Codetations: Rich, Granular, and Maintainable Annotations for Evolving Code

Edward Misback
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA, USA
misback@cs.washington.edu

Eunice Jun
UCLA
Los Angeles, CA, USA
emjun@cs.ucla.edu

Erik Vank
Brown University
Providence, RI, USA
erik_vank@brown.edu

Zachary Tatlock
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA, USA
ztatlock@cs.washington.edu

Steven Tanimoto
Paul G. Allen School of Computer Science & Engineering
University of Washington
Seattle, WA, USA
tanimoto@cs.washington.edu

Abstract

Developers accumulate rich contextual knowledge about their code (e.g., design rationales, edge cases, worked examples) that does not fit in source files and goes stale when kept in separate documents or in human memory. We present Codetations, a VSCode extension enabling persistent, interactive annotations on arbitrary code spans without modifying the underlying file. A novel hybrid annotation anchoring method combines editor tracking for local edits with LLM-based semantic matching for offline changes, keeping annotations stable as code evolves. Annotation tools are modules sharing a minimal API for document access and state management. We demonstrate that this API is expressive enough to support multimedia embedding, automatic documentation maintenance, and external tool integration. In initial user studies with nine experienced developers, we found that attachment behavior matched their expectations. The diverse tools also enabled developers to imagine a variety of domain-specific use cases for annotations, particularly for validating and managing LLM-generated code. We discuss how a lightweight annotation layer can preserve contextual knowledge and open up a novel design space of developer-defined annotations.

CCS Concepts

• **Software and its engineering** → **Documentation**; **Integrated and visual development environments**; **Software evolution**; • **Information systems** → *Document representation.*

Keywords

document annotation, live and visual programming, documentation, software evolution



This work is licensed under a Creative Commons Attribution 4.0 International License.
UIST '26, Detroit, MI, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2856-3/2026/11
https://doi.org/10.1145/3830398.3830552

ACM Reference Format:
Edward Misback, Eunice Jun, Erik Vank, Zachary Tatlock, and Steven Tanimoto. 2026. Codetations: Rich, Granular, and Maintainable Annotations for Evolving Code. 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, 11 pages. https://doi.org/10.1145/3830398.3830552

1 Introduction

Software development involves managing far more information than exists in source files. Design decisions, implementation tradeoffs, edge cases, debugging workflows, and domain-specific constraints form a conceptual model that developers maintain mostly in their memory or in scattered external resources [11, 14]. Empirical studies have documented the consequences: developers spend significant time reconstructing context after interruptions, during code review, and when onboarding to unfamiliar codebases [4, 14].

The standard remedy is a mixture of inline comments and external documentation, but each forces developers to sacrifice one desirable property to obtain another. *Inline comments offer granularity without richness*: they can reference a specific expression or line, but they are limited to plain text. Developers routinely work around this constraint with ASCII art [11], computational notebooks [21], and other forms of literate programming [19], readily integrating diagrams, state tables, and execution traces into their explanations. *External documentation offers richness without granularity*: READMEs, wikis, and design documents can incorporate visuals alongside structured content, but they attach at the file or module level rather than to the specific operations, expressions, or cross-cutting slices where implementation decisions actually live [12, 37]. *Neither offers maintainability*: inline comments fall out of date as code evolves, and external documents drift further still, disconnected from the code they describe.

To address these needs, we present Codetations, a system for constructing rich annotations (e.g., text, visuals, or both) on user-defined spans of code, including read-only codebases. Codetations (i) implements a framework for anchoring annotations to arbitrary pieces of code and (ii) provides an API for constructing *annotation tools* to create annotations of various kinds. This paper contributes:

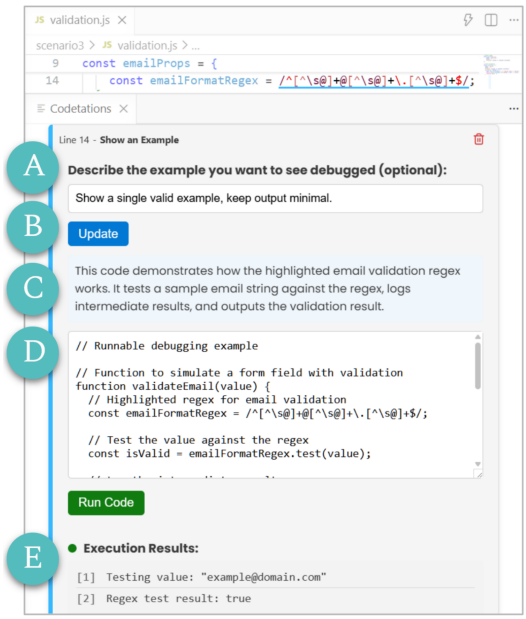


Figure 1: Codetations helps developers create and maintain rich, self-updating annotations on fine-grained, semantically-defined parts of any code file. Here, the behavior of (top) an underlined in-line regular expression is illustrated in (bottom) the *Show an Example* annotation tool. The developer directs an example code execution they want to see with A high-level debugging instructions, then clicks on B "Update" to receive C a summary of D LLM-generated testing code, and E live execution results. As the document evolves, the annotation stays anchored to the underlined text and remains interactive for one-click maintenance. The *Show an Example* tool was generated by an LLM using the Codetations API in just a few minutes.

(1) **Codetations**, that annotation system (Section 3);
(2) **A diverse library of annotation tools** spanning multimedia embedding, external tool integration, and automatic documentation maintenance, to demonstrate the Codetations API's expressivity (Section 4, Appendix B);
(3) **An evaluation of reattachment** accuracy, cost, and latency on a public benchmark (Section 5); and
(4) **Findings from a user study** with nine developers, who felt the system addresses a need for richer inline documentation and envisioned new domain-specific annotations (Section 6).

2 Related Work

Prior systems navigate tradeoffs across the richness, granularity, and maintainability of code annotations. Table 1 summarizes major anchoring strategies and their limitations.

Developers create context beyond source files through whiteboards, sketchbooks, and wikis, but these artifacts quickly lose their connection to the code, driving developers to mix notes and ASCII art into source files [7, 11, 35]. Computational notebooks [19, 21] and Babylonian programming [30] embed rich dynamic output beside code but require dedicated file formats or toolchains. Language-level annotation mechanisms (e.g., Java annotations, C# attributes, OCaml ppx) add metadata but are language-specific and stored in source rather than externalized. These approaches inspire the rich context Codetations supports while motivating its format- and language-independent design.

