other parts*" (P8, P10). Yet this preference was not rigid: for high-level changes, a concise summary likely provided sufficient scaffolding.

Most participants preferred bulleted summaries for their procedural alignment and scannability, though paragraphs were valued for narrative coherence. Across all granularity levels, participants more often switched to structured NL (Figure 8, left), finding it better aligned with code structure and easier to skim (P1, P3, P5, P6, P8, P10, P11, P12, P16, P17, P20). Some, however, preferred paragraphs for their logical continuity (P2, P4, P7, P8, P15), noting that bullet lists could feel "*fragmented in overall logic*" (P7) and obscure causal relationships between sentences.

5.2.3 Closeness of Mapping (RQ3). **Interactive mapping was the central action in the workflow, serving as the primary bridge between NL and code.** With 488 occurrences, it was the most

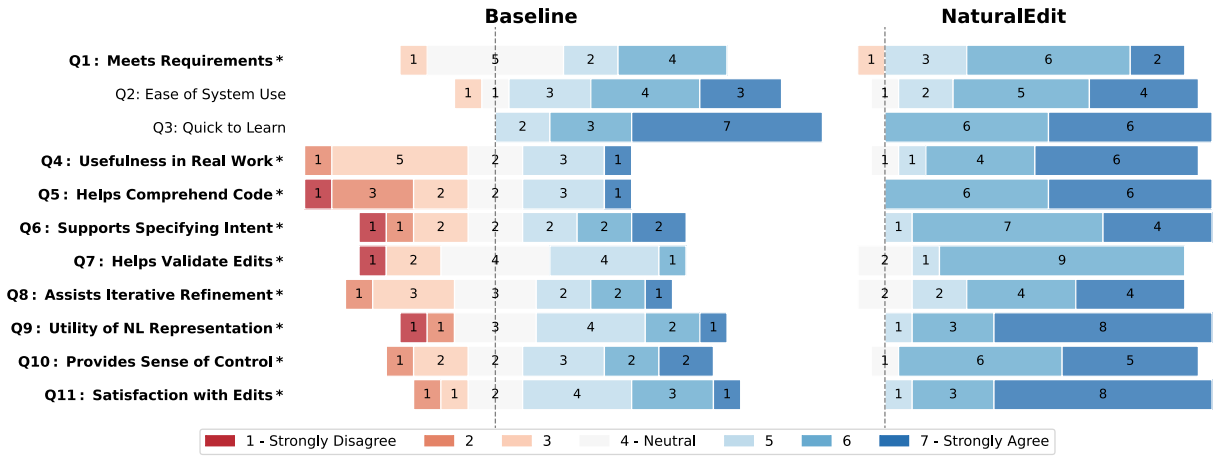


Figure 5: Comparison of usability (Q1–Q2, UMUX-LITE) and utility (Q3–Q11) between *Baseline* and NaturalEdit. Questions with statistically significant differences ($p < 0.01$) are marked with an asterisk (*), based on Wilcoxon signed-rank tests.

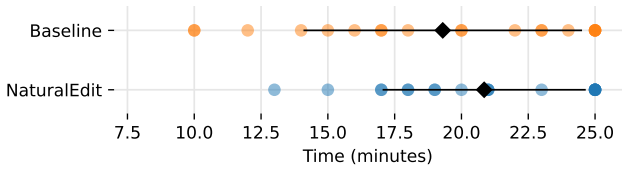


Figure 6: Distribution of time spent on tasks per participant.

frequent action in NaturalEdit (Figure 7), and 15/20 participants strongly agreed that it made the code-summary relationship explicit (Figure 13). Mapping also played a pivotal transitional role: it was the most common first action after generating a summary (94/140) or switching sections (17/20), and the most common action both before (158/190) and after (181/190) adapting granularity. Crucially, mapping guided active, sequential comprehension that prepared participants for modification (P3, P5, P7, P9, P17, P20): it was the most common action immediately before initiating a summary edit (60/79) or direct instruction (29/32). As P5 noted, it "*helped me not only understand the code but also plan how to change it.*"

Mapping externalized the cognitive load of mentally tracing NL-code relationships and extended its value across the full modification lifecycle. P2 noted that, while typical AI tools require "*the human brain to do the mapping*," NaturalEdit "*automatically did the mapping, which reduced the burden.*" Developers also preferred faster localization by being able to "*just click on it, and it would take me to the corresponding chunk of code.*" Backed by AST-based structural alignment, the mappings remained stable and accurate throughout sessions, even as the code evolved through manual edits. After NaturalEdit generated a code change, developers' most common first action was to inspect the updated mapping (44/68), relying on it to validate the automated changes. As P12 noted, it could "*guide me to quickly verify if the summary is correct.*"

5.2.4 Viscosity, Visibility & Consistency (RQ4). **NaturalEdit's intent-driven workflow reduced the viscosity of intent specification by integrating the flexibility of direct instructions into summary editing.** 14/20 participants strongly agreed that expressing intent through high-level instructions was natural and efficient (Figure 13), supported by significantly higher ratings for intent specification (Q6: $Mdn_N = 6.0$, $Mdn_B = 4.5$, $W = 1.50$, $p = .002$). As shown in Figure 7, 58/79 `Commit Modified Summary` actions followed directly from `Apply Instruction to Summary` without manual refinement, with P5 noting that it "*did a lot of the heavy lifting.*" The intermediate summary diff further supported intent clarification: initial instructions were concise ($Mdn = 12$ words), whereas the corresponding summary edits were significantly longer ($Mdn = 18$ words, $W = 556.0$, $p < .001$; see Figure 14). This expansion was not merely an implementation detail; it actively resolved ambiguity before code generation. For instance, when P1 vaguely instructed the system to find the *best* NDCG score, the summary revised this to *highest*, clarifying the metric's directionality. Similarly, P3 noted that the process turned a "*vague, conversational request*" into a "*more specific summary.*" Although the system occasionally misinterpreted subtle instructions, e.g., by hard-coding literal month abbreviations ("*Jan, Feb*") instead of inferring a general date-formatting rule (P3, P12), the intermediate layer made such failures visible and correctable before code generation.

Bidirectional synchronization provided consistency and visibility that developers valued highly (Figure 13). NaturalEdit received significantly higher ratings for validating modified code (Q8: $Mdn_N = 6.0$, $Mdn_B = 4.0$, $W = 3.50$, $p = .001$) and supporting iterative refinement (Q9: $Mdn_N = 6.0$, $Mdn_B = 5.0$, $W = 13.50$, $p = .004$). Participants compared the auto-updated diff to reviewing a GitHub commit for "*accidental changes*" before merging (P6), and relied on it to "*see at a glance where the modification was*" (P8) rather than "*re-reading everything*" (P5).

