Code in Context: Keeping Developer Context and Interfaces on Evolving Software

Edward L. Misback

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Steven L. Tanimoto, Chair

Zachary Tatlock, Chair

René Just

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Code in Context:

Keeping Developer Context and Interfaces on Evolving Software

Edward L. Misback

Chairs of the Supervisory Committee:
Zachary Tatlock Steven L. Tanimoto
Paul G. Allen School of Computer Science & Engineering

Despite progress in programming tools and environments, developers still struggle to maintain mental models of the systems they build and the context surrounding their code. This dissertation advances a framework for enhancing program understanding and manipulation for all agents working with a codebase—including developers and automated agents such as large language models (LLMs)—through notes on shared context and high-level representations that are attached to code and synchronized with changes as it evolves, e.g., external notes and UIs that stay linked to sections of a program where they are meaningful, with their content updated to remain relevant.

This dissertation first explores the design of high-level programming tools, addressing the specific case of floating-point error analysis with a custom tool

developed during the PhD process called Odyssey. Odyssey is a workbench for floating-point analysis that transforms an existing low-level, black-box expression improvement tool (the Herbie floating-point expression rewriting tool[PSSWT15]) to support a high-level, scaffolded analysis process where users can analyze, generate, and iterate on automated suggestions using their own expertise.

Next, considering how contextual information like that from an Odyssey analysis session might be attached to actual floating-point programs, I explore maintaining semantic connections between document sections and metadata using a new technique named Magic Markup. This system uses an LLM to "magically" maintain the positions of external notes on an evolving document without write access through semantic anchoring rather than brittle syntactic approaches.

Finally, I formalize and implement this approach in Codetations, a VS-Code extension that helps developers contextualize documents with rich, interactive notes and tools. Codetations builds on the Magic Markup method to keep notes outside the document while integrating edit-tracking and exposing a rich API for annotations to respond to buffer changes and leverage editor features. In a qualitative evaluation, developers recognize this method as enabling more extensive and useful forms of documentation, and worked examples show the additional context improving code repair performance in LLMs.

Together, these systems indicate a practical path by which existing pro-

gramming systems can begin to offer powerful, developer-customized tools that meet developers' needs for contextual information and high-level analysis.

Contents

Acknowledgments

I had many people to support me in my time as a PhD student here. Through their help and guidance, many things that would have been impossibly heavy were light instead, and any hard work has remained worth the effort.

First, I would like to acknowledge the work of my advisors, Steve Tanimoto and Zach Tatlock. I am deeply grateful to both of them.

When I began working with Steve five years ago, I was unfamiliar with concepts like live programming and design space search which are now basic parts of the way I think about programming system and interface design. Based on nothing but an application essay, Steve took a chance on me and helped to develop these perspectives. Steve's clear enthusiasm for discussing these and other novel ideas with me, as well as his respect and patient encouragement of my own opinions and ideas, have changed the way I think and regard myself and the way I have worked with my own students. At the same time, Steve has held the work I have shown him to a high standard that helped me to grow as a writer and researcher.

Zach took a significant chance on offering me a position as part of a

research project and has provided constant support whenever I have struggled with the rapid pace and demands of a substantial ongoing project. He also later took another chance on becoming my co-advisor. Through Zach, I have learned most about acting with conviction and urgency. He has also lent me the resources necessary to learn about project management and delegation. Zach has shown me how powerful it can be to take basic principles like "the perfect is the enemy of the good" and "many hands make light work" and push them as far as they can go.

Finally, Steve and Zach have both shown a continuous interest in my well-being outside of research. This kind of relationship feels natural to me now, but I don't think it should ever be taken for granted.

I'd additionally like to thank Pavel Pancheckha, who was a project co-advisor for my work on Odyssey. I cannot thank Pavel enough for being incredibly direct and accurate in everything he does and says. I'm quite relieved, because from now on, whenever I need to be most careful and focused, I can just imagine my internal Pavel watching and critiquing the work bluntly but completely fairly, in manageable steps, and iterate from there without fear.

There are a number of other people I should acknowledge here.

I'd like to particularly thank Eunice Jun, who took me on as a research assistant and collaborator on the rTisane project. Anything that is good about my work with user studies and writing papers for an HCI audience thus far can be credited completely to Eunice.

I'd also like to thank René Just and Jeff Heer for the privilege of being their collaborator and for conducting some of the most interesting classes I took as a graduate student.

I'd like to thank Amy Ko for her work as the GSR for my final examination, and René again for serving on this paper's reading committee.

The following people helped me find my way into programming and on to grad school: Professor David Kosbie (CMU), Professor Daniel Mossé (Pitt), Donald Shoup and Elizabeth Cook (MEPPI), Thomas Weng and Robert Hönig (Recurse Center), my neighbor Dr. Dean Pomerleau (CMU), and Professor Martina Rau (UW-Madison/ETH Zurich), who helped me to navigate the application process.

I'd like to thank the RAs I've worked with over the past several years for tackling a significant amount of work and learning from the process, and for being a joy to advise: Arabella Yao, Elias Martin, Caleb C. Chan, Rich Chen, Ben Wang, Erik Vank, Jaela Field, Nick Baret, Parth Desai, and Zane Enders.

Without the work of certain people in UW advising and administration like Joe Eckert and Anna Wehowsky, I would not have graduated or gotten my travel reimbursements on time, so I am quite thankful they were around.

The whole PLSE lab has been very important to my time at UW, and I want to particularly thank Yihong for being my accountability partner, Oliver for being around to watch anime and play games, Haobin for carrying me in Monster Hunter, Amy for having the fanciest whiskey, Kevin for being the

other Vocaloid fan in the lab, Brett for being an adult, Anjali for excellence at Set, Audrey for excellence at Roq proofs, Steven for being a great Bugsplorer with me, and Ben, for being fun to talk with and for his excellence at poker.

I also would like to thank the members of the Race Condition Running Club, including Chandra, Ellis, Ewin, Gus, Max, Ethan, Reshabh, Yuxuan, and Nick.

I am very grateful to a particular group of previous UW grads who hung out and played tennis with me when I first arrived in Seattle. I learned a lot of what I know about how to be a happy PhD student from these people: Ofir Press, Nathan Hatch, Sam Ainsworth, Ian (and Darcy) Covert, Ivan Evtimov and others.

I'm grateful to a number of other students at UW, including Jasper, Kentrell, Kaiming, and Alisa, for always being super friendly; to Josh Horowitz for always being happy to look at papers and talk with me about his cool projects and to discuss ideas about the future of programming; and to Champ and Samantha for hanging out at the Japanese study club.

I would be remiss if I didn't mention Purva from bossasaservice.com, who has helped me stay on track with my goals for the past couple of years. Purva has made a huge difference in my life by sending me one short message a day.

We also have the personal non-UW friends, beginning with around 30 anonymous gamers whom I cannot name here without doxxing myself, but have already thanked individually.

My named friends include Dennis Sy, who has helped me learn Japanese

together every week or two for about 7 years now; Anthony Verardi, who has, without meaning to, changed my life multiple times; and Kevin Shebek, who remains undefeated as my best friend.

Finally, I'm very grateful for my therapist Hayden, without whom I would not have as many of these friends, and deeply thankful for William and Charles, Christine and Robert, Thor, Montague, Metolius, and Winston, Fenrir, Luke, Kerry, Sampson, and Sandy, and Jan, who does not use computers much but will hopefully enjoy hearing of this work's completion.