Table 1: Annotation anchoring systems. Rich = rich per-annotation state; Granular = attach to *any* text span; Ref = robust to offline renaming/refactoring; Exec = execute/sync with anchor text; NP = no parsing; NE = editor-independent.

System	Rich	Gran.	Maintain.		For all code	
			Ref	Exec	NP	NE
Code comments	✗	✗	✓	✗	✓	✓
Comp. notebooks [21]	✓	✗	✗	✓	✓	✗
Editor-tracked [17, 36]	✓	~	✗	✗	~	✗
AST/regex match [9]	✗	✗	✓	✓	✗	✓
String match [15, 31]	✓	✓	✗	✗	✓	✓
LLM retagging [26]	✓	✓	✓	✓	✓	✓

Keeping external annotations *anchored* (positioned correctly) as documents change is a long-standing challenge [6, 32]. Editor-tracked systems anchor annotations through in-source markers [36], structural element tracking [5, 8, 14, 17], or editor-maintained positions [10, 18, 33, 34, 36], but all break when code is modified outside their host environment. Text-similarity methods from Reiss [32], used by Catseye [15] and Sodalite [13], handle many offline edits but fail under refactoring. Syntax-based systems [3, 20, 23, 27, 28] (notably Lorgnette [9], whose custom projections are close in spirit to Codetations annotation tools) bind interactive tools to AST nodes or regex patterns. They offer rich visualizations but restrict annotations to supported languages and syntactic structures without capturing semantics or permitting per-instance state and user-selected spans. Modern IDEs layer non-modifying context over code (gutter icons, hover documentation, inline blame), but these are neither user-extensible nor persistent across sessions.

Unlike all of these prior systems, annotations in Codetations can survive in a shared evolving codebase by design and support significant per-instance annotation state (for example, a hosted color picker can remember which colors were previously picked for that instance), opening a novel tool design space (Section 4). This is achieved by extending the LLM-based semantic reattachment technique of Misback et al. [26] for offline edits, pairing it with standard offset tracking for local keystrokes. Where that prior work focused narrowly on anchoring, this paper explores combining reliable anchoring with a rich, extensible annotation tool API.

Retrieval-augmented generation [22] and protocols like MCP [29] connect LLMs to external context, but their agent-driven retrieval is transient and stochastic. Editor-level context specifications (e.g., Cursor's rules files [16]) operate at the file level and target LLM consumption rather than human readers. DocPrism [38] uses an LLM to detect code-documentation inconsistencies, considering maintenance issues that may be important to Codetations tool developers. Codetations takes a complementary, developer-driven approach:

persistent, inspectable annotations authored at arbitrary granularity that serve both human and future LLM contextual needs.

3 System

Codetations is a toolkit that consists of two parts: (i) an attachment module for semantically anchoring annotations to code and (ii) a minimal API for constructing tools that are guaranteed-by-construction to work on any file to which Codetations is attached.

3.1 Attachment module

Keeping document-external annotations in the right places over time is a hard problem, as Table 1 indicates. Codetations matches developer expectations with a two-tier strategy. For **online edits** (keystrokes in the editor), Codetations shifts anchor positions using offset tracking, matching the behavior of document editors like Microsoft Word. For **offline edits** (code updates from other users who aren't using Codetations, branch merges, or external tool changes), Codetations is the first annotation system we are aware of to use an LLM-based semantic reattachment method [26] for maintaining document-external markup.

To decide whether reattachment is needed, Codetations keeps a copy of each file's text alongside its annotations and compares this copy against the buffer contents whenever the extension is loaded; when they disagree, the affected annotations are marked detached and the user is notified.[1]

With the user's permission, Codetations re-anchors each detached annotation by prompting an LLM with (1) the prior file version—kept per annotation as a diff from a system-local copy of the file on disk—with the prior anchor text delimited by a special character, and (2) the updated file with line numbers prepended. Like Misback et al. [26], our prompt requests a JSON response containing the new anchor text, the new line number range, and a match index that disambiguates repeated occurrences of the anchor within a single line, but we also add model reasoning to this output to increase the system's transparency and improve model performance. Our full prompt appears in Appendix C.1. If a relevant new position cannot be found, the system currently emits a warning; prior work suggests possible improvements such as archiving these "orphaned" annotations [15]. At any time, users can also manually adjust the positions of annotations.

3.2 Annotation Tool API

In order to make a range of tools easy to construct and possible to host through the toolkit at the same time, tools must be modularized with a consistent API. We made each annotation tool a single React function component that accepts a single React props object and returns a React JSX interface. The props object provides four system capabilities to hosted tools:

(1) **Read/write the anchor text**, or the segment of the file the annotation is attached to.
(2) **Read/write per-annotation persistent state**, stored in a per-annotation record.
(3) **Read/write the full document**, for checking annotation context and maintaining consistency.
(4) **Access to the host editor's API**, for getting metadata or calling an LLM through VSCode's Language Model API [24].

Capability (1) is the most important. Because the system maintains the anchor attachment, an annotation tool developer can write their tool as a simple function of the anchor text. The tool developer does not need to use a language-specific parser or pattern matcher to locate and operate on the relevant code. If the semantic attachment system is thought of as maintaining markup [26], Codetations provides a flat document object model for rendering externally-marked entities from the source file as a list of annotations.

Anecdotally, the research team found that new tools were straightforward to create by hand given the minimal API above. Moreover, providing the API description and examples of existing tools to general-purpose LLMs was sufficient for the team to generate complex, working tools from short prompts in minutes (Appendix B.2).

3.3 Implementation

Codetations is a VSCode extension that lets a developer select any text span in a file, attach an annotation tool of a chosen type, and interact with that annotation in a panel adjacent to the code. Annotation data in our prototype system are stored in a JSON file within a hidden `.Codetations` directory at the repository root, separate from the source; multiple annotation lists could be kept for a file to distinguish personal and public annotations, though this is not yet implemented. Each record contains a unique ID, start and end positions, an annotation type name, and a data field for persistent state. Records are independent and can be updated in parallel. Besides the library of tools we discuss in Section 4, Codetations also loads custom annotation tools from a `.Codetations/components` directory at the repository root. Storing annotation data and tool definitions in the repository means they are versioned with the code. Although it is not yet clear whether git is the best storage mechanism for annotation data (see Section 7), annotations can easily be shared through git in the current implementation.