6 Discussion and Design Implications

6.1 Richer and Malleable NL Representations

Our study showed that structured NL representations support comprehension: developers preferred bulleted summaries because they

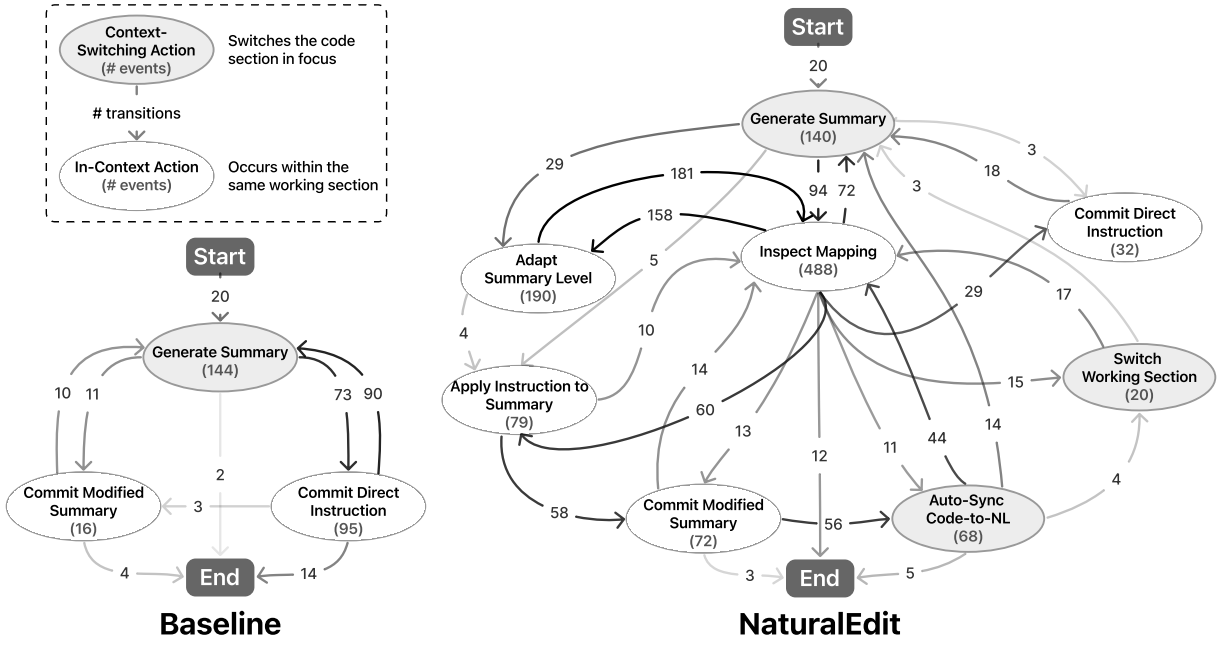


Figure 7: Transition graphs of user workflows in the *Baseline* (left) and NaturalEdit (right) conditions. Both graphs represent a complete code modification episode, with explicit start and end states. Nodes represent user actions, with occurrence counts shown in parentheses. Edges represent transitions between actions, with opacity log-scaled by transition frequency (darker represents more common pathways). For clarity, transitions that occurred only once between intermediate actions are omitted.

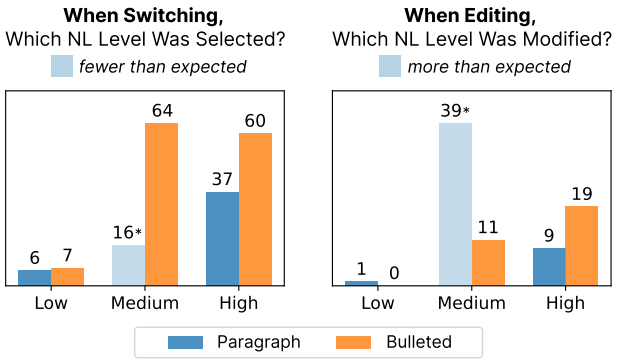


Figure 8: Usage counts across NL granularity and structure, considering only explicit switches or edits. *Medium-Paragraph* (lighter blue, *) reflects a default-view bias: underrepresented in switches but overrepresented in edits.

align with code's procedural logic (Section 5.2.2), consistent with Information Foraging Theory [38], which posits that structured cues reduce the "cost of access" to relevant information. However, NaturalEdit represents only a first step. As P2 observed, bulleted lists convey "*parallel and hierarchical relationships*" but struggle to capture causality, while codebases are inherently non-linear, combining *tree* structures (files, classes, and methods) with *graph* structures (call dependencies and data flow). Future NL representations should therefore move beyond simple lists to incorporate richer forms that better reflect how developers navigate large codebases [14, 23].

Beyond structure, these representations should also be malleable and personalized [6, 8, 41]. From a cognitive perspective, NaturalEdit's six predefined variants already enable "epistemic actions" [21]: developers reconfigure the representation to offload cognitive processing by adjusting information density to match their working memory capacity. Yet participants expressed a need for deeper customization; for example, P2 preferred function signatures over descriptive summaries at low granularity. This suggests that future systems should both implicitly learn developers' information needs across contexts (e.g., expertise, task complexity) and support proactive customization of representation types.

6.2 Generalizing to Spec-Driven Development

While NaturalEdit focuses on localized code modification, its design principles connect to the emerging paradigm of *spec-driven development*, formalized in tools such as GitHub Spec Kit[7] and Amazon Kiro[8], where lightweight NL documents (e.g., `design.md`, `tasks.md`) serve as the primary medium for human-AI interaction, guiding implementation and validating code against written constraints [50]. Both workflows face a common challenge: maintaining alignment between an NL representation of intent, AI-generated code, and developer understanding [51]. NaturalEdit's three design principles address this alignment problem: *adaptive representation* supports navigating NL artifacts across levels of abstraction; *interactive mapping* can link spec items to their corresponding implementations, making NL documents live and navigable rather

[7]https://github.github.com/spec-kit/
[8]https://aws.amazon.com/documentation-overview/kiro/

than static; and *intent-driven bidirectional synchronization* enables developers to modify NL to drive code changes while reflecting those changes back, creating a tighter feedback loop than current tools provide. More broadly, this points to a *three-way alignment problem* common to both workflows: developers must verify not only that the code is correct, but also that the NL artifact captures their intent and that the AI's actions remain consistent with it. NaturalEdit's diff and validation workflow is designed to make this misalignment visible and resolvable.

6.3 An Ecosystem for NL-Centric Programming

Our findings point to two directions for extending NaturalEdit into a broader programming ecosystem. First, NL representations should surface runtime errors: as P9 suggested, when code fails, the system should highlight the problematic logical segment in the summary, thereby integrating debugging into the comprehension-editing workflow [63]. Second, the non-linear nature of AI-assisted exploration calls for lightweight personal history management beyond the current diff-based revert mechanism, including support for exploratory branching, path comparison, and selective merging. This may be particularly important for vibe coding [44], where developers iteratively refine large-scale edits (P5, P8, P12, P17).

7 Limitations and Future Work

Implementation Constraints. As a prototype, NaturalEdit has several practical limitations. First, its interactive responsiveness is constrained by the latency of the underlying LLM. Although we applied optimizations such as request parallelization, feedback is not instantaneous and can disrupt the fluidity of the interaction loop. Second, our approach incurs substantial computational cost, which may be prohibitive for larger codebases. Future work should explore more efficient methods (e.g., pre-indexing) to establish coarse-grained alignment between code and summary before invoking a more expensive model for fine-grained mapping. Third, summary sessions do not persist across IDE restarts; project-level persistence is left as future work.

NaturalEdit can also exhibit instability when regenerating summaries: as reflected in the expert ratings (Section 4.2.2), diff salience decreases at finer granularity, where great change volume and inconsistent phrasing obscure the most important edits. Future work should explore semantically controlled generation techniques, e.g., extending constrained decoding [59] from structurally simple settings to semantically richer representations. Additionally, while AST anchor re-resolution degrades gracefully by design (Section 3.4.1), its robustness has yet to be systematically quantified.

Threats to User Study Validity. Our within-subjects comparison used a feature-ablated baseline, which supports comparison against prior NL-mediated approaches but cannot isolate the contribution of individual mechanisms. We instead examined each feature through per-dimension ratings (Figure 13), interview quotes, and transition analysis of interaction logs (Figure 7), where actions map onto individual features. A full factorial ablation remains future work. We also did not compare against commercial AI coding tools (e.g., Cursor, Copilot), as differences in underlying models, mature UIs, and participants' prior proficiency would introduce confounds that obscure attribution. Our goal is to offer design principles that complement such tools rather than outperform them (Section 6.2), and we leave a direct comparison as future work.