Chapter 1

Introduction

Software development is an inherently complex process that requires developers to maintain extensive mental models of the systems they build. These mental models encompass far more than the code itself—they include design rationales, implementation trade-offs, performance considerations, edge cases, and debugging insights [LVD06] [LAK+23] [HLHF23] [MLAC20]. Yet despite significant advances in programming tools and environments, much of this vital context remains un-externalized—it exists primarily in developers' minds or fragmented across disconnected artifacts like documentation, chat logs, and personal notes [MLAC20] [LVD06].

This "context gap" creates significant challenges. When developers return to their own code after weeks or months, they often struggle to reconstruct their original reasoning. When code is handed off to new team members, critical context is frequently lost, increasing onboarding time and maintenance

costs. As codebases grow and evolve, this problem compounds: the distance between code and its context widens, making development more difficult, error-prone, and frustrating.

Recent developments in AI-assisted programming have highlighted this issue from a new angle. Large language models (LLMs) can generate code and provide suggestions, but they face the same limitations human developers do when working with code that lacks adequate context. LLMs may hallucinate or make incorrect assumptions when the codebase fails to externalize important context [MIT23]. This demonstrates that the context gap is not merely a human cognitive limitation, but a fundamental issue in how we represent and work with code.

Traditional approaches to code contextualization have serious limitations. Code comments intermingle with the code itself, cluttering the codebase and creating maintenance challenges as comments drift out of sync with evolving code. Documentation lives separately from code, forcing developers to manually maintain connections between explanations and implementation. Computational notebooks like Jupyter notebooks [KRKP+16] provide richer context but at the cost of fragmenting code into specialized environments disconnected from production systems.

This dissertation addresses these challenges by developing a framework for enhancing program context for typical codebases for all agents interacting with the code. The core insight is that more intelligent programming tools can allow rich contextual information to be externally attached to code

while maintaining connections between the code and that context as code evolves. Specifically, I point out that humans are able to maintain connections between a section of the code and notes which pertain to the meaning of that section, and claim that intelligent programming tools can follow a similar process to keep such notes attached or *anchored* to the idea behind those sections (rather than their exact text, which may change) by reasoning about the combined meaning of the sections and their notes (*semantic anchoring*). This approach combines the best aspects of both integrated documentation (comments, notebooks) and external documentation, enabling context that is both rich and persistent.

The central claim of this dissertation is that

> *programming environments can and should now provide semantically anchored, persistent contextual information to enhance program understanding and manipulation for both human developers and AI agents.*

The work described here spans three interconnected systems:

1. Odyssey: A workbench for floating-point expression analysis that transforms an automated tool into an interactive, user-driven process, providing a concrete example of how contextual information enriches a complex programming task.

2. Magic Markup: A system that uses LLMs to "magically" maintain connections between evolving documents and external annotations through

semantic understanding rather than brittle syntactic approaches.

3. Codetations: A comprehensive VSCode extension that combines the insights from the previous systems, allowing developers to attach rich, interactive notes and tools to code while robustly tracking changes through a hybrid approach of edit tracking and semantic anchoring.

Together, these systems demonstrate how we can bridge the context gap in modern software development by creating persistent, intelligent connections between code and its surrounding context. The approach enhances programming for human developers while simultaneously making code more accessible to automated agents. This dissertation lays the groundwork for a future where code is no longer divorced from its context but seamlessly integrated with the knowledge needed to understand, modify, and extend it.

1.1 Overview

The work in this dissertation began with a specific problem domain—floating-point expression optimization—and progressively expanded to address the broader challenge of maintaining rich context for evolving code. This section provides a chronological overview of how each project built upon insights from the previous one, leading to a comprehensive framework for contextualizing code.

The work started with Odyssey, an interactive workbench for expert-driven floating-point expression rewriting. Floating-point arithmetic is noto-

riously challenging, with subtle rounding errors that can dramatically affect program correctness. Automated tools like Herbie [PSSWT15] had been developed to identify and fix such errors, but users struggled to effectively integrate these tools into their workflows. Through interviews with both novices and experts, my team discovered that users needed a more interactive process that allowed them to diagnose problems, generate solutions from multiple sources (including automated tools and their own expertise), and tune the results based on domain-specific requirements.

Odyssey addresses these needs by implementing a three-stage workflow that combines the power of automated tools with user-driven exploration. The system provides visualizations for diagnosis, a central repository for collecting and comparing rewritings, and tools for tuning expressions to meet specific accuracy and performance goals. Evaluations with numerical computing experts show that Odyssey enables them to solve challenging floating-point problems by leveraging both automated suggestions and their own domain knowledge.

A key insight from Odyssey was that much of its value came from providing contextual information about numerical expressions—error visualizations, alternative rewritings, and derivations—that helped users understand and improve their code. This raised a critical question: how could such contextual information be persistently attached to code in real-world programming environments, where code evolves continuously?

This question led to Magic Markup, a system that explores how to main-

tain semantic connections between document sections and associated metadata as documents change. Traditional approaches to document annotation anchoring rely on syntactic features like string matching or offset tracking, which break down when documents undergo significant changes. Magic Markup introduced a novel approach leveraging large language models to understand the semantics of document sections, allowing annotations to follow the meaning of code rather than its exact textual representation.

To evaluate this approach, I developed the Tagged Code Updates benchmark, a synthetic dataset of code snippets that undergo various transformations while maintaining semantic equivalence. Experiments showed that LLM-based anchoring could successfully re-attach annotations after significant code changes, outperforming traditional anchoring methods. This demonstrated the feasibility of using semantic understanding to maintain connections between code and context as code evolves.

Building on these insights, Codetations expanded the Magic Markup approach into a complete system for attaching rich, interactive notes to real codebases. Codetations, implemented as an extension for Microsoft's popular Visual Studio Code editor, combines real-time edit tracking with LLM-based reanchoring to robustly maintain annotation positions across both online and offline edits. It introduces an architecture where annotations are not mere comments but can contain rich data including interactive UIs that respond to code changes and provide live feedback.

Through user studies with developers, I found that Codetations addressed

key pain points in current documentation practices: fragmentation across disconnected systems, difficulties in keeping documentation synchronized with code changes, and limitations in expressiveness. Developers recognized the potential of document-external annotations to store more context with their code without cluttering the codebase itself. The system also demonstrated benefits for LLMs working with annotated code, showing improved performance in code repair tasks when contextual information was available.

This body of work progresses from addressing a specific numerical computing challenge to developing a general framework for code contextualization. The three systems—Odyssey, Magic Markup, and Codetations—each tackle progressively broader aspects of the fundamental problem: how to maintain meaningful connections between code and its surrounding context. The result is a practical approach to external, persistent context management that benefits both human developers and automated tools without sacrificing code clarity or introducing maintenance burdens.

1.2 Contributions

The work in this dissertation explores ways of giving programmers access to much more contextual information and automated analytical power during parts of the programming process through a combination of specialized tools and recorded information that are brought to bear on particular points in a program.

The work aims at increasing user capabilities when tackling hard problems like design space search during the programming process. It begins by studying this type of problem through through the particular lens of floating-point expression rewriting, finding that contextual information like analyses and improvement suggestions from high-level tools can enable a user-driven expression improvement workflow. The work then seeks practical and general ways such contextual information could be attached to programs in the face of real-world obstacles like read-only documents and significant code refactoring. It also includes discussion of the implications of attaching context to programs for LLMs.