4 Example Annotation Tools

At this time, Codetations provides a library of ten annotation tools created by the research team with iteration based on user study feedback. We select three examples (also demonstrated in the video figure) to showcase the range of novel workflows enabled by the system's focus on richness, maintainability, and granularity. We describe additional annotation tools in Appendix B. Our claim of novelty does not involve individual tools but rather the system architecture that permits them all. Because it combines per-annotation state, hybrid semantic anchoring, and LLM access, Codetations is—to our knowledge—the first annotation system that can host all of these tools at once for many kinds of codebases. Even "classic" tools gain new capabilities in this setting. For example, a simple color picker can use state to support per-instance color naming and store user choice history (Appendix B.1).

4.1 Rich Multimedia Comments

Developers frequently sketch the structure of tricky code on whiteboards or in physical notebooks before and during implementation [7, 11]. In front-end development, it is also common to reason

[1] The video figure, 00:21–00:52, shows this flow for a file changed on disk while Codetations was closed.

about code behavior alongside UI screenshots or mockups. Traditional code comments cannot accommodate this kind of rich media, leaving developers who want to diagram, sketch, or visually annotate their code without a way to keep those artifacts connected to the specific lines they describe.

The *Markdown Note* annotation tool (Appendix Figure 2) addresses this gap by letting developers attach extended, multimedia-rich comments including whiteboard images and diagrams to any code span, regardless of programming language, without modifying the source file. The tool provides a Markdown editor that supports formatted text and embedded images within a single annotation panel. Because notes are stored in the annotation record rather than inline, they do not compete with code for readability in the buffer, and because they are anchored to user-selected spans, they remain associated with the specific code they describe as the file evolves.

4.2 Maintaining Dynamic Examples and Tests

Prior work by Aghajani et al. finds that users struggle significantly with generating and maintaining synchronization of documentation with code [2]. The two tools below support developers through two kinds of syncing: generating and updating line-level natural-language documentation to maintain its relevance, and testing and updating line-level code to satisfy a documented natural-language user specification. In both tools, synchronization is maintained with the click of a button, and could be further automated while maintaining options for user curation.

The *Show an Example* tool (Figure 1) lets a user attach an annotation to any code span and provide natural-language instructions describing example data they would like to see the code executed on (e.g., "Test a valid email address"). An LLM then generates executable code from the anchor text, file context, and user instructions. Execution results are displayed beneath, similar to a Jupyter notebook [21]. If the file changes, the user can regenerate the test while preserving existing instrumentation. The key novel interaction is *testing at arbitrary granularity*: a developer can select a single expression without constructing a harness for the enclosing scope.

The *LM Unit Test* tool (Appendix Figure 3) takes a complementary approach. Rather than illustrating behavior, it lets a user pose a yes-or-no question about a code span (e.g., "Is this code well-commented?" or "Is this method free of side effects?"). The question is evaluated by an LLM on demand. Additionally, a "no" answer triggers a suggested code modification that can be applied to the buffer with one click. Because the question persists with the annotation, it can be re-evaluated after any edit, functioning as a lightweight validator that keeps the document in sync with the documentation.

4.3 Connecting External Domain-Specific Tools

In some domains, individual code snippets of various granularities can require extensive analysis to ensure correctness and performance. Floating-point arithmetic is a canonical example: a single expression may accumulate significant error, and tools like Odyssey [25], a web-based workbench for exploring and optimizing floating-point expressions, exist to help developers identify and correct such issues. However, these tools are disconnected from the codebase. Developers must move between their code and these tools. As a result, analysis often leads to optimized code that obscures the original intent, leaving future developers with questions: What was the original expression? Was a validated process followed? How does the optimization compare?

Codetations addresses these gaps by providing a stable, granular attachment point for external analyses and a place to persist the resulting state. Our *Analyze Floating-point Expression* annotation tool (Appendix Figure 4) demonstrates this pattern. Given a Python expression such as `math.sqrt(x + 1) - math.sqrt(x)`, the tool uses an LLM through the Codetations API to translate the expression into Odyssey's MathJS input format, producing `sqrt(x + 1) - sqrt(x)`. The user can then open Odyssey with this translated expression pre-loaded, explore accuracy-performance tradeoffs, and save the resulting analysis session back into the annotation state. Because the LLM handles translation, this integration works immediately for any input language the model understands, without dedicated parsers or transpilers.

The annotation thus serves as a persistent record of the full provenance of an optimized expression: the original code, the translated form, the analysis results, and the rationale for the chosen optimization. If the implementation later changes, the same annotation is immediately available to validate new choices.

More broadly, this tool illustrates a reusable pattern: any external tool with a well-defined input representation can be wrapped as a Codetations annotation, using the system as a lightweight glue layer that handles attachment to a granular part of the code and translation from any language into the tool's expected format.

5 Technical Evaluation

We evaluated the method of Section 3.1 on the benchmark provided by Misback et al. [26]. The benchmark consists of 90 synthetic program edits across five programming languages, each paired with a human-validated tracked selection target. We show an example benchmark in Appendix C.2.

Accuracy. We ran the benchmark with Qwen3.6-27B-FP8, a small open model (April 2026) that runs on consumer GPUs with quantization. After tuning the prompt on the benchmark's 9 training examples, the model's first attempt matched the target exactly in 86 of 90 cases. For the remaining four cases (discussed in Appendix C.3), the model made a reasonable choice among multiple possible new attachment points. On the whole, semantic reattachment survives edits that defeat the prior anchoring strategies shown in Table 1: for example, we found that even when a function like `function quickSort(arr, low = 0, high = arr.length - 1)` has been refactored into a class method like `class Sorter { execute(data, start = 0, end = data.length - 1) ...` located in a different part of the file and sharing no identifier names, the model easily reattaches annotations to it.

Cost and latency. Reattachment cost scales with the sizes of the old and new file versions and the model's reasoning length. At current OpenRouter prices for Qwen3.6-27B ($0.30/M input, $3/M output tokens) [1], one reattachment call costs roughly $0.001, and the full 90-example suite costs $0.11. Average generation latency is 4.4s per call, and calls can be parallelized.

6 Initial User Studies

We conducted two rounds of think-aloud studies with a total of nine participants, who had from five to 30 years of programming experience in web development, scientific computing, programming languages research, and database programming.

Each study session lasted 60 to 90 minutes and included three parts: (1) a pre-demo discussion focused around a short written survey, (2) a hands-on session with Codetations, and (3) a post-demo discussion focused around a second written survey. In the hands-on session, participants installed the extension on their own machines and worked in an example codebase with five code artifacts (detailed in Appendix A.1). The codebase also contained two pre-existing annotations, letting us check comprehension in a walk-up-and-use scenario. Participants attached five annotations in response to verbally-stated tasks, such as "add a *Show an Example* annotation to understand line 14"; the full task list appears in Appendix A.1. Other than guidance for the external floating-point analysis tool (Section 4.3), all participants completed these tasks unaided and used the resulting annotation interfaces to draw conclusions about the code. Finally, each participant was guided through a scenario in which a file was edited on disk while Codetations was closed, so that reloading the extension triggered semantic reattachment (protocol in Appendix A.1). After the first five participants, we iterated on the designs of several tools in the demonstration and added quantitative measures to our pre- and post-demo surveys.