Our findings on confidence and control are also subject to automation bias: a readable NL diff may increase perceived control while still concealing semantic errors or inaccurate NL-code mappings. Two observations partially mitigate this concern: task correctness with NaturalEdit exceeded the baseline (95.0% vs. 81.7%), and the editable summary diff exposed misread instructions before code generation (Section 5.2.4). However, our logs do not capture the intent behind summary edits, so we could not quantify how often the diff misrepresented an instruction or how often developers caught such misreads; both questions warrant dedicated study.

External validity is further limited by task scope and participant sample. Although participants found the tasks realistic, they were necessarily bounded by a lab session and could not capture the complexity of long-term software maintenance, reflecting the trade-off between experimental control and ecological validity in programmer studies [22, 48]. In addition, predefined modification goals bypassed the open-ended problem identification and diagnosis common in real-world development. Although our 20 participants varied in experience, the sample may still not represent the broader developer population.

Moreover, part of our analysis relies on self-reports, such as interviews and usability ratings. To mitigate this, we triangulated these findings with behavioral data throughout Section 5.2: e.g., self-reported value of mapping by its 488 logged occurrences as the most frequent action. As in many lab studies of interactive systems, participants' perceptions may also have been influenced by the novelty of the interaction design. Future work should therefore evaluate our approach through longitudinal, real-world deployments [64] with a larger and more diverse participant pool to examine how usage patterns evolve and whether novelty effects persist. Because NaturalEdit is implemented as a VS Code extension, it is well-suited to such in-situ evaluation in realistic development settings.

8 Conclusion

We present NaturalEdit, a system that operationalizes design principles for making static NL summaries usable as interactive representations for code modification. Grounded in the *Cognitive Dimensions of Notations*, NaturalEdit introduces three interaction mechanisms to reduce cognitive load and enhance developer control. Through a two-stage technical evaluation and a controlled user study with 20 developers, we show that the approach is both technically viable and improves the developer experience. Our work offers validated design principles for future NL programming tools and points toward richer representations of code.

Acknowledgments

This research was supported in part by a Google Cloud Research Credit Award, a Google Research Scholar Award, a gift from Adobe, an Edison Innovation Fellowship from the Notre Dame IDEA Center, and NSF grants CCF-2211428, CCF-2315887, CCF-2100035, CCF-2211429, and CCF-2442682. Any opinions, findings, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N Bennett, Kori Inkpen, et al. 2019. Guidelines for human-AI interaction. In *Proceedings of the 2019 chi conference on human factors in computing systems*. 1–13.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.

[4] Benjamin B Bederson and James D Hollan. 1994. Pad++ a zooming graphical interface for exploring alternate interface physics. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*. 17–26.

[5] Frederick P Brooks Jr. 1995. *The mythical man-month: essays on software engineering*. Pearson Education.

[6] Yining Cao, Peiling Jiang, and Haijun Xia. 2025. Generative and Malleable User Interfaces with Generative and Evolving Task-Driven Data Model. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–20.

[7] Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. 2024. Can It Edit? Evaluating the Ability of Large Language Models to Follow Code Editing Instructions. In *Conference on Language Modeling (COLM)*.

[8] Meng Chen and Amy Pavel. 2026. TaskArtisan: Designing Composable Generative Widgets for LLM-Assisted Analysis. *arXiv preprint arXiv:2607.17394* (2026).

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[10] Françoise Détienne. 2001. *Software design–cognitive aspect*. Springer Science & Business Media.

[11] Yifeng Di and Tianyi Zhang. 2025. Enhancing Code Generation via Bidirectional Comment-Level Mutual Grounding. *arXiv preprint arXiv:2505.07768* (2025).

[12] Ilker Etikan, Sulaiman Abubakar Musa, Rukayya Sunusi Alkassim, et al. 2016. Comparison of convenience sampling and purposive sampling. *American journal of theoretical and applied statistics* 5, 1 (2016), 1–4.

[13] Alvan R Feinstein and Domenic V Cicchetti. 1990. High agreement but low kappa: I. The problems of two paradoxes. *Journal of clinical epidemiology* 43, 6 (1990), 543–549.

[14] Scott D Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 2 (2013), 1–41.

[15] Francis Geng, Anshul Shah, Haolin Li, Nawab Mulla, Steven Swanson, Gerald Soosai Raj, Daniel Zingaro, and Leo Porter. 2025. Exploring Student-AI Interactions in Vibe Coding. *arXiv preprint arXiv:2507.22614* (2025).

[16] Thomas RG Green. 1989. Cognitive dimensions of notations. *People and computers V* (1989), 443–460.

[17] Sandra G Hart. 1986. NASA task load index (TLX). (1986).

[18] Edwin L Hutchins, James D Hollan, and Donald A Norman. 1986. Direct manipulation interfaces. In *User centered system design*. CRC Press, 87–124.

[19] Joyce Karreman and Nicole Loorbach. 2007. Paragraphs or Lists? The Effects of Text Structure on Web Sites. In *2007 IEEE International Professional Communication Conference*. IEEE, 1–5.

[20] David Kirsh. 2010. Thinking with external representations. *AI & society* 25, 4 (2010), 441–454.

[21] David Kirsh and Paul Maglio. 1994. On distinguishing epistemic from pragmatic action. *Cognitive science* 18, 4 (1994), 513–549.

[22] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20 (2015), 110–141.

[23] Amy J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering* 32, 12 (2006), 971–987.

[24] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. 2017. *Research methods in human-computer interaction*. Morgan Kaufmann.

[25] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th international conference on program comprehension*. 184–195.

[26] James R Lewis, Brian S Utesch, and Deborah E Maher. 2013. UMUX-LITE: when there's no time for the SUS. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 2099–2102.

[27] Kaixin Li, Qisheng Hu, James Zhao, Hui Chen, Yuxi Xie, Tiedong Liu, Michael Shieh, and Junxian He. 2024. Instructcoder: Instruction tuning large language models for code editing. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*. 50–70.

[28] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D Gordon. 2023. "What it wants me to say": Bridging the abstraction gap between end-user programmers and code-generating large language models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–31.

[29] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. *Proceedings of the ACM on human-computer interaction* 3, CSCW (2019), 1–23.

[30] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–16.

[31] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.

[32] Brad A Myers, John F Pane, and Amy J Ko. 2004. Natural programming languages and environments. *Commun. ACM* 47, 9 (2004), 47–52.

[33] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.

[34] Jakob Nielsen. 1994. *Usability engineering*. Morgan Kaufmann.

[35] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic books.

[36] Ken Perlin and David Fox. 1993. Pad: an alternative approach to the computer interface. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 57–64.

[37] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. 2013. An empirical study of API usability. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 5–14.

[38] Peter Pirolli and Stuart Card. 1999. Information foraging. *Psychological review* 106, 4 (1999), 643.

[39] Strategic Planning. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology* 1, 2002 (2002), 1.

[40] John W Ratcliff, David E Metzener, et al. 1988. Pattern matching: The gestalt approach. *Dr. Dobb's Journal* 13, 7 (1988), 46.

[41] Alexander Richter and Kai Riemer. 2013. Malleable end-user software. *Business & Information Systems Engineering* 5, 3 (2013), 195–197.

[42] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[43] Jeffrey Rubin and Dana Chisnell. 2008. *Handbook of usability testing: How to plan, design, and conduct effective tests*. John Wiley & Sons.

[44] Advait Sarkar and Ian Drosos. 2025. Vibe coding: programming through conversation with artificial intelligence. *arXiv preprint arXiv:2506.23253* (2025).

[45] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[46] Kensen Shi, Deniz Altınbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, et al. 2024. Natural language outlines for code: Literate programming in the llm era. *arXiv preprint arXiv:2408.04820* (2024).

[47] Ben Shneiderman and Richard Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences* 8, 3 (1979), 219–238.

[48] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on internal and external validity in empirical software engineering. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 9–19.