My work has made the following contributions toward the problem of externalizing and associating development context with the developed code:

- An investigation of the needs of novices and experts during the specific process of floating-point expression rewriting. (Section 3.5)
- An iteratively-developed workbench, Odyssey, that supports a three-stage workflow to address those needs, involving diagnosis, solution generation, and tuning. This workflow combines both automated tools and human rewritings. (Section 3.4.2)
- A user evaluation of Odyssey, with valuable lessons for design space search tools in general. (Section 3.8)
- A vocabulary for annotations incorporating the notion of annotation intent, which is vital for the maintenance of annotations as a document

evolves. (Section 4.3)

- Magic Markup, a prototype LLM-based annotation anchoring method that leverages document semantics to outperform prior anchoring systems; I include notes on the LLM prompt used and our prompt development process. (Section 4.5)
- A synthetic Tagged Code Updates benchmark dataset (Section 4.4) and evaluation on the benchmarks (Section 4.6).
- A study of developers' needs when managing code-related contextual information. (Section 5.5.2)
- A discussion of design considerations when implementing an annotation management system. (Section 5.3.3)
- Codetations, a system for keeping dynamic, interactive annotations on semantic entities as documents evolve. (Section 5.4)
- A qualitative user evaluation of Codetations highlighting its benefits (Section 5.5.3) and potential for impact on future programming processes (Section 5.5.4).
- A detailed exploration of several kinds of annotation types enabled by Codetations. (Section 5.6)
- A worked example exploring the benefits and risks of use of Codetations-style context by a LLM. (Section 5.7)

Chapter 2

Background and Related Work

2.1 Annotations

This section examines the landscape of annotation systems. We begin with a historical overview, then focus specifically on the challenge of anchoring annotations to code, examining approaches ranging from XML representations and IDE integration to string similarity matching and our proposed semantic anchoring method. We refer to a prior system that uses annotations themselves as an interface to IDEs. Finally, we examine the integration of rich and live programming tools as code annotations, drawing insights from computational notebooks to inform our approach to embedding interactive tools directly within traditional codebases.

2.1.1 History

As Sedig and Parsons observe in their comprehensive analysis of interaction design patterns, annotation serves as a fundamental pattern through which users externalize their understanding and engage with visual representations of information (including text) to enhance cognition [SP13]. This cognitive importance of annotation has driven decades of research into digital annotation systems.

Early Systems and Foundations

Early work on digital annotations faced fundamental challenges in maintaining connections between annotations and evolving documents. In systems that are able to observe edit actions, schemes like the "sticky pointers" of Fischer and Ladner [FL79] provided one of the first solutions, allowing annotations to follow text as it moved within a document during editing. However, offline systems—where edits occur outside the annotation system's observation—required different approaches for dealing with arbitrary document updates.

The late 1990s saw systematic efforts to catalog and understand digital annotation technologies. An early survey by Ovsiannikov et al. examined annotation systems for documents and multimedia, identifying common features that would become standard in the field [OAM99]. Their analysis revealed that most of the 17 systems they reviewed stored annotations in separate databases rather than embedding them in documents, supported

annotation search capabilities, and prioritized format insensitivity—the ability to maintain annotations despite changes in document format. They also documented early support for diverse media types, including handwritten annotations.

By the early 2000s, Wolfe had conducted another comprehensive review of 25 document annotation systems and their common features [Wol02]. Wolfe and Neuwirth argued that annotations were moving from peripheral academic curiosity to central importance in digital document interaction [WN01].

Robust Annotations and Formal Definitions

As annotation systems matured, researchers developed more sophisticated approaches to the anchoring problem. Work on robust annotation systems for digital documents (like Microsoft Word documents and HTML pages) advanced the field by accounting for user expectations through methods like keyword anchoring [BBGC01]. Annotation orphaning—the loss of tag position when documents change—emerged as a key problem, with robust systems needing explicit strategies for dealing with orphans.

Agosti et al. laid theoretical foundations for the field through historical studies and formal models that clarified the design space for digital annotation management systems [ABDF07, AF07].

Collective and Social Annotations

The early 2000s witnessed a shift toward collaborative knowledge creation and sharing. Focus moved from individual annotation tools to systems that supported collective annotation [CCCJ07]. This shift aligned with broader movements toward collective intelligence on the web, including the semantic web vision, which sought to enable machines to understand and process the meaning of web content through structured annotations and metadata [SBLH06].

Collaborative annotation systems emerged across multiple domains. In data visualization, Heer et al. demonstrated how shared annotations could enhance asynchronous information visualization, allowing users to share insights and build upon each other's observations [HVW07]. In information retrieval, systems were developed to leverage collective annotation for entity tagging and information extraction across large document sets [KSRC09].

The educational domain particularly embraced social annotation, in which users can annotate a document together. Sun's systematic review of this two-decade span identified key trends including the rise of collaborative learning through shared annotations, the development of specialized educational annotation platforms, and the evolution of pedagogical practices around social reading [SHY+23, NRJ12].

In software development contexts, studies revealed that developers employ annotations primarily for reminding and refinding purposes, creating personal information spaces within large codebases [SRS+09]. This work

highlighted the challenges of annotating code, where the underlying text changes frequently and annotations must maintain semantic rather than purely textual connections.

Beyond Digital Documents

Annotations also began expanding beyond traditional digital documents. Hansen's analysis of ubiquitous annotation systems provides examinations of how annotation could span digital and physical environments, including various anchoring methods such as physical markers, RFID tags, and GPS-based systems [Han06].

More recently, augmented reality (AR) has opened new frontiers for annotation systems, with multi-user collaborative AR systems allowing teams to annotate physical spaces and objects in real-time [WQ22].

Automated Annotation Creation

Most recently, the field has begun exploring the automation of annotation tasks through large language models (LLMs). Studies have demonstrated that LLMs can outperform crowd workers for certain text-annotation tasks, particularly in information extraction [GAK23]. This development shows that annotation creation—historically a human-intensive activity—can now be augmented or automated by AI systems in specific contexts.

This historical progression illustrates not just technological advancement but a fundamental evolution in how we conceptualize annotations: from sim-

ple marginalia to formal, collaborative, multi-modal information layers that span digital and physical spaces. We build upon these foundations, particularly the early insights about annotation orphaning and the need for robust anchoring mechanisms, to address the unique challenges of maintaining annotations in modern programming environments.

2.1.2 Annotation Anchoring for Code

Several efforts have attempted to solve the problem of anchoring annotations to positions in code.

Using XML or IDEs

In 2002, Collard described an XML representation of all code [CMM02] meant for general adoption; however, it is infeasible today to implement this editor-managed solution globally (but perhaps not locally) for modern programming environments and unstructured documents. Juhár [Juh19] distinguishes language-level annotations that are part of code, structured comment annotations, and external annotations requiring the support of a special system, typically an IDE. These types differ in how annotations are defined, applied, and in what code elements they are able to bind to, and Juhár develops an IDE-based annotator that maintains annotation positions externally.