We had the following research questions:

RQ1. Do the richness (diverse annotation tools), granularity (code span-level attachment), and maintainability (automatically generated and code-synced annotation content) of annotations in Codetations offer better workflows to developers than familiar documentation methods like code comments? Do users have any design concerns?

RQ2. How do developers from different domains react to the annotation tools Codetations makes possible? What new kinds of annotations or tools do they envision after using the system?

We collected three sources of data throughout the study sessions: close observations of participants' interactions with the system, audio transcripts, and detailed researcher notes. We conducted a thematic analysis, detailed in Appendix A.2. We report key findings across both user studies below.

6.1 Findings

6.1.1 RQ1: Annotation tools in Codetations don't need to be complex to address "pain points" in established workflows. Participants consistently identified the separation of annotations from source code as resolving fundamental tensions between richness and granularity. P5 noted that separate annotations let them "write down all these ideas and just have them around, but they're not imposing on the actual code," (enabling desired richness) while P3 observed that in-code comments create a scaling problem: "if everybody adds their comment to the code, then the code blows up with a huge amount of comments." Granular attachment to arbitrary spans addressed a gap that traditional comments leave open; P3 noted that in conventional comments "it's hard to point to other parts of the code in an easy way." For maintainability, P3 valued semantic reattachment directly: "if a function is moved from one file to the other, then I would fail in trying to go through the old documentation I have. But I think for this one, if there is retag [semantic reattachment], the higher chance I will be able to do that." However, participants were clear that lowering the *representation* barrier does not eliminate the *initiation* barrier: P2 stated plainly, "the hardest part is just doing it [choosing to document code]." P6, the most experienced participant (30 years), raised an important design concern: the current tools skew toward low-level line comments rather than method- or module-level documentation, which is "what I struggle to keep updated." We attribute this response to our demonstration's focus on novel fine-grained attachments, which foregrounded differences from prior systems; tool designers can immediately serve users like P6 by targeting the coarser, more familiar granularities where documentation debt currently accumulates.

6.1.2 RQ2: Developers found our demonstrated tools useful for workflows in their respective domains; they envisioned future tools reshaping coding conventions and managing LLM-generated code. Participants responded to the four highlighted tools with concrete use cases grounded in their own workflows. For *image annotations* (Section 4.1), P2 wanted to attach rendered equations to numerical code: "if there's a function that is doing some sort of math for me… you could just have a picture of the equation there," and P7 wanted to embed algorithm diagrams from research papers next to implementations. For the *floating-point analyzer* (Section 4.3), P6 recognized that it "gives me a framework for evaluating the [optimization] alternatives, which is great," valuing structured provenance over a raw list of options. For *Show an Example* (Section 4.2), P9 valued testing at arbitrary granularity: "I didn't have to construct a whole input-output pair for some larger function. I could just be like, I want to test this line." For the *LM Unit Test* (also Section 4.2), P8 immediately connected it to continuous integration: "now if someone changes this code, like, refactors it, but forgets to change the comments… then this test will fail in CI."

Beyond reacting to existing tools, participants envisioned novel uses. P8 wanted interactive data structure explorers: "if every single cargo test I had that involved an egraph had an [annotation tool] that let me explore the egraph and interact with it… that would be pretty cool." P9 speculated that sub-function testing could reshape coding conventions: "if function boundaries weren't the only level of granularity I could test… maybe I don't want that function anymore." Looking further ahead, P9 envisioned annotations as a bridge to AI-generated code: "here's a pile of code. And then here's some evidence that there's a person who has thought about it," imagining pseudocode persistently linked to generated implementations.

7 Discussion

Codetations opens a design space of rich, granular, and maintainable code annotations with capabilities no prior system supports simultaneously. Our participants were not only able to use the system but also envisioned using it to author annotations at a range of granularities, from single-line expressions to functions to entire files or even collections of lines across different files. Because existing tools do not support this level of customization, developers currently apply ad hoc solutions. For example, P9 mentioned decomposing functions into sub-functions in order to test and document them. This suggests that when developers have to choose between

rich annotations and code coherence, they prioritize rich, granular annotations. Therefore, we see CODETATIONS as a starting point for new kinds of annotations and investigations of developer practices.

Furthermore, participants identified the maintainability of annotations as a major benefit: because hybrid attachment keeps tool output synchronized with the latest code (Table 1), annotations can act as inspectable records of human intent and validation across a changing codebase. Participants imagined the system filling this role in their own domains (Section 6.1.2).

That new tools can be produced by an LLM in minutes from short prompts suggests that the design space is highly accessible. Developers need not wait for tool authors; they can create annotations tailored to their codebase, domain, or workflow on the fly. LLM chat interfaces are currently popular as a way to rapidly generate code, but the role of LLMs in more mature codebases is arguably still unclear. CODETATIONS shows how LLMs may enable the behavior and evolution of complex artifacts like code to be annotated and monitored over time, permitting developers to reliably attach rich interfaces to semantic objects that were previously impossible to address. These new capabilities reframe code annotations from a fixed taxonomy of comments and documentation into a developer-defined medium that is both separate and inseparable from code.

8 Limitations and Future Work

A few limitations warrant investigation. LLM-based reanchoring occasionally failed due to the use of an older LLM early in our study, particularly on regular expressions where models struggled to reproduce exact syntax. Participants manually reattached annotations in these cases, confirming graceful degradation but highlighting dependence on model capability. Newer models are substantially stronger and cheaper (Section 5), but that benchmark is synthetic; evaluating reattachment on real-world edit histories remains important future work. Our prototype stores annotations in a repository-local JSON file, risking merge conflicts under concurrent editing. Future work should explore a local-first storage model with optional selective sharing to better support collaboration. Participants also requested extending our system to support cross-file annotations, document-wide metadata, and tighter version-control integration.

In our study, nine developers used pre-implemented annotation tools on an example codebase we provided rather than on their own code. Controlled studies are still needed to assess the usability of the API in creating new annotation tools, and future evaluations, such as deployments on developers' own codebases, could observe long-term usage patterns and let researchers study codebase-specific needs—including UIs for rapid tool generation and intelligent context management.