[49] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. 2–13.

[50] Ningzhi Tang, Chaoran Chen, Zihan Fang, Gelei Xu, Maria Dhakal, Yiyu Shi, Collin McMillan, Yu Huang, and Toby Jia-Jun Li. 2026. Programming by Chat: A Large-Scale Behavioral Analysis of 11,579 Real-World AI-Assisted IDE Sessions. *arXiv preprint arXiv:2604.00436* (2026).

[51] Ningzhi Tang, Chaoran Chen, Gelei Xu, Yiyu Shi, Yu Huang, Collin McMillan, Tao Dong, and Toby Jia-Jun Li. 2026. How coding agents fail their users: A large-scale analysis of developer-agent misalignment in 20,574 real-world sessions. *arXiv preprint arXiv:2605.29442* (2026).

[52] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. Developer behaviors in validating and repairing llm-generated code using ide and eye tracking. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 40–46.

[53] Ningzhi Tang, Emory Smith, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2025. Exploring Direct Instruction and Summary-Mediated Prompting in LLM-Assisted Code Modification. In *Proceedings of the 2025 IEEE Symposium on Visual*

Languages and Human-Centric Computing (VL/HCC). IEEE. arXiv:2508.01523 To appear.
[54] Yuan Tian, Jonathan K Kummerfeld, Toby Jia-Jun Li, and Tianyi Zhang. 2024. Sqlucid: Grounding natural language database queries with interactive explanations. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. 1–20.
[55] Yuan Tian, Zheng Zhang, Zheng Ning, Toby Jia-Jun Li, Jonathan K Kummerfeld, and Tianyi Zhang. 2023. Interactive text-to-SQL generation via editable step-by-step explanations. *arXiv preprint arXiv:2305.07372* (2023).
[56] Christoph Treude, Justin Middleton, and Thushari Atapattu. 2020. Beyond accuracy: Assessing software documentation quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1509–1512.
[57] Bret Victor. 2011. Up and down the ladder of abstraction. *Retrieved September* 2 (2011), 2015.
[58] Robert Wallace, Aakash Bansal, Zachary Karas, Ningzhi Tang, Yu Huang, Toby Jia-Jun Li, and Collin McMillan. 2025. Programmer visual attention during context-aware code summarization. *IEEE Transactions on Software Engineering* (2025).
[59] Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702* (2023).
[60] Michael Williams and Tami Moser. 2019. The art of coding and thematic exploration in qualitative research. *International management review* 15, 1 (2019), 45–55.
[61] Niklaus Wirth. 1971. Program development by stepwise refinement. *Commun. ACM* 14, 4 (1971), 221–227.
[62] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.
[63] Yinuo Yang, Ashley Ge Zhang, Steve Oney, and April Yi Wang. 2026. SPARK: Real-Time Monitoring of Multi-Faceted Programming Exercises. *arXiv preprint arXiv:2601.22256* (2026).
[64] Yinuo Yang, Zheng Zhang, Ningzhi Tang, Xu Wang, Alex Ambrose, Nathaniel Myers, Patrick Clauss, and Toby Jia-Jun Li. 2026. Lessons from real-world deployment of a cognition-preserving writing tool: Students actively engage with critical thinking and planning affordances. *arXiv preprint arXiv:2603.15777* (2026).

A Data and Code Availability

To support transparency and reproducibility, we have made our research artifacts publicly available in a replication package[9].

This package contains: (1) the complete source code for NaturalEdit[10] and *Baseline*; (2) materials for the technical evaluation, including the benchmark evaluation and the expert rating; (3) materials for the user study, including study tasks and all anonymized raw data (interaction logs and questionnaire responses); and (4) analysis artifacts, including the scripts for data analysis and visualization. All figures, tables, and statistical results can be fully reproduced using the provided scripts. The NaturalEdit extension is available in the VS Code Marketplace.[11]

B Prompt Templates

Listing 1: Prompts for Generating Multi-Faceted NL Representation

```
You are an expert code summarizer. For the following code,
     generate 6 summaries, one for each combination of
     detail level (low, medium, high) and structure (
     unstructured, i.e., paragraph, structured, i.e.,
     bulleted):
- low_unstructured: One-sentence, low-detail, paragraph
     style.
- low_structured: 2-3 short bullet points, low-detail, as a
     single string. Each bullet must start with "•" and be
     separated by \n. Never return an array.
- medium_unstructured: 2-3 sentences, medium-detail,
     paragraph style.
- medium_structured: 3-5 bullet points, medium-detail, as a
     single string. Use "•" for first-level bullets, and
     ENCOURAGE the use of two-level bullets (use "○" for the
      second level, and indent the second-level bullet with
     2 spaces before the "○") when logical groupings exist.
     Bullets must be separated by \n. Never return an array.
- high_unstructured: 3-4 sentences, high-detail, paragraph
     style.
- high_structured: 4-8 bullet points, high-detail, as a
     single string. Use "•" for first-level bullets, and
     ENCOURAGE the use of two-level bullets (use "○" for the
      second level, and indent the second-level bullet with
     2 spaces before the "○") when logical groupings exist.
     Bullets must be separated by \n. Never return an array.

IMPORTANT:
- For medium_structured and high_structured, if there are
     logical groupings, you should use two-level bullets ("•
     " and "○"). For the second-level bullet ("○"), always
     indent with 2 spaces before the "○".
- The file context below is provided ONLY for reference to
     help understand the code's environment.
- Your summary MUST focus ONLY on the specific code snippet
     provided.
- Return your response as a JSON object with keys: title,
     low_unstructured, low_structured, medium_unstructured,
     medium_structured, high_unstructured, high_structured.

File Context (for reference only):
${fileContext}

Code to summarize:
${code}
```

Listing 2: Prompts for Building Interactive Cross-Representation Mapping

```
You are an expert at code-to-summary mapping. Given the
     following code and summary, extract up to 10 key
     summary components (phrases or semantic units) from the
      summary.

IMPORTANT:
1. Each summaryComponent you extract MUST be a substring (
     exact part) of the summary text below.
2. Extract summaryComponents in the exact order they appear
     in the summary text.
3. Do NOT hallucinate or invent summary components that do
     not appear in the summary.

For each summaryComponent, extract one or more relevant code
      segments from the code that best match the meaning of
     the summary component.
- For each code segment, return both the code fragment (as a
      string) and its line number in the original code (1-
     based).
- Prefer to use a complete code statement (such as a full
     line, assignment, function definition, or block) as the
      code segment if it clearly represents the summary
     component's meaning.
```

[9] https://github.com/ND-SaNDwichLAB/naturaledit-replication-package
[10] Also available at https://github.com/TTangNingzhi/NaturalEdit.
[11] https://marketplace.visualstudio.com/items?itemName=sandwich-lab.naturaledit

```
- If a full statement is not appropriate or would be
     ambiguous, you should use a smaller, relevant fragment
     (such as a variable, function name, operator, or part
     of an expression).
- Only include enough code to make the mapping meaningful
     and unambiguous.
- If a code segment contains multiple lines, split them into
      separate objects in the codeSegments array.

Return as a JSON array of objects:
[
  {
    "summaryComponent": "...",
    "codeSegments": [
      { "code": "code fragment 1", "line": 12 },
      { "code": "code fragment 2", "line": 15 }
    ]
  },
  ...
]

Code (with line numbers for reference):
${codeWithLineNumbers}

Summary:
${summaryText}
```