String and AST Similarity

Keeping annotations attached to the right entities is closely related to the question of how to track changes in code over time. Reiss labeled changes in 53 lines of code across 25 versions of a Java source file, then evaluated 18 tracking methods with various parameters to find that a relatively simple, low-computational-cost combination of string similarity [L+66] and context comparison yielded the best results [Rei08]. Reiss's tracking method has been used in other systems for attaching annotations to a tracked line, including Horvath et al.'s Catseye [HMMR22] and Sodalite [HMM23] systems for adding comments to source code and allowing users to reference source code in documentation, respectively. Horvath et al.'s idea in these systems is notable because it is very similar to our own: when Catseye or Sodalite load, they attempt to reattach old annotations to the file using Reiss's tracking method. Horvath et al. note that this method was able to resolve 86.5% of cases in their testing. However, the method has no semantic awareness and still suffers from the text similarity issues we've noted previously.

A number of clone detection systems have been designed to find similar code that has been copied around a code base [DER10] [RBS13]. These systems are typically specialized to handle specific kinds of copying, and require domain-specific parsers to handle code semantics. However, such systems may provide extra guarantees about equivalence when a copied piece of code is detected.

Further, Horvath et al.'s Meta-Manager introduces an automated process

for tracking the provenance of code edits that could further enhance annotation systems by attaching edit history and developer interactions with external tools directly to relevant sections of the codebase [HMM24]. MetaManager follows version history of code snippets by parsing the AST syntactically using the method of Wittenhagen et al. [WCB16] and is currently restricted to tracking user edits on TypeScript files in the VSCode editor.

Semantic Similarity

Our key insight follows historical discussions about notions of program similarity. Walenstein et al. break program similarity into two types–representational similarity (concerning text, syntax, and structure) and semantic or behavior similarity (concerning a program's function or execution) [WERC+07]. As this thesis points out, an LLM's understanding of document semantics allows it to outperform Reiss's method with a simple prompt to maintain the positions of notes through *semantic anchoring*, matching human expectations for an anchoring system (term defined in Section 1; see Chapter 4 for a full description of the problem and the prompt used for our solution). Our proposed system uses this method despite some weaknesses in its performance when using current LLMs (see Section 5.4 and Appendix 5.10.6).

2.1.3 Annotations as IDE

Notably, Reiss's work on the Field programming environment [Rei90] considers the implications of attaching arbitrary tools to parts of a program.

Field annotations are maintained on a document by the editor, and system applications can subscribe to updates from those annotations. A pilot study (Section 5.3) motivated the user evaluation and implementation of Codetations to focus on a more constrained editor-embedded system with tighter coupling than Reiss imagines in order to help users recognize possible workflows.

2.1.4 Rich Tools for Live and Literate Programming

Codetations applications must consider the problem of embedding *live* and *rich* tools in codebases as program annotations. Tanimoto [Tan13] describes *live programming systems* as those that can be edited as they are running to obtain immediate feedback on the results of a programmer's actions. Horowitz and Heer [HH23b] define *rich systems* as allowing a programmer to edit programs through domain-specific visualizations. A broad range of systems meet these definitions. Horowitz and Heer [HH23a] note that live and rich tools and systems face challenges with inter-operation; in particular, problems arise when trying to inter-operate with the "tools and environments in the outside [non-live, rich] world." Our system provides a new way to attach these tools to typical codebases.

Many prior works have attempted to integrate live and rich tools with standard programming tools. Computational notebooks are perhaps the most successful example; these are a form of *literate programming* [Knu84] wherein code can be attached to detailed explanations and tools for data

visualization. Their typical interface, which lists executable cells (occasionally displayed out-of-order to aid in presentation) resembles annotations that might be shown in the margin of another document. As such, we considered Lau et al.'s description of the design space of computational notebooks [LDMG20] in the design of our own annotation hosting interface. Unlike Codetations, notebooks require a codebase to be written and accessed using special notebook formats and editor interfaces rather than augmenting existing code and editors.

The recent work of Gobert and Beaudouin-Lafon on Lorgnette [GBL23] details a subtype of live and rich tools called *projections* and an implementation of a projection hosting system with customizable tools very similar to the more general tool hosting system presented here. Unlike Codetations, Lorgnette's tools are attached to all instances of a pattern using regex matching and cannot save the state of an individual instance outside of the code.

2.2 AI-powered Code Editors

More recently, popular AI-powered code editors have emerged that address contextual information in different ways. Notably, the Cursor editor [dt] includes "rule" annotations, stored in their own subdirectory of a codebase, and an implementation of the Model Context Protocol (MCP) to provide standard ways for developers to add and connect AI assistants with context. Our Codetations system takes a complementary approach by allowing

developers to explicitly attach persistent, rich annotations directly to code spans.

Chapter 3

Odyssey

This chapter is adapted from my paper:

[MCS+23] Edward Misback, Caleb C. Chan, Brett Saiki, Eunice Jun, Zachary Tatlock, and Pavel Panchekha. "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*, pages 1–15, 2023.

3.1 Overview

The first study in this dissertation examines how experts analyze and improve floating-point expressions, a domain where small errors can cascade into significant computational inaccuracies. Odyssey addresses a crucial gap between automated tools like Herbie and the workflow needs of numerical analysts. By

transforming a batch-oriented command-line tool into an interactive workbench, Odyssey demonstrates how contextual information—visualizations, derivations, and alternative solutions—significantly enhances expert capabilities. This chapter establishes a foundational premise of the dissertation: that developers benefit from rich, persistent contextual information that complements rather than replaces their expertise. The three-stage workflow identified here (diagnosis, solution generation, and tuning) reveals patterns of context use that inform later systems and highlights the importance of making contextual information accessible, comparable, and editable.

3.2 Introduction

Floating-point arithmetic is widely used in scientific, engineering, and graphical applications to approximate arithmetic on real numbers; typically, it is the only practical option available.[1] However, floating-point arithmetic must be used with care, as rounding errors can cause floating-point arithmetic and real-number arithmetic to give dramatically different results. For example, naïve implementations of well-known mathematical equations like the quadratic formula can exhibit unacceptably-high rounding error (Figure 3.1b). Rounding error can also ruin results for even extremely simple expressions. Figure 3.1a shows that, for large floating-point values of `x`,

[1] While alternatives exist, e.g., arbitrary-precision arithmetic, exact rational arithmetic, and constructive real arithmetic, they are orders of magnitude slower than hardware floating-point, and are thus inappropriate for many applications.

the expression `x + 1 - x` can evaluate to 0 instead of the mathematically correct 1! Floating-point rounding error has caused unreproducible scientific research, distorted stock market indices, and wartime casualties [AM03, MV99, U.S92, Eur98, Qui83, AGM03, WW92].

As a specific example, a major bug in the implementation of asinh/acosh in the Rust standard math library went unnoticed for seven years. An automated test suite caught the bug in 2022 [Pan22].

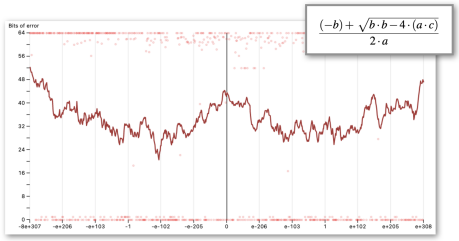
In order to diagnose and repair this kind of error, *numerical analysis experts* have developed techniques and tools for analyzing and rewriting floating-point expressions over the last decade. These tools support and facilitate automated test generation [CGRS14], error analysis [BFM09, GP11, ID17, DBG+20, SJRG15], and repair [DM17, PSSWT15]. For example, the open-source, state-of-the-art Herbie tool [PSSWT15] takes as input a floating-point expression and uses algebraic and analytic identities to rewrite the expression via a complex search process. Despite wide adoption of tools like Herbie in industrial and national labs, users still find results are too complicated and that tools overlook seemingly obvious rewritings.