Acknowledgments

This material is based on work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number DE-SC0022081. This work was also supported by the National Science Foundation under Grant No. 2346395. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] 2026. *Qwen3.6 27B - API Pricing & Benchmarks.* https://web.archive.org/web/20260720225532/https://openrouter.ai/qwen/qwen3.6-27b?endpoint=450be8a0-0ccb-4a62-844f-e83040dfc9cc

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1199–1210.

[3] Tom Beckmann. 2026. *Composing Domain-specific Notations in Programming Systems.* Ph.D. dissertation. Hasso Plattner Institute, Potsdam University, Potsdam, Germany.

[4] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (2022), 35–57.

[5] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2503–2512.

[6] AJ Bernheim Brush, David Bargeron, Anoop Gupta, and Jonathan J Cadiz. 2001. Robust annotation positioning in digital documents. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 285–292.

[7] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J Ko. 2007. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 557–566.

[8] Robert DeLine and Kael Rowan. 2010. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 207–210.

[9] Camille Gobert and Michel Beaudouin-Lafon. 2023. Lorgnette: Creating Malleable Code Projections. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–16.

[10] Björn Hartmann, Mark Dhillon, and Matthew K Chan. 2011. HyperSource: bridging the gap between source and code-related web sites. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2207–2210.

[11] Devamardeep Hayatpur, Brian Hempel, Kathy Chen, William Duan, Philip Guo, and Haijun Xia. 2024. Taking ascii drawings seriously: How programmers diagram code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–16.

[12] Tom-Michael Hesse, Arthur Kuehlwein, Barbara Paech, Tobias Roehm, and Bernd Bruegge. 2015. Documenting Implementation Decisions with Code Annotations. In *SEKE*. 152–157.

[13] Amber Horvath, Andrew Macvean, and Brad A Myers. 2023. Support for Long-Form Documentation Authoring and Maintenance. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 109–114.

[14] Amber Horvath, Andrew Macvean, and Brad A Myers. 2024. Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–17.

[15] Amber Horvath, Brad Myers, Andrew Macvean, and Imtiaz Rahman. 2022. Using Annotations for Sensemaking About Code. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–16.

[16] Anysphere Inc. 2025. Cursor – rules. https://docs.cursor.com/context/rules

[17] Mik Kersten and Gail C Murphy. 2005. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th international conference on Aspect-oriented software development*. 159–168.

[18] Mary Beth Kery, Amber Horvath, and Brad A Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists.. In *CHI*, Vol. 10. 3025453–3025626.

[19] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[20] Amy J Ko and Brad A Myers. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. 387–396.

[21] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. 2020. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–11.

[22] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[23] Andrew McNutt and Ravi Chugh. 2023. Projectional editors for json-based dsls. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 60–70.

[24] Microsoft. 2024. Visual Studio Code Language Model API. First released in the July 2024 (version 1.92) release of Visual Studio Code. https://code.visualstudio.com/api/extension-guides/language-model

[25] Edward Misback, Caleb C Chan, Brett Saiki, Eunice Jun, Zachary Tatlock, and Pavel Panchekha. 2023. Odyssey: An interactive workbench for expert-driven floating-point expression rewriting. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–15.
[26] Edward Misback, Zachary Tatlock, and Steven L. Tanimoto. 2024. Magic Markup: Maintaining Document-External Markup with an LLM. *ArXiv* abs/2403.03481 (2024). https://api.semanticscholar.org/CorpusID:268253244
[27] Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling typed holes with live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 511–525.
[28] Cyrus Omar, Young Seok Yoon, Thomas D LaToza, and Brad A Myers. 2012. Active code completion. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 859–869.
[29] Anthropic PBC. 2026. What is the Model Context Protocol (MCP)? https://modelcontextprotocol.io/docs/getting-started/intro
[30] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *arXiv preprint arXiv:1902.00549* (2019).
[31] Steven P. Reiss. 1990. Connecting tools using message passing in the Field environment. *IEEE Software* 7, 4 (1990), 57–66.
[32] Steven P Reiss. 2008. Tracking source locations. In *Proceedings of the 30th international conference on Software engineering*. 11–20.
[33] Roy Rutishauser, André A. Meyer, Reid Holmes, and Thomas Fritz. 2023. Semi-Automatic, Inline and Collaborative Web Page Code Curations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1866–1877.
[34] Nicholas Sawadsky, Gail C Murphy, and Rahul Jiresal. 2013. Reverb: Recommending code-related web pages. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 812–821.
[35] Margaret-Anne Storey, Jody Ryall, R Ian Bull, Del Myers, and Janice Singer. 2008. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th international conference on Software engineering*. 251–260.
[36] Margaret-Anne Storey, Jody Ryall, Janice Singer, Del Myers, Li-Te Cheng, and Michael Muller. 2009. How software developers use tagging to support reminding and refinding. *IEEE Transactions on software engineering* 35, 4 (2009), 470–483.
[37] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.
[38] Xiaomeng Xu, Zahin Wahab, Reid Holmes, and Caroline Lemieux. 2025. DocPrism: Local Categorization and External Filtering to Identify Relevant Code-Documentation Inconsistencies. *arXiv preprint arXiv:2511.00215* (2025).

A User study supplemental information

Table 2: Participant backgrounds for our user study.

ID	Round	Background	Prog. Exp.
P1	1	Undergrad	5 years
P2	1	Senior Research Specialist, chemE	10 years
P3	1	Postdoc, proof engineering	18 years
P4	1	MBA, SQL	25 years
P5	1	Undergrad RA, DSL/IRs	7 years
P6	2	Senior Lecturer, formal methods	30 years
P7	2	PhD student, PL/SE; compBio	12 years
P8	2	PhD student, compilers; game dev.	9 years
P9	2	PhD student, compilers; maps, cloud	10 years

The professional and academic backgrounds of participants in our study are shown in Table 1.

In round 2 of our study, we asked participants the following quantitative questions before our demo to begin our discussion:

Question: On a scale of 1-7, in your experience, how difficult is it in practice to write documentation while programming?

Question: On a scale of 1-7, in your experience, how difficult is it to keep documentation synchronized with code changes over time?

Table 3: Quantitative responses from round 2 of our user study. (*P7 works mainly with computational notebooks and reported less trouble with maintaining documentation.)

ID	Difficulty of Doc. (1-7)	Difficulty of Doc. Sync. (1-7)	Likelihood of Using Our Tool (1-7)
P6	6	7	5
P7	2*	4	6
P8	6	5	7
P9	6	7	6

We asked participants the following quantitative question after our demo to wrap up our discussion:

Question: On a scale of 1-7, if a polished version were available through your IDE, how likely would you be to use this tool?