Listing 3: Prompts for Applying Edit Instruction on Modifiable Summary

```
You are an expert at editing code summaries. In this
     scenario, a developer is using a summary-mediated
     approach to modify code:
1. Instead of directly editing the code, the developer
     modifies the summary to express their desired code
     behavior.
2. The modified summary will later be used to generate the
     actual code changes.
3. Your task is to integrate the developer's instruction
     into the summary, making it clear what the new code
     should do.

Given the following original summary and a direct
     instruction, update the summary to incorporate the
     developer's intent:
- The code context below is provided ONLY for reference to
     help understand the summary's environment.
- Preserve the parts of the original summary that are not
     affected by the instruction.
- Maintain the original summary format (sentence, bullet
     points, etc.).
- Make it easy to identify what changed by keeping unchanged
      parts exactly as they were.
- Integrate the instruction seamlessly into existing
     sentences or bullet points as much as possible.
- However, add new sentences or bullet points if the
     instruction cannot be naturally integrated into
     existing ones.
- The updated summary MUST clearly express what the new code
      should do, incorporating ALL information from the
     instruction.
- Output only the updated summary, nothing else.

Code Context (for reference only):
${originalCode}

Original summary:
${originalSummary}

Developer's instruction (integrate this intent FULLY into
     the updated summary):
${instruction}

Updated summary:
```

Listing 4: Prompts for Code Modification via Edited NL Summary

```
You are an expert code editor. Given the following original
     code and an updated summary (detail level: ${
     detailLevel}, structure: ${structuredType}), update the
      code to reflect the changes in the new summary.
- The file context below is provided ONLY for reference to
     help understand the code's environment, and your code
     changes MUST focus ONLY on the specific code snippet
     provided.
- Only change the code as needed to match the new summary,
     and keep the rest of the code unchanged.
- Preserve the leading whitespace (indentation) of each line
      from the original code in the updated code. For any
     modified or new lines, match the indentation style and
     level of the surrounding code.
- Pay close attention to the differences between the
     original summary and the edited summary, which reflects
      developer's intent of what the new code should be.
- Output only the updated code, nothing else.

File Context (for reference only):
${fileContext}

Original code:
${originalCode}

Original summary (detail level: ${detailLevel}, structure: $
     {structuredType}):
${originalSummary}

Updated summary (detail level: ${detailLevel}, structure: ${
     structuredType}):
${editedSummary}

Updated code:
```

Listing 5: Prompts for Generating NL Representation with Incremental Diffs

```
You are an expert code summarizer. Your task is to generate
     a new summary for the MODIFIED code below, using the
     original code and its previous summary as reference.

Instructions:
- Your new summary MUST focus on the code differences (
     addition, deletion) between the original and modified
     code and clearly reflect those changes, even if they
     are small, such as inline comments.
```

```
- Make the changed parts of the summary easy to identify (e.
     g., by being explicit about what changed, or by using
     wording that highlights the update). I mean, rather
     than describing the change itself (e.g., updated the
     function to ...), seamlessly integrate the changes into
      the new summary in one coherent, descriptive sentence.
- The new summary should be close to the old summary, only
     updating the parts that are affected by the code change
     : If a part of the summary is still accurate for the
     new code, keep it unchanged; If a part of the summary
     is no longer accurate, change only that part to reflect
      the new code. Do not add unnecessary changes or
     rephrase unchanged parts.
- For all structured (bulleted) summaries, return as a
     single string. Each bullet must start with "•" and be
     separated by \\n. For medium_structured and
     high_structured, if there are logical groupings, you
     should use two-level bullets ("•" and "◦"). For the
     second-level bullet ("◦"), always indent with 2 spaces
     before the "◦". Never return an array.
- Return your response as a JSON object with keys: title,
     low_unstructured, low_structured, medium_unstructured,
     medium_structured, high_unstructured, high_structured.

File Context (for reference only):
${fileContext}

Original code:
${originalCode}

Old summary:
{
  "title": "${oldSummary.title}",
  "low_unstructured": "${oldSummary.low_unstructured}",
  "low_structured": "${oldSummary.low_structured}",
  "medium_unstructured": "${oldSummary.medium_unstructured
       }",
  "medium_structured": "${oldSummary.medium_structured}",
  "high_unstructured": "${oldSummary.high_unstructured}",
  "high_structured": "${oldSummary.high_structured}"
}

MODIFIED code:
${newCode}
```

C Technical Evaluation Materials

C.1 Benchmark Evaluation Results

The full results of the benchmark performance of the NL-mediated modification (Section 4.2.2) are shown in Figure 10.

C.2 Expert Rating Guide

Overview. This document provides the definitions and criteria for evaluating the quality of the intermediate artifacts generated by the NaturalEdit system. The goal is to systematically assess the quality of the NL summaries, their mapping to the source code, and the diffs between summary versions. All items will be rated on a 5-point scale, where the meaning of each point is defined for the specific dimension being evaluated. As a general guide:

- **5: Excellent / Strongly Agree** - The artifact is of very high quality, with no significant issues.
- **4: Good / Agree**
- **3: Acceptable / Neutral** - The artifact is adequate for its purpose but has noticeable room for improvement.
- **2: Poor / Disagree**
- **1: Very Poor / Strongly Disagree** - The artifact is of low quality and has significant, potentially misleading, flaws.

Part A: Summary Quality. (This is rated for each of the 36 buggy code summaries and the 36 ground-truth code summaries.)

A.1. Accuracy: Does the summary correctly describe the functionality and logic of the corresponding code?

- **5 (Excellent):** Perfectly accurate. It correctly captures all key logic, operations, and outcomes without any errors or misinterpretations.
- **3 (Acceptable):** Mostly accurate. It captures the main purpose of the code but may contain minor omissions or slight inaccuracies that do not fundamentally mislead the reader.
- **1 (Very Poor):** Inaccurate. It contains significant factual errors, describes functionality that doesn't exist, or completely misses the core purpose of the code.

A.2. Clarity: Is the summary written in clear, unambiguous, and easy-to-understand language for a developer?

- **5 (Excellent):** Very clear, concise, and well-written. A developer could understand the code's function almost instantly.
- **3 (Acceptable):** Generally understandable, but the language might be slightly awkward, verbose, or contain jargon that could be simplified.
- **1 (Very Poor):** Unclear, confusing, grammatically incorrect, or so full of jargon that it is difficult to parse.

Part B: Segmentation & Mapping Quality. (This part has two components: ratings for each segmentation as a whole for 72 summaries, and ratings for each of the 478 individual mapping highlights.)

B.1. Segmentation Granularity: Is the code broken down into segments of an appropriate and useful size?

- **5 (Excellent):** The granularity is perfect. Each segment is a meaningful, well-sized logical chunk.
- **3 (Acceptable):** The granularity is mostly fine, but some segments might be slightly too large (coarse-grained) or too small and fragmented.
- **1 (Very Poor):** The segmentation is not useful. It is either one giant, monolithic block or is excessively fragmented into meaningless single lines or tokens.

B.2. Mapping Accuracy (Precision): Is this specific link that connects the summary segment to the code correct?

- **5 (Correct):** The link is perfectly correct. The code highlight is directly relevant to the summary segment.
- **1 (Incorrect):** The link is wrong. The code highlight is irrelevant to the summary segment.

B.3. Mapping Coverage (Recall): Does this summary segment successfully link to all the relevant parts of the code that implement the concept it describes?

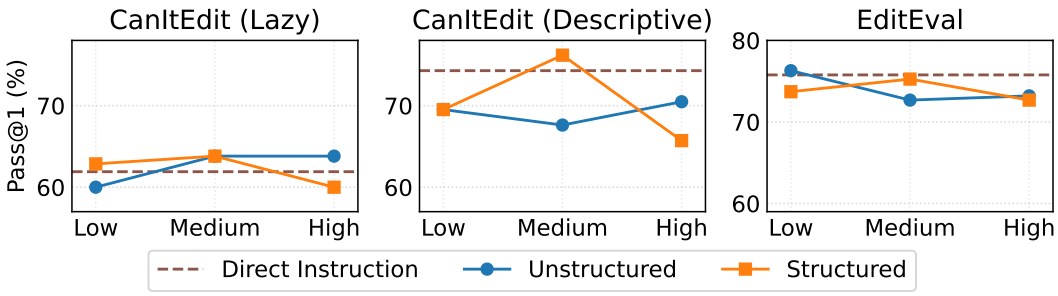


Figure 9: Pass@1 of our NL-mediated workflows (blue and red lines) against the Direct Instruction baseline (dashed brown line), evaluated on CanItEdit [7] (using both *Lazy* and *Descriptive* instructions) and EditEval [27] benchmarks (Section 4.2.2).