Fully understanding and fixing the bug in Rust required rewriting the naive definition $\log(x + \sqrt{x^2+1})$ as $\mathrm{log1p}(x + \frac{x}{(\mathrm{hypot}(1,x)+x)})$. To arrive at the solution, numerical analysts needed to repurpose internal operations of existing tools and apply their own expert knowledge. This example illustrates how experts must work with a constellation of complicated analysis tools, none of which answer their questions about an expression directly. Our

```
    # FP arithmetic
 seems ok
    >>> x = 1e15
    >>> x + 1 - x
    1.0

    # ... until it
doesn't!
    >>> x = 1e16
    >>> x + 1 - x
    0.0
```

(a) Example of "catastrophic cancellation" in Python.



(b) Average "bits" ($\log_2(ulps)$) of floating-point error with respect to b when evaluating the quadratic equation over randomly-sampled inputs. For many applications the error is unacceptable, but few programmers are equipped to address such numerical issues.

Figure 3.1: Floating-point error is pernicious; even familiar, simple expressions can yield meaningless results.

goal is to enable numerical analysis experts and developers of mathematical libraries to find and fix similar bugs and prevent their occurence in the future.

Towards this goal, we observed novices and experts in an in-lab design study and found that users struggle with specifying their objectives and interpreting Herbie's results, facing issues of tool/user objective mismatch, lack of trust in the automated tool, and a need for independent exploration. We also identified a three-stage floating-point rewriting workflow: (1) *diagnosing problems*, in which users identify the problematic operations within expressions; (2) *generating solutions*, in which users gather potential expression rewritings from automated tools, references, or their own creativity; and (3) *tuning*, where users test, tweak, and compare different rewritings to optimize the resulting expression for their own accuracy, performance, and maintainability needs. This workflow is not well-addressed by existing tools. For example, end-to-end tools like Herbie can take minutes to return a batch of analysis results, and there is no tool support for comparing and improving rewritings drawn from multiple sources.

To support this workflow, we designed and implemented Odyssey, an interactive workbench that allows users to identify problem areas in floating-point expressions using error visualizations, collect and manage expression rewrites using an interactive table, and combine rewrites to minimize rounding error. Odyssey leverages Herbie as an analysis and rewriting engine but retains context about the user's objectives, allowing it to return common analyses in less than a second.

To evaluate the effectiveness of Odyssey, we conducted a study with five experts in numerical computing and floating-point arithmetic. On average, the experts successfully completed five out of seven challenging tasks drawn from real-world numerical problems in roughly 40 minutes after a 12-minute tutorial on the use of Odyssey.[2] The interactive nature of Odyssey enabled experts to concentrate on high-level problem-solving and facilitated the swift evaluation and comparison of expression rewritings.

Odyssey contributes to a growing body of work on expert tools. Unlike end-users, experts have highly specialized workflows and significant low-level implementation knowledge they need to express and incorporate in tools. Examples of expert tools include Roly-poly, a tool for guided optimization of Halide image processing code [IRKF+21]; PerformanceHat, a tool for analyzing application runtime performance [CLB+18]; and Tsugite, a tool for interactive design and fabrication of wood joints designed for expert machinists with limited experience working in a particular domain [LYUI20]. By combining the power of automated systems with a dynamic, human-driven workflow, Odyssey is an example of how to enable more users to work efficiently along-side automated tools in complex domains beyond floating-point.

This paper makes four contributions:

1. An investigation of the needs of novices and experts, summarized in a three-stage workflow for floating-point expression rewriting: diagnosis,

[2]No comparison was made to a condition without Odyssey in this study since these kinds of problems may each take time on the order of hours to debug and resolve; an example of this process, performed by one of my co-author Pavel, is shown in [Pan22].

solution generation, and tuning. This workflow combines both automated tools and human rewritings.

2. An iteratively developed workbench, Odyssey, that supports this workflow.

3. A study of Odyssey's effectiveness based on feedback from expert users who completed a set of challenging tasks drawn from real-world numerical problems.

4. A discussion of the implications of our work for the design of interactive expert tools that combine human and automated design space search.

3.3 Background and Related Work

Odyssey draws on techniques from the developer tool literature on program visualization and program history to addresses key challenges developers face in the domain of floating-point arithmetic.

3.3.1 Program Visualization for Debugging

Floating-point error analysis and repair involves a mix of debugging and performance optimization work. Odyssey is thus inspired by work aimed at program visualization for debugging. Systems such as Whyline [KM04], Timelapse [BBKE13], and FireCrystal [OM09], which connect code with runtime behavior by visualizing execution traces, inspire several of Odyssey's

interactions, including the interactive "local error" heatmap visualizing per-operation floating-point error for a particular input. Moreover, a series of papers on integrating visualizations with code, such as Theseus [LBM14], which provides always-on visualizations of runtime state; Projection Boxes [Ler20], which gives programmers more control over which runtime values are visualized; and Hoffswell et al. [HSH18], which provides recommendations for embedding visualizations in code, are reflected in our design of Odyssey's error graph, which allows programmers to visualize floating-point error and control which input values and rewritings are visualized. Odyssey sees similar benefits from these designs as prior work: opening up space for programmer exploration and observation, and thereby giving programmers a fuller understanding of the problem space and a richer set of interactions for comparison and repair.

That said, floating-point rounding error is a continuous, numeric quality of a program, and the "tuning" stage of numerical work therefore has a lot of analogs to performance optimization. Beck et al. [BMDR13] and PerformanceHat [CLB+18], for example, visualize the proportion of runtime spent at each each line of code in the program. These approaches inspire our "heatmap" design for local error information, coloring each floating-point operation in the program based on the amount of floating-point rounding error it contributes to the result. The Roly-poly [IRKF+21] project is also quite similar to Odyssey, aiding developers in exploring and selecting performance optimizations for image processing code. Odyssey explores a similar

system-aided optimization workflow, but for accuracy instead of performance optimization.

3.3.2 Maintaining and Reviewing Code Versions

To understand, experiment with, and collaborate on code, developers author and compare multiple program alternatives and histories [CRDB15]. Tools such as Azurite [YM15], Verdant [KM18], and Variolite [KHM17] provide explicit support for multiple program versions. For example, Verdant helps data scientists compare, replay, and simplify histories for code in computational notebooks [KM18]. Also, Head et al. [HHB+19] introduce "code gathering" techniques that find the minimal code slices in a program that produce a selected set of results. Comparing and combining multiple alternative rewritings is a also key part of floating-point error repair.

Odyssey maintains a history of rewritings both to provide a history of how a rewriting was developed and also allow developers to visualize, compare, and combine multiple alternatives, providing explicit internal support to what would otherwise be internal mental operations, thereby reducing cognitive load and allowing developers to focus on the higher-level problem-solving aspects.

3.3.3 Floating-Point Arithmetic and Numerical Analysis

Floating-point arithmetic, defined by the IEEE 754 standard [IEE08], and variations of this standard form the standard number representation in most programming languages [Mon08]. However, floating-point arithmetic is subject to rounding error, and even elementary computations often permit significant error [Gol91]. Numerical analysis provides a set of mathematical tools to analyze, bound, and reduce this error [Ham87]. However, many programmers are unfamiliar with numerical analysis techniques, and even fewer have a thorough understanding of how to apply these tools.