Participant responses are shown in Table 2.

A.1 Study Materials

Example codebase. Table 4 lists the artifacts in the example codebase participants worked with. All examples were synthesized for the purposes of the study.

Table 4: Artifacts in the example codebase used in the hands-on session.

Artifact	Language	Role in study
Payment API documentation	HTML	Task 1
Quicksort (classroom-style)	JS	Task 2
Form-validation code	React/JS	Task 3
Financial modeling computation	Python	Task 4
Simple for-loop	JS	Task 5

Annotation tasks. Participants completed five verbally-stated annotation-adding tasks; Table 5 reproduces the task wordings as given.

Guided reattachment scenario. For the reattachment scenario, at the end of Task 4, we asked participants to first disable VSCode's auto-save feature, then make a few changes of their choice to the buffer and close the editor without saving. Upon reloading the file and reopening Codetations, they were asked to observe a dialog indicating that annotations needed to be reattached and asked to click a button to fix it. Several participants commented that this process should run in the background.

A.2 Study Themes and Representative Quotes

We open-coded the transcripts from our interviews, iteratively refined and grouped the codes following a grounded-theory approach, and then conducted a thematic analysis across all three sources. The resulting themes fell into three families: user-identified needs (e.g., documentation fragmentation), responses to Codetations (e.g., the value of non-intrusive annotations), and future visions (e.g., annotations that host AI assistants); Appendix A.2 lists the themes with representative quotes. We compared results from both

Table 5: Verbally-stated annotation tasks from the study.

#	Task (as stated)
1.1	"Observe the annotation shown to the right and describe what you think it means."
1.2	"See the extension README, then add a Comment annotation to part of the file."
1.3	"Add a Display HTML annotation to another part of the file."
2	"Observe the annotation shown to the right [a GIF showing quicksort] and describe its purpose."
3	"Add a Show an Example annotation to understand line 14."
4	"Add an Analyze Floating-point Expression annotation to the expression on line 19." [Followed by a guided exercise to obtain a better expression with the external interface.]
5	"Add an LM Unit Test annotation to the for-loop to ensure that the code is well-commented." [It is not, so a correction is suggested.]

rounds of user evaluation and found that they were consistent with one another. We show these themes and representative quotes in Table 6.

Table 6: Theme families from our thematic analysis (Section 6), with representative quotes.

Theme	Representative quote
User-identified needs	
Documentation fragmentation	"If you're looking at a piece of code and have a question about it, it's hard to know whether there exists a doc that would give some useful information." (P9)
Documentation generation & maintenance	"I teach software engineering, and I struggle with this... Surely I'll do the documentation afterward... Spoiler: I won't." (P6)
Media limitations	"I like to use a notebook [a physical notebook, rather than code comments], because I can draw things out other than words" (P5)
Responses to Codetations	
Non-intrusive annotations	"This is different from changing the code, because if everybody adds their comment to the code, then the code blows up with a huge amount of comments." (P3)
Robust code tracking	"If a function is moved from one file to the other, then I would fail in trying to go through the old documentation I have. But I think for this one, if there is retag, the higher chance I will be able to do that." (P3)
Different user roles	"[A system like Codetations] would help with the QA process." (P4)
Interactive testing	"I think the test generation is like the thing that for me, I would be most likely to want model generated code for." (P9)
Future visions	
AI and annotation integration	"Maybe people will write pseudocode and then that will be linked in some way to the code implementation of the pseudocode." (P9)
Domain knowledge representation	"I think that as programming becomes more connected to 'real world' tasks outside of pieces of code—for example by scientists, social scientists, businesses—it becomes more and more important to include the contextual information and domain knowledge related to programs with the programs themselves." (P7)

B Annotation tool library

B.1 Implemented tools

Below, we summarize all annotation tools currently included with Codetations that are not described in detail in Section 4.

Color Picker. A lightweight color picker that lets the user select a color and save a descriptive name next to it. This tool demonstrates that even minimal, non-LLM annotation tools are easy to build with the API, and it goes beyond previous syntax-attached color pickers with its stateful per-instance color naming. We also considered adding a per-instance history of previously chosen colors.

Comment. A simple annotation for adding a threaded comment to a code span, analogous to margin comments in document editors. This annotation goes beyond traditional code comments by permitting an extended discussion to be kept with code. Our current implementation is a mockup; a production version would need to record commenter identity and timestamps, and real-time comment updates from multiple users would require the annotation data to be stored in a backend that supports live synchronization rather than a static JSON file.

Image Upload. Lets the user attach an image or animated GIF to a code span. This again goes beyond traditional code comments by permitting a significant amount of explanatory data to be attached to particular lines of code without inserting fragile links or raw data into the file. The image data is Base64-encoded and stored directly in the annotation's data field. While convenient for small assets, this approach increases the size of the annotation file; future systems could offload binary data to a database or object store.

Display HTML. Attempts to render the anchor text as HTML in the annotation panel, providing a live preview. This is useful for files containing markup, such as HTML templates or SVG fragments. Because the tool has per-annotation state, future versions could capture and persist a reader's interactions with the rendered content to check particular behaviors.

Explain in English. An experimental tool that calls an LLM to produce a high-level, plain-language summary of the code it is attached to. Because it is persistent, the summary can be regenerated to update it after edits. We envision this as a lightweight alternative to inline comments for complex logic, though further evaluation is needed to understand when LLM-generated summaries are preferred over hand-written ones.

Generator. The Generator tool is intended to let end-users create new annotation tools without leaving the editor: the user describes

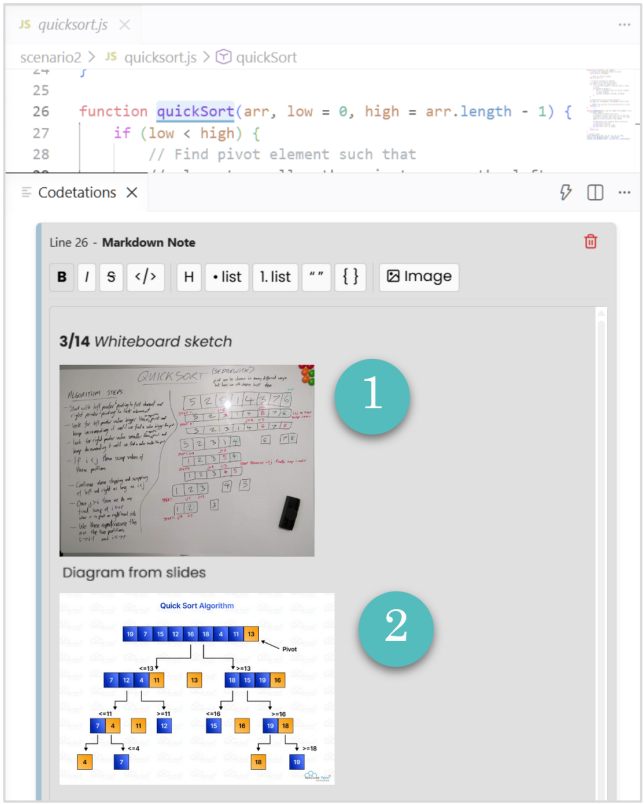


Figure 2: Developers can use Codetations to persistently attach multimedia to code segments more robustly than prior systems. Above, a function has been annotated with a Markdown Note annotation tool (below). This prototype tool, produced by an LLM in a few minutes, permits the developer to describe the function with a mixture of rich text and pasted images such as 1 whiteboard sketches and 2 diagrams without cluttering the source file.