- **5 (Complete Coverage):** Yes. The summary segment is linked to every piece of relevant code. No relevant code segments are missed.
- **3 (Minor Omissions):** Mostly. The main, most critical code blocks are linked, but a minor, less important part might be missing (e.g., a related variable declaration or a peripheral logging statement).
- **1 (Major Omissions):** No. Key code blocks that directly implement the summary segment's description are completely unlinked.

Part C: Summary Diff Quality. (This is rated for each of the 36 diffs between the before-and-after summary pairs.)

C.1. Faithfulness: Does the change in the summary accurately represent the change that occurred in the code?

- **5 (Excellent):** A perfect representation. The summary diff mirrors the semantic code change exactly.
- **3 (Acceptable):** Mostly faithful. It captures the main idea of the code change but might misrepresent a minor detail or nuance of the implementation.
- **1 (Very Poor):** Unfaithful. The diff is misleading, inaccurate, or completely unrelated to the actual change in the code.

C.2. Completeness: Does the summary diff capture all of the important semantic changes from the code diff?

- **5 (Excellent):** Fully complete. No important aspect of the code change is missing from the summary diff.
- **3 (Acceptable):** Mostly complete. It captures the primary functional change but might omit a secondary change (e.g., a change to a variable name or a log message that accompanies a logic change).
- **1 (Very Poor):** Incomplete. It completely misses one or more significant functional changes made in the code.

C.3. Salience: Does the summary diff make the most important aspects of the change stand out clearly and effectively?

- **5 (Excellent):** Highly salient. The change is presented clearly and concisely, making the core of the modification immediately obvious (e.g., a single, clear bullet point is added).
- **3 (Acceptable):** Moderately salient. The change is correctly represented, but it might be buried in other minor textual edits or phrased in a way that requires careful reading to understand the key point.
- **1 (Very Poor):** Not salient. The important change is obscured, minimized, or lost in a sea of trivial rewording, making it hard for a user to spot the key difference.

C.3 Expert Rating Results

The full results of the expert evaluation of intermediate representations (Section 4.2.2) are shown in Figure 10.

D Controlled User Study Materials

D.1 Participants

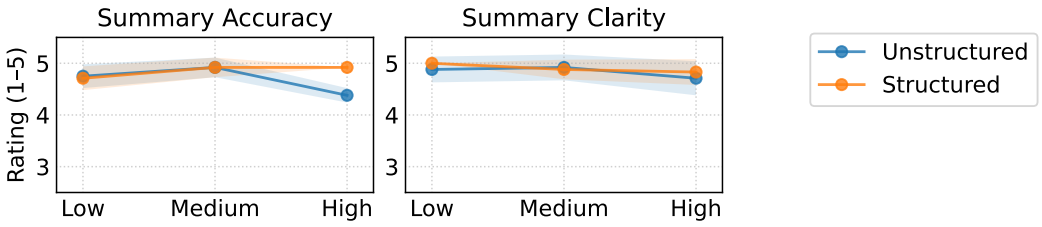
Table 1 provides a detailed summary of participant demographics.

Table 1: Summary of Participant Demographics.

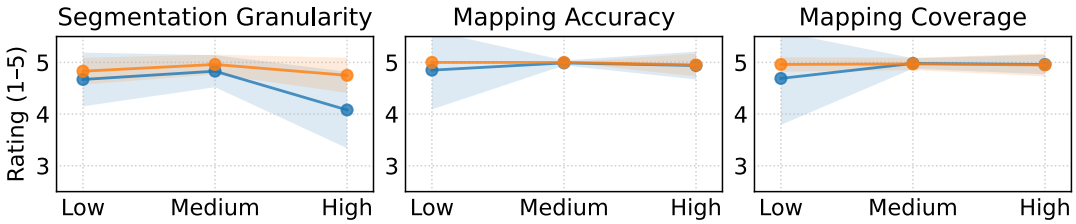
ID	Gender	Age	Occupation	Experience
P1	Male	23	PhD Student	5 years
P2	Male	25	Software Engineer	8 years
P3	Male	24	Graduate Student	6 years
P4	Male	25	PhD Student	6 years
P5	Female	27	PhD Student	11 years
P6	Female	52	IT Professional	30 years
P7	Female	22	PhD Student	5 years
P8	Female	22	Software Engineer	6 years
P9	Male	21	Undergraduate Student	3 years
P10	Female	21	Undergraduate Student	3 years
P11	Male	26	R&D Engineer	7 years
P12	Male	24	PhD Student	8 years
P13	Female	28	PhD Student	7 years
P14	Male	23	PhD Student	6 years
P15	Male	48	Software Engineer	40 years
P16	Male	25	PhD Student	8 years
P17	Female	30	Software Engineer	10 years
P18	Male	30	R&D Engineer	12 years
P19	Female	24	PhD Student	4 years
P20	Male	23	PhD Student	6 years

D.2 Baseline

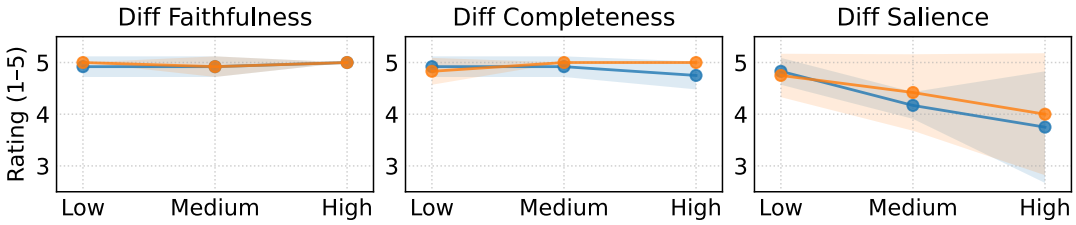
Figure 11 shows a screenshot of the baseline.



(a) Summary quality across representation variants.



(b) Mapping quality between NL summaries and code.



(c) Diff quality for NL summary updates.

Figure 10: Expert ratings of NaturalEdit artifacts: points indicate mean ratings, and shaded regions represent standard deviations (Section 4.2.2).