Researchers have thus developed a vast menagerie of tools automating specific numerical analysis techniques, including Rosa [DK14] for affine arithmetic, FPTaylor [SJRG15] for error Taylor series, and Ariadne [BVLS13] for root finding. Other tools repurpose static analysis techniques to find floating-point rounding errors; such tools include Fluctuat [GP11], which uses abstract interpretation; FPDebug [BHH12], which uses a dynamic execution with shadow variables; and CGRS [CGRS14], which uses evolutionary search. These tools can find inputs with high rounding error or, in some cases, certify the absence of such errors. Programmers can then use the error found to attempt to understand the source of the rounding error, and ultimately fix it. One popular tool combining these steps is Herbie [PSSWT15]. Herbie uses sampling techniques to identify floating-point error; constructs candi-

date rewrites using algebraic and analytic identities, and tests those rewrites against higher-precision executions to identify the rewrite with the lowest floating-point error. In recent releases, Herbie can output multiple suggestions with different performance and accuracy characteristics [SFN+21].

Unfortunately, all of these tools, Herbie included, are difficult for developers to use and integrate into their workflows. Users are typically expected to identify the expression and inputs of interest up front; compare them to other sources or the user's own ideas; and make trade-offs between accuracy and other goals (e.g., maintainability), all without tool support. Users are often recommended to switch between their code editor, version control system, a mathematical visualization tool, and multiple Herbie instances in order to solve a single problem [Kne17]. VSCode-PRECiSA [MD23], a VSCode interface for the PRECiSA command-line tool [TFMM18] designed to support the process of analyzing a single program in several ways, is somewhat of an exception; however, it does not address the problem of tool interoperation. We developed Odyssey to address these limitations by providing a single integrated workbench for the full floating-point rounding error workflow. To lower the barriers to adoption, Odyssey uses Herbie, a widely used and open source tool [Tea , Kne17], under the hood.

3.3.4 Expert Tools for Design Space Search

Odyssey is an expert tool for numerical analysts to re-write floating point expressions. Unlike tools for end-users, expert tools are designed for users

with extensive design and implementation experience. Experts have honed specialized workflows, leverage insights to improve upon automated or semi-automated approaches, and are comfortable wading into low-level details. For example, expert developers optimize the performance of applications [CLB+18] and specialized pipelines. In the domain of high-performance image processing, Roly-Poly [IRKF+21] is a system built on top of the Halide compiler [RKBA+13] for expert engineers to explore trade-offs and decide among possible optimizations. Odyssey is similar to Roly-Poly in that it supports interactive workflows with an automated tool to support expert users. In the statistical analysis domain, multiverse analysis tools such as Boba [LKAH20] and Multiverse Debugger [GJA23] enable expert statistical analysts to assess the robustness and sensitivity of analysis results. The intended users are experts in statistical analysis but not necessarily in multiverse authoring. Similarly, Tsugite helps expert fabrication users create new wood joints [LYUI20]. Odyssey adds to this growing body of research on expert tools for a new domain, and we discuss key insights that could serve as design principles generalizable across domains (section 3.9).

3.4 Usage Scenario

Consider a hypothetical numerical analysis expert Alex, who has received a report that there is an issue in the `asinh` function of a popular programming

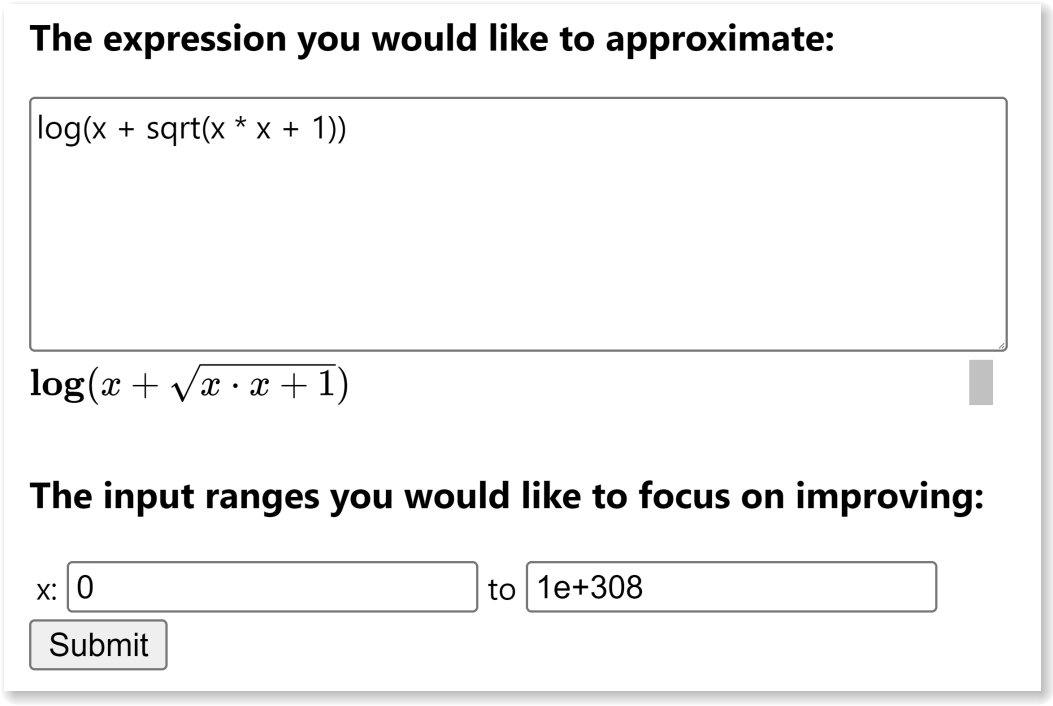


Figure 3.2: Users enter a new expression.

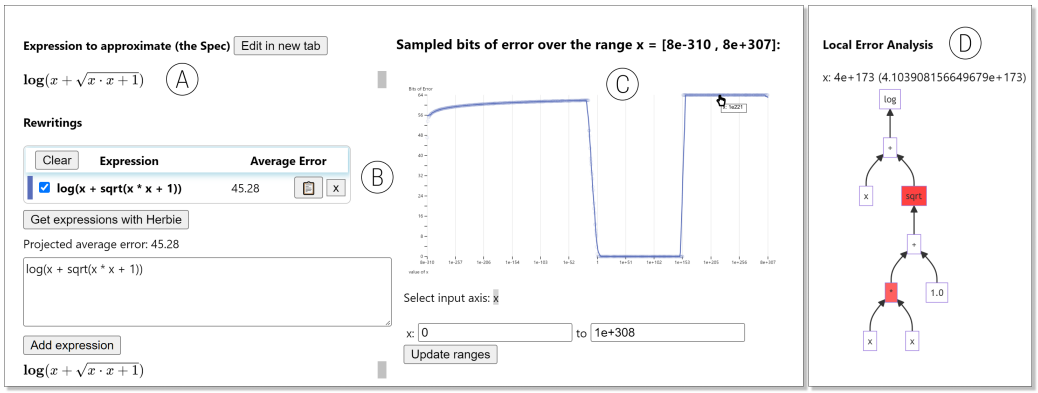


Figure 3.3: Diagnosis. The specification (A) shows the expression the user is trying to implement. The rewritings table (B) shows the expressions the user has tried. The error plot (C) shows the error of the current expression. The local error heatmap graph (D) shows the error breakdown of the currently selected point.