the desired behavior in natural language, and the tool produces a working React component using the Codetations API. This tool works for simple demonstrations and is still under active development.

B.2 Generating tools with an LLM

B.2.1 Markdown Note. Here we show prompts used to rapidly generate the Markdown Note annotation tool.

The prompts were given directly to Claude Opus 4.6 via the claude.ai website, along with the code for instantiating 5 previously-existing annotation types.

1: "Add a WYSIWYG markdown note annotation type that permits images to be uploaded or pasted from the clipboard and displayed inline. You are permitted to use a library if needed."

This produced a working component. However, the component rendered markdown with a live adjacent preview rather than a WYSIWYG editor.

2: "I don't want a live preview, I want WYSIWYG editing."

This produced a working component with WYSIWYG editing.

3: "Make the editor viewport height user-adjustable, preserving the value for the height as annotation state."

This produced a working component with adjustable height.

B.2.2 Show an Example. Here we show prompts used to rapidly generate the tool shown in Figure 1.

The prompts were given directly to Claude Sonnet 3.7 via GitHub Copilot Chat in VSCode, along with the code for instantiating 5 previously-existing annotation types.

1: "Add a Show Debugged Example annotation type that attaches to javascript code and asks a language model to produce code that runs an example execution of the selected region (with any missing variables set to reasonable defaults) with debugging instrumentation. The produced code should be held in a cached user-editable field and should execute (for now just using the webview interpreter) and print results visibly for inspection when the user clicks a button. In order to make sure the produced code uses reasonable defaults you should use props.value.document to inform the language model about code context (the version of the code cached in the annotation), as in LMUnitTest."

This produced a working component with all of the fields shown in Figure 1 except for the natural language end-user-facing customization field at the top of the component. We note reusable information about the Codetations API (e.g. the use of `props.value.document`) could be passed to the LLM as part of a system prompt.

2: "Update this component to include a field that allows a user to adjust what is debugged with a natural language prompt."

This produced a working component with all of the fields shown in Figure 1.

3: "When regenerating, the debugging code may already be populated. Any existing debugging instrumentation and any examples used should be preserved if possible."

This produced a working component with the underlying generation prompt updated to include previous context.

C System supplemental information

C.1 Semantic Reattachment Prompt

The full reattachment prompt referenced in Section 3 is reproduced below. The old file version ($INPUT_FILE below) includes the prior anchor text delimited by a special character; the new file version ($UPDATED_FILE) has line numbers prepended. The model returns a JSON object containing its reasoning, the new anchor text, the new line number range, and a match index.

C.1.1 Full prompt. Consider the following file:

<INPUT> $INPUT_FILE </INPUT>

A specific segment of code has been marked with "$DELIMITER". The segment refers to ONLY THE TEXT BETWEEN THE "$DELIMITER" marks:

 $SEGMENT

Next, consider the following updated file:

<UPDATED> $UPDATED_FILE </UPDATED>

You are responsible for placing an identical annotation on this updated file. It is extremely important that you place the annotation in the correct place. Important metadata is attached to this segment.

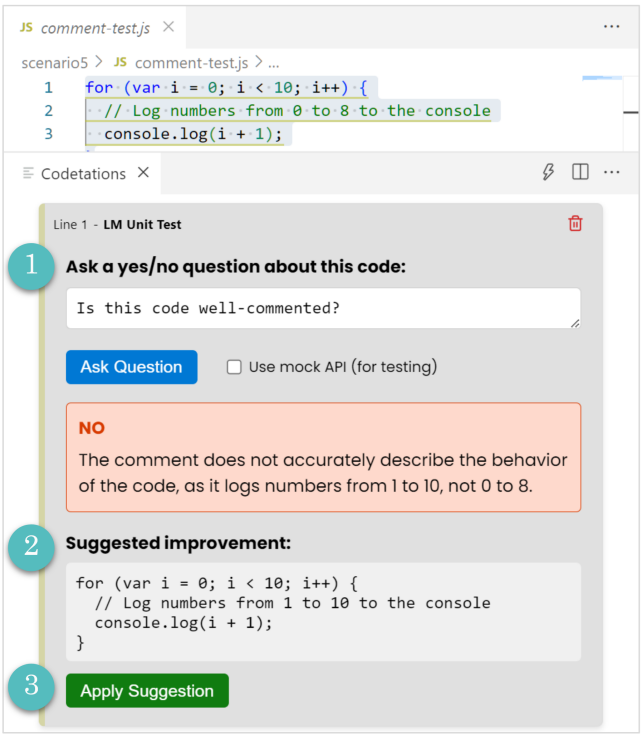


Figure 3: The LM Unit Test annotation type lets users 1 ask yes-or-no questions about the code and 2 suggests changes if the answer is "no." 3 Suggestions can be applied to the buffer. The question stays with the code and can be asked again after the code is updated. This annotation type was generated in a few minutes by an LLM (see Appendix B.2.2 for an example).

Describe possible sections the specific segment could be said to be located in. It is possible the segment has not changed, or that it has been refactored. Pick the most correct choice. Remember to be detailed about the start and stop of the segment. If the segment has been updated, it may need to expand or shrink. BE CAREFUL TO INCLUDE NOTHING EXTRA. Then, provide the following numbered answers as a JSON object:

0) Print your brief notes on choosing the exact segment in the updated file. Pay extra attention to ensuring that the "type" of the annotation, including the beginning and ending characters forming the boundaries, is as preserved as much as possible.

1) Print ONLY the text of the updated specific segment. You must print all of the text here.

2) State ONLY the line number in UPDATED that (1) starts on.

3) State ONLY the line number in UPDATED that (1) ends on.