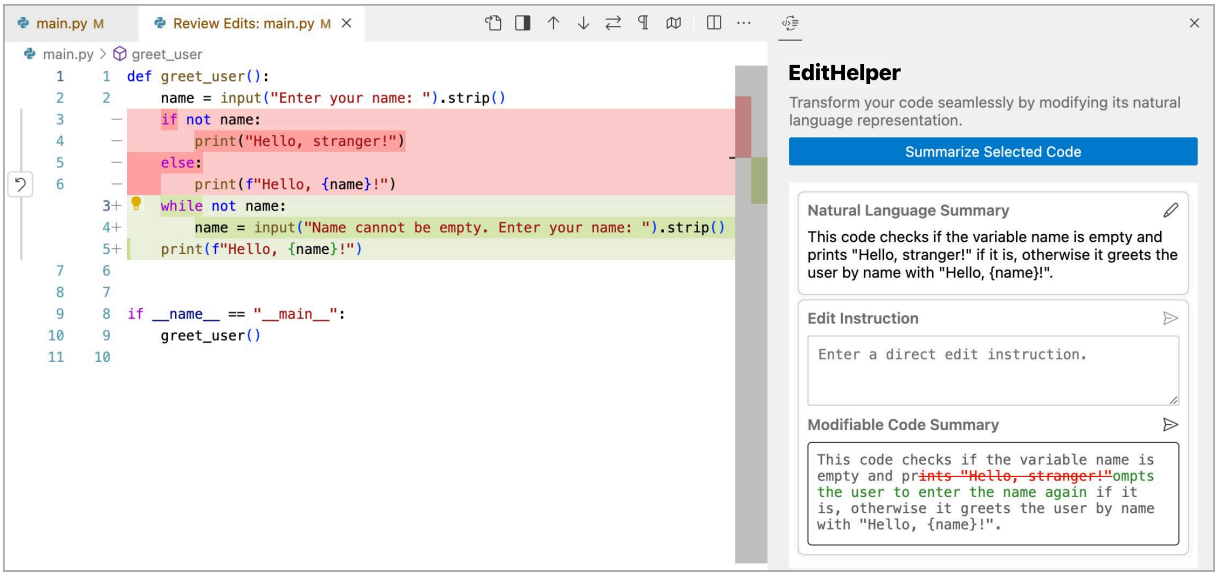


Figure 11: A screenshot of the baseline system, implemented as an ablation of NaturalEdit.

D.3 Programming Tasks

Figure 12 presents screenshots of the task descriptions used in the study; full task code is provided in the replication package.

D.4 Questionnaires

D.4.1 NASA-TLX. (1=Very low, 7=Very high) unless otherwise noted

1. **Mental Demand:** How mentally demanding was the task?
2. **Physical Demand:** How physically demanding was the task?
3. **Temporal Demand:** How hurried or rushed was the pace of the task?
4. **Performance:** How successful were you in accomplishing what you were asked to do? *(1=Perfect, 7=Failure)*
   Note: Participants were informed that a lower score means perfect, while a higher score indicates failure.

D.4.2 UMUX-LITE. (1=Strongly disagree, 7=Strongly agree)

1. This system's capabilities meet my requirements.
2. This system is easy to use.

D.4.3 Self-Defined Likert Scale Items. (1=Strongly disagree, 7=Strongly agree)

Post-Task Questions *(Asked immediately after each task, alongside the NASA-TLX)*

1. I have a high level of understanding of the original code before it was modified.
2. I have a high level of understanding of the code modifications made by the system.

Task Realism *(Asked once at the end of the entire session, the same applies below)*

1. The study tasks felt as realistic as my daily programming.

Evaluation of both systems (NaturalEdit and *Baseline*)

1. I could quickly learn how to use the system.
2. I would use this system in my real development work if it were available.
3. The system helps me comprehend the original code.
4. The system supports me in specifying my intentions.
5. The system helps me understand and validate the modified code.
6. The system assists with the iterative refinement of code.
7. The natural language representation is useful for achieving the task goal.
8. I felt a good sense of control over the system's behavior.
9. I am generally satisfied with the code modifications produced by the system.

Evaluation of NaturalEdit's specific features

1. The adaptive and multifaceted summaries helped in understanding the code at different levels.
2. The interactive mapping between summary and code made their relationship explicit and easy to follow.
3. By applying direct instructions to the summary, I was able to express my intentions flexibly and efficiently.
4. The auto-updated summary with visual diffs helped me validate the changes in a consistent workflow.

D.5 Semi-Structured Interview Protocol

The following questions served as a guide for our semi-structured interviews. This protocol was used flexibly; we often prioritized follow-up questions based on direct observations of a participant's behavior and their real-time commentary over strictly adhering to this script.

Specific Observations
- "I observed you did [action] when [situation]. Could you elaborate on your thought process there?"

General Code Modification Workflow
- What are your main challenges when modifying existing, unfamiliar code (e.g., understanding its logic, validating changes)?
- What is your typical workflow for modifying existing code, and what are your main concerns during that process (e.g., introducing hidden bugs)?

Feedback on NaturalEdit's Core Features
- Compare the experience of modifying code via its NL representation to your usual workflow of editing code directly.
- How did the different summary views (in terms of structure and granularity) affect your workflow? Did you have a preference?
- How useful was the interactive mapping between NL and code for understanding or modification?
- Did the automatically updated NL summary, with its visual diff, help you validate code changes?

Perception of Output Quality and Control
- Did you notice any quality differences between the modified code and the generated NL summaries?
- Describe any moments where you felt more or less in control of the system's output.

Contextual Factors
- How did your familiarity with the codebase affect your interaction with the system?
- Did the type of modification or the clarity of your goal influence how you used the system?
- When might NL-based features be most or least useful, depending on the code's complexity?

Special Role (For industry professionals)
- How might this NL-based approach fit into your daily work, and what impact could it have?

Concerns and Future Work
- What are the main drawbacks of NaturalEdit, and what would prevent you from using it in your real work?
- Did anything about the study tasks feel artificial or unrealistic?
- How can this approach evolve to support large, multi-file projects?

D.6 Additional Study Results

Figure 13 shows participants' ratings of NaturalEdit's four core interactive features along the corresponding cognitive dimensions.

Figure 14 shows the distribution of word-level differences between participants' instructions and the corresponding summary edits generated by NaturalEdit.

D.7 Qualitative Codebook

This appendix presents the full codebook used for qualitative analysis. Codes are organized into thematic categories.

Comprehension Strategies for AI Code

Task 1: Finance Dashboard ← Link

Overview

A web development project that visualizes stock prices using a combination of a `Node.js` backend, which retrieves data from Yahoo Finance, and a `React` frontend, which displays the data in a line chart.

How to Run

1. Frontend: `cd frontend && npm run dev`
2. Backend: `cd server && node index.js`

Subtasks

Figure 1. Current interface awaiting modification.

A. X-axis Tick Marks

`frontend/src/StockChart.jsx` Implement the helper function `formatDateString` and apply it!

B. Current Price Data

`server/index.js -> /api/stock/:symbol` The frontend also wants to know the current stock price!

C. Current Price Line

`frontend/src/StockChart.jsx` Create a ReferenceLine for it and match the expected style!

Figure 2. Expected result interface for your tasks!

(a) Task 1: Finance Dashboard

Task 2: MVP Predictor ← Link

Overview

A machine learning pipeline ranks NBA players. The data, scraped from basketball-reference.com, is processed and then used to train a gradient boosting model that outputs predicted MVP rankings.

How to Run

Directly run `scraper.py`, `preprocess.py`, and `rank_model.py` in order.

You should see a folder called `data` with the statistics and graphs.

Subtasks

A. + Advanced Metrics

`scraper.py -> scrape_season_stats` Scrap (existing) basic stats + advanced stats & Merge them!

- Basic (2025): https://www.basketball-reference.com/leagues/NBA_2025_per_game.html
- Advanced (2025): https://www.basketball-reference.com/leagues/NBA_2025_advanced.html

Merge Keys | Basic Stats | Advanced Stats

```
Year,Name,MVP_Points,PTS,AST,TRB,FG%,BPM,WS/48,TS%,VORP,WS,PER,Rank
2018,James Harden,965,30.4,8.8,5.4,0.449,9.9,0.289,0.619,7.7,15.4,29.8,1.0
2018,LeBron James,738,27.5,9.1,8.6,0.542,8.7,0.221,0.621,8.2,14.0,28.6,2.0
2018,Anthony Davis,445,28.1,2.3,11.1,0.534,6.7,0.241,0.612,5.9,13.7,28.9,3.0
2018,Damian Lillard,207,26.9,6.6,4.5,0.439,7.2,0.227,0.594,6.3,12.6,25.2,4.0
2018,Russell Westbrook,76,25.4,10.3,10.1,0.449,6.3,0.166,0.524,6.1,10.1,24.7,5.0
2018,Giannis Antetokounmpo,75,26.9,4.8,10.0,0.529,6.2,0.207,0.598,5.7,11.9,27.3,6.0
2018,Kevin Durant,66,26.4,5.4,6.8,0.516,7.3,0.215,0.64,5.5,10.4,26.0,7.0
2018,DeMar DeRozan,32,23.0,5.2,3.9,0.456,2.8,0.17,0.555,3.2,9.6,21.0,8.0
2018,LaMarcus Aldridge,6,23.1,2.0,8.5,0.51,3.9,0.209,0.57,3.7,10.9,25.0,9.0
```

Figure 3. Example of scraped and pre-processed data `data/merged_data.csv`

Please rerun `scraper.py` and `preprocess.py` now.