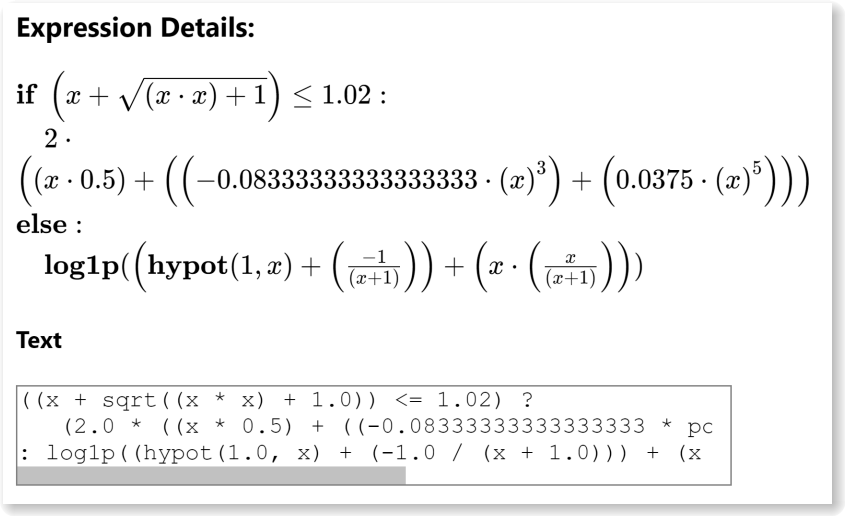


Figure 3.4: The Expression Details view shows a LaTeX rendering and plain text to help users understand and work with the selected expression.

language's standard library.[3]

Alex now needs to develop an accurate implementation of the `asinh` function.

3.4.1 A Typical Debugging Process

The `asinh` function is defined, for positive x, as $\text{asinh}(x) = \log(x + \sqrt{x^2 + 1})$. Based on the report, Alex hypothesizes that the issue involves the high range of the function's input. The x^2 term will overflow for large x.

Alex isn't immediately sure how to fix this, and turns to a state-of-the-art automated tool, Herbie, for help. Alex runs Herbie on the *asinh* expression. Herbie suggests a replacement expression and shows an error plot for the

[3]As mentioned earlier, this issue is based on a real-world problem that a numerics expert recently found and addressed for the Rust standard library using Herbie [Pan22]

original and final expressions.

Alex wants to start rewriting, but now faces a series of obstacles.

First, Herbie's error plot suggests that there is another source of error in the expression—small inputs, between 0 and 1. Alex needs to *diagnose the cause* of this error by finding a subexpression to rewrite. Alex sets up a REPL for the math library and manually steps through each subexpression. Alex considers its input and output ranges to see where errors occur.

Second, Alex needs to *generate new solutions* and test them. Although Herbie suggests a potential rewrite, it is still error-prone for small inputs. Drawing on their experience, Alex wants to try out new expressions, but Herbie does not support this. As a result, Alex abandons Herbie, writes a new expression, and sets up a new testing framework. Alex is frustrated about having to figure out how to set this testing up, even though Herbie has internal tools that are capable of this. Future iterations will require Alex to start all over and discourage Alex from exploring and finding an expression with more desirable error characteristics.

Third, Alex finds two rewrites which fix adjacent parts of the domain, and now wants to join them. This requires *tuning* the constant used for picking the branching point. However, in Alex's current test framework, the consequences of changing the constant are not evident. In other words, an iterative design process is not supported.

In order to address the above issues, Alex spends hours stitching together workarounds.

Alex needs an integrated tool designed for human-directed expression debugging and interactive rewriting. Odyssey is designed to help experts like Alex who fix floating-point bugs that impact the core of a programming language.

3.4.2 Using Odyssey

First Stage: Diagnosing Problems

Using Odyssey, Alex begins by typing the mathematical definition, `log(x + sqrt(x * x + 1))`, into Odyssey's expression entry box (see Figure 3.2), along with the range of possible `x` values. In this case, the definition is only valid for positive x, so Alex enters 0 as the lower bound. Since this is a library function that can be executed on any input, Alex leaves the default upper bound of 10^{308} in place. The expression and initial input range are used to initialize Odyssey's main screen (Figure 3.3) and appear in the top left corner of the screen (Figure 3.3A). If the user needs to launch multiple Odyssey sessions, this part of the screen will help them differentiate them.

Beneath the initial expression, Alex sees Odyssey's rewritings table (Figure 3.3B). The rewritings table allows the user to collect multiple versions (or "rewritings") of the expression and compare them for accuracy. Each rewriting in the table shows its average accuracy, and rewritings can also be selected or hidden to control the display of other information in Odyssey. Initially, the rewritings table contains a single rewriting, the direct imple-

mentation of their expression. In this example, the initial rewriting has quite high error (45.28 bits out of 64) indicating that there is quite some work left to do to produce an accurate implementation.

To better understand the source of this error, Alex refers to the error plot (Figure 3.3C). This plot shows the error of every rewriting in the table. The horizontal axis shows different input values x spanning hundreds of orders of magnitude; the vertical axis shows error, with higher values being worse. In this example, three regions are clearly visible: inputs $x < 1$, with high error; inputs $1 < x < 10^{150}$, with low error; and inputs $10^{150} < x$, with high error again. Distinct regions like these often have distinct causes of error and are a starting point for exploring more deeply.

To begin investigating, Alex clicks on one of the points in the error plot; this updates Herbie's "local error heatmap" display (Figure 3.3D). Local error is an internal heuristic in Herbie that identifies which operations in a rewriting cause rounding error at a given point. By clicking on one point with $10^{150} < x$, and another point with $x < 1$, Alex confirms that this expression has two distinct sources of error: for large inputs x, the source of error is the `sqrt` and `*` operations, while for small inputs x, the source of error is the `log` operation. After diagnosing the operations with error and the affected inputs, Alex begins generating solutions to these floating-point rounding error problems.

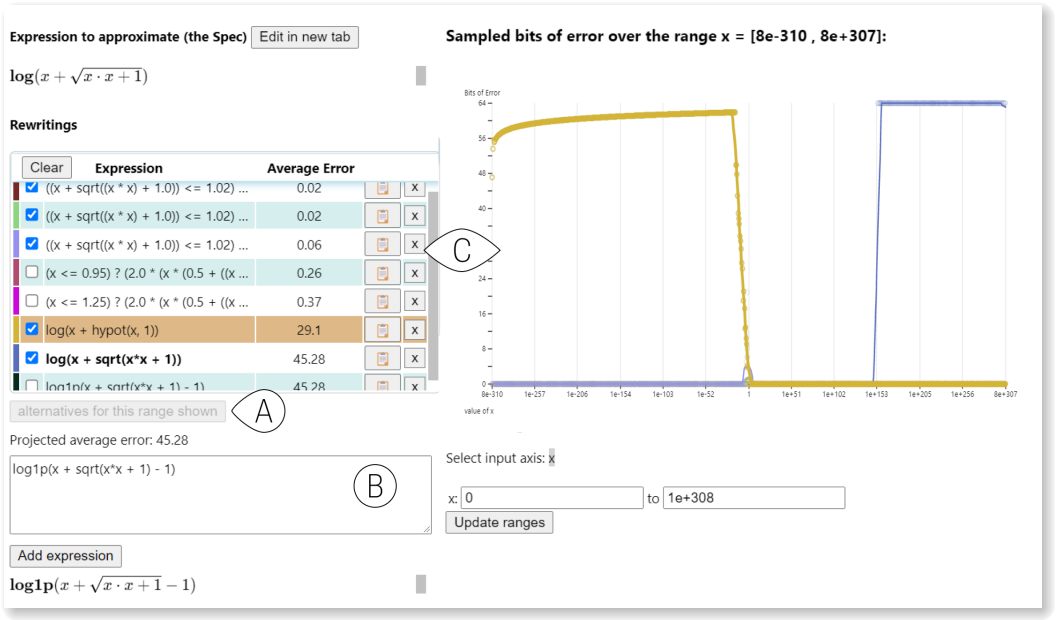


Figure 3.5: Solution generation. User can request rewritings from Herbie by pressing a button (A) or enter their own using the expression edit box (B), which provides live feedback and estimates the expression error on the current sample. The rewritings table and the error plot (C) are updated every time a rewriting is added, allowing the user to compare the quality of different rewritings.