4) (1) may occur multiple times in the section given by [(2),(3)]. Which number occurrence, as ONLY a 1-indexed number, is (1)?

The object must look like: {1: <code>, 2: <number>, 3: <number>, 4: <number>}

The answer to 1 should be a code string only, without markdown formatting or extra notes.

C.2 Example Benchmark Case

Here we show training example 1 from the benchmark, a simple Python grocery price calculation script. The initial state, with the selection identified by <START> and <END> tokens, looks like this (ellipses added to hide unrelated code):

```
const cart = [];
...
function calculateTotalPrice() {
  let totalPrice = 0;
  for (let item of cart) {
    totalPrice += <START>item.price<END>;
  }
  return totalPrice;
}
...
```

The final state, with a "correct" selection marked:

```
const cart = [];
...
function calculateTotalPrice() {
  let totalPrice = 0;
  for (let item of cart) {
    // Updated to multiply by the quantity
   totalPrice += <START>item.price * item.quantity<END>;
  }
  return totalPrice;
}
...
```

Given the initial state and an unmarked final state, the system is challenged to place the start and end tokens "correctly" (matching a human programmer's expectation). We note that this example happens to be ambiguous—it could also be correct for only `item.price` to be selected in the final version. We discuss ambiguous cases further in Appendix C.3.

C.3 Ambiguous Benchmark Cases

Although they describe "ambiguously correct" responses, Misback et al. do not distinguish between kinds of ambiguities [26], so we include a discussion of the issue here. In four of the 90 benchmark edits (Section 5), the model's reattachment did not exactly match the recorded human-validated target, but the chosen span is as defensible as the target. For each case we give the language, the edit, the target selection, the model's selection, and why both are defensible.

(1) Benchmark index 4, Python. Previously, a line with an `input` statement was selected. In the edit, the statement is refactored into a loop that handles input errors by retrying input collection (basically converting `foo = float(input())` to `while True { try { foo = float(input()); break } except ValueError: continue }`). The benchmark target selects only the line with the input statement inside the loop; the model selects the full loop. The loop essentially functions as a safer input assignment, so without further context, the model's choice is reasonable.

(2) Benchmark index 25, JavaScript. Previously, a hardcoded value used in an integer comparison was selected. The edit

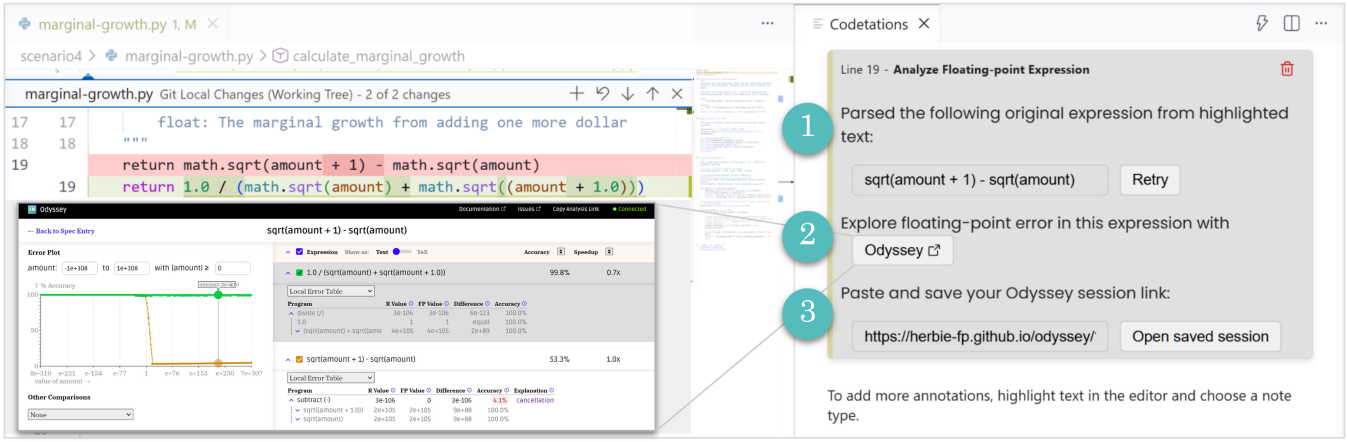


Figure 4: Codetations connects a floating-point mathematical expression, shown in naïve and optimized forms by a source code diff (top left), to the web-hosted Odyssey analysis tool (bottom left) as follows: an Analyze Floating-point Expression annotation tool (right) uses an LLM through the Codetations API to 1 translate the Python expression into a general form, 2 lets the developer open Odyssey with that translated form as input, and 3 lets the developer save the resulting analysis session state. Later developers can reuse this annotation to quickly understand the full provenance and performance of the optimized code. This tool immediately brings Odyssey's analyses to in-line expressions in any language understood by the LLM.

is a refactoring that introduces a named parameter for this value. The benchmark target selects the value on the RHS of the parameter assignment; the model selects the parameter where it is now used in the comparison. Without further context, either choice may be reasonable.

(3) Benchmark index 63, JSX. Previously, an array named "friends" was selected as follows: "const <START>friends = [...]<END>". The edit moves the values from this array into an assignment to an object named "data" in a new function. The benchmark target selects as follows: "const data = <START>[...]<END>"; however, the model expands to "<START>const data = [...]<END>". Without more context, this expansion to include the "const" keyword seems benign.

(4) Benchmark index 79, Racket. Previously, two consecutive expression closing parentheses were selected as follows "(outer ... (if (x?) (y<START>))<END>)". (This example resulted from a code synthesis error in the benchmark generator, but was retained since even strange selections should be reattached as well as possible.) A partial edit has transformed this into "(outer ... (if (x?) (loop (a) (b))))". The target selects "(outer ... (if (x?) (loop (a) (b))<START>))<END>"; the model selects "(outer ... (if (x?) (loop (a) (b<START>))<END>))". After some consideration, we think *neither* of these is exactly correct (we would claim "(outer ... (if (x?) (loop (a) (b)<START>))<END>)" is probably most accurate), but we think it's mostly important that, even in this degenerate case, the model was still able to place the tag within a character of the correct position using our system.

Together, the above cases indicate a few areas for improvement in future work. First, an attachment system may take into account that possible attachment points can multiply or vanish, and tentatively attach annotations to multiple places or archive them in these cases, pending user review. Second, for ambiguous points, context from the annotation content or user directions might be used to disambiguate the attachment point. Third, definition of behavior for degenerate cases, such as selections that violate normal expression boundaries or repeated characters, may help the system remain robust to unexpected uses.