B. Tune Model Parameters

`rank_model.py -> main` Which `n_estimators` is best for XGBRanker from `100, 1000, 2000`? Use it to make the final prediction and create visualizations!

```
➜  workspace git:(main) ✗ /usr/local/bin/python /project/workspace/rank_model.py
Training XGBoostRanker with 100 estimators...
NDCG Score with 100 estimators: 0.7428
Training XGBoostRanker with 1000 estimators...
NDCG Score with 1000 estimators: 0.7508
Training XGBoostRanker with 2000 estimators...
NDCG Score with 2000 estimators: 0.7482
Best n_estimators: 1000
    Year                 Name  Rank  PredictedScore  PredictedRank
93  2024          Nikola Jokić   1.0      -12.622600            9.0
94  2024  Shai Gilgeous-Alexander   2.0      -10.613235            7.0
95  2024             Luka Dončić   3.0       -5.107751            5.0
96  2024   Giannis Antetokounmpo   4.0      -10.803287            8.0
```

Figure 4. Example output of running XGBRanker

C. Better Visualization

`rank_model.py -> plot_ranking_predictions` ~~Regular~~ → **Grouped** bar chart (color palette `coolwarm`)

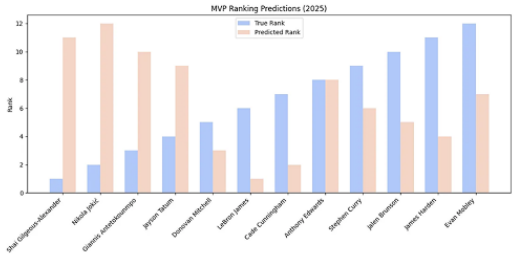


Figure 5. Example visualization `data/ranking_predictions_2025.png`

(b) Task 2: MVP Predictor

Figure 12: Screenshots of task descriptions used in the study.

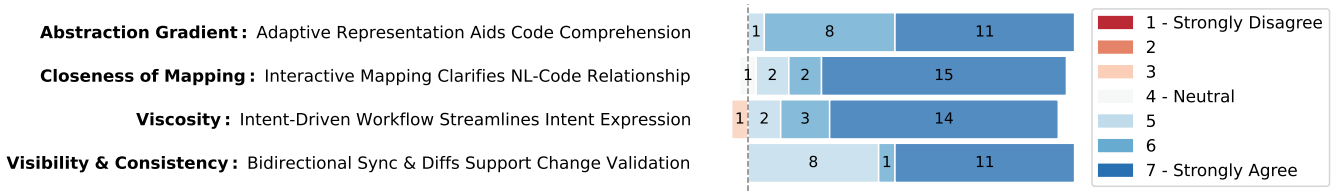


Figure 13: User ratings of NaturalEdit features across four cognitive dimensions.

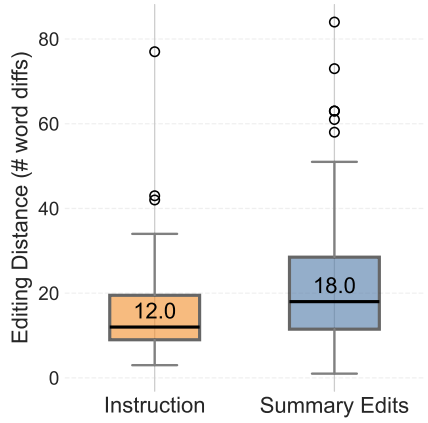


Figure 14: The editing distance of the instruction and the corresponding summary edits generated by NaturalEdit, across all summary granularities, measured as word-level differences using the Gestalt pattern matching [40]. Median values are shown inside each box.

- Different needs for understanding code semantics
- Avoid comprehension of code due to cognitive workload
- Preference for concise and clear summaries
- Reading the summary helps understanding
- Reading the summary involves a high workload
- Macro-level comprehension prioritized over line-level
- Top-down workflow: start with macro, then narrow to details

Validation Strategies for AI Modifications

- Read the code to validate AI edits
- Run the code to validate AI edits
- Summarize edits with AI to assist validation
- Use AI to refine or fix edits directly
- Run tests to validate AI edits

Direct Edit Instructions for AI Modification

- Direct instructions are natural and convenient
- Direct instructions are usually vague and casual
- Distrust in the accuracy of direct instructions

Modifiable Code Summaries for AI Modification

- Manually editing summaries is a high workload
- Manual edits are more prone to error
- Summaries lack intuitive edit points
- Editing decisions depend on the simplicity of modifications

Applying Edit Instruction to the Summary

- It makes editing intuitive and reduces workload
- Intermediate edited summaries pre-validate modifications
- Summaries help refine or clarify vague intent
- Summaries help understand LLM edits
- Summaries increase sense of control over edits
- Less trust in LLM's ability to focus on summary differences
- Preference for automation without manual confirmation
- Trade-off between controlling logic and trusting instructions

Adjustable Granularity and Structure

- Different granularity adis different comprehension levels
- More granularity variations give more edit points
- Low granularity: easy to skim, useful for large-scale edits
- High granularity: detailed understanding, contains edit points, useful for modification
- Medium granularity balances detail and efficiency
- Granularity should align with code units (block, function)
- Structured summaries align with program logic
- Structured summaries are easier to read
- Structured summaries support hierarchy and skim reading
- Structured summaries help locate edit points
- Structured summaries help with large-scale edits
- Paragraph summaries are cohesive and easy to read
- Paragraph is lengthy, jumbled together, and harder to parse
- Hierarchy and indentation keep summaries structured
- Summaries should be context-aware or customizable
- Content of different granularities needs careful design
- Order mismatches between summary and code structures

Interactive Mapping Between Summaries and Code

- Mapping helps understand code more effectively
- Mapping provides evidence for edits and builds trust
- Mapping works as auto-generated comments
- Mapping helps locate modification points
- Mapping reduces cognitive workload
- Clicking mappings supports code navigation on demand
- Mapping may be better for structural blocks than lines
- Visualizing mappings in summary input area is helpful
- Structure and mappings benefit non-English speakers
- Need mappings between code diffs and summary diffs

Auto-Updated New Summaries with Diffs

- Auto-updated summaries maintain workflow consistency
- New summaries with diffs help iterative modification
- Diffs support validation
- Low trust in diff quality
- Highlight colors may blend with mapping colors

Factors Impacting System Usability

- Familiar code requires only direct instructions
- Unfamiliar code increases reliance on summaries
- Imperative favors instructions, declarative favors summaries
- Code needing more control benefits from summaries
- Reliance on AI may hinder coding skill learning
- Scoping AI modifications improves control and testability

Future Improvements for NaturalEdit

- Support for multiple files
- Large projects need structured and cross-file connections
- Multi-file requires coarser mapping
- Code diffs should allow accept/reject options
- Support search and locating modification points
- Auto-summarize opened file without manual selection
- Jupyter Notebook support
- Code edit history management
- Alternative representations (dependency graph, data flow)
- Add debugging feature
- Include multimodal support, e.g., images
- Runtime sandbox to catch errors early