Second Stage: Generating Solutions

To start generating solutions quickly, Alex queries an automated tool using the "Get expressions with Herbie" button (Figure 3.5A). This automatically translates the expression into Herbie's input format; invokes Herbie; evaluates the error of each of Herbie's suggestions; and translates each one back to a human-readable format.

In this case, invoking Herbie adds five suggestions to the rewritings table and to the error plot (Figure 3.5C). Since each rewriting in the table lists its error, Alex sees immediately that Herbie's suggestions reduce the original 45.28 bits of error to as low as 0.02 bits of error. Moreover, each rewriting's error is also graphed on the error plot, with different rewritings shown in different colors. Users can highlight the plot for an expression by clicking on its row in the table. For example, by clicking Herbie's fifth suggestion, `log(x + hypot(1, x))`, Alex sees that this expression avoids error for $10^{150} < x$ but still has error for smaller values of $x < 1$. Multiple suggestions will probably need to be consulted, compared, and combined to achieve Alex's accuracy, performance, and maintainability goals.

Herbie is not the only source of rewritings in Odyssey. In fact, human creativity is often needed to overcome roadblocks for automated tools, and rewritings may also be sourced from other tools, from papers, or from online references. Therefore, Odyssey allows Alex to add rewritings directly to the rewritings table using the edit box (Figure 3.5B). As they type, their expression is automatically rendered and an error estimate is provided, to help

avoid typos and other low-level mistakes. As Alex works on this expression, the table of rewritings will grow to contain all of the various rewritings or ideas that have been considered during the session. By leaving this basic organizational task to Odyssey, Alex is able to focus on high-level reasoning.

Third Stage: Tuning

After generating solutions to the various floating-point issues in this expression, Alex wants to understand how these rewritings can be combined to produce a single implementation of the expression that satisfies their accuracy, performance, and maintainability goals (Figure 3.6).

Since, in this case, many of the rewritings are generated by Herbie, Alex starts by understanding those rewritings in greater depth. To do so, Alex clicks on one of these rewritings and looks at the derivation provided for it (Figure 3.6A). The derivation of a Herbie-generated rewriting shows the sequence of steps Herbie used to produce it. Alex scans one derivation that has caught their attention both for ideas that can be lifted and combined with a different rewriting, as well as for potentially dangerous steps. In this case, Alex notices that Herbie used a Taylor series expansion to derive one of the rewritings. Taylor series expansions are dangerous, because they are often valid only for inputs in a certain range, and can lead to high error if used outside of that range. In this case, Herbie guarded the Taylor series with the conditional $x \leq 1$; however, it may be possible to tune the condition further.

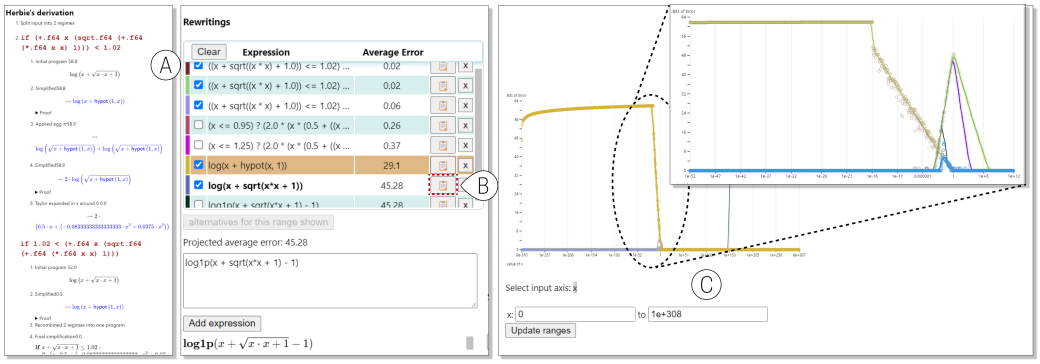


Figure 3.6: Tuning. The user can use derivations (A) to help them understand Herbie-generated rewritings. Each expression can be copied using the copy button (B) for easy editing of existing rewritings. The user can use the input range editor (C) to "zoom in" on critical ranges—i.e., resample and reanalyze all expressions on a new range. Above, the user has tried rounding some of an expression's constants after zooming.

To begin tuning this piece, Alex uses Odyssey's range adjustment control (Figure 3.6C). Since the conditional has a threshold at 1, Alex enters a range of inputs near 1: $10^{-52} \leq x \leq 10^{12}$. When Alex updates the range, Odyssey samples a new set of inputs all chosen from the selected range, and the plot updates to show only the new set of inputs. Because these inputs are all clustered near 1, Alex can now examine error in this range at much higher resolution. Here, the higher resolution reveals what inputs around 1 have a spike in error.

To fix this new-found problem, Alex continues to test new rewritings using the expression edit box. Since, at this point, Alex has already found many quite-accurate rewritings, they choose to modify an existing rewriting using the copy-to-clipboard button (Figure 3.6B). This allows Alex to easily make

small adjustments, such as raising or lowering the threshold by rewriting the branch condition, and see how that affects numerical error for the relevant parts of the input range. Alex may not always tune expressions for accuracy; but might instead simplify rewritings to make them run faster, or make modifications to improve readability and maintainability. In those cases, the error graph allows Alex to validate that error has not increased unacceptably. Finally, Alex has adjusted or *tuned* the expression to meet the application's needs, and then use the copy-to-clipboard button to copy the final expression and insert it into the program.

Reviewing these steps, Alex used a three-step floating-point error improvement workflow: diagnosing the sources of floating-point error; generating candidate solutions to these source of error; and then tuning and validating the resulting solution until it best met accuracy, performance, and maintainability goals. The entire process was orchestrated through Odyssey's table of rewritings and error plot, which track the various rewritings Alex already considered and allow Alex to easily compare rewritings over the input range. Odyssey additionally provided convenient ways to leverage the automated error-improvement tool Herbie, including invoking Herbie, visualizing internal heuristics, and presenting derivations. Combined, these features allow Alex to focus on higher-level concerns such as accuracy-improving rewrites and acceptable trade-offs between goals.

3.5 Iterative design process

To understand how to meet the needs of Herbie's users, after reviewing user-submitted bug reports, testing changes to the existing Herbie user interface, and mocking up a new interface, we conducted an iterative user design study. As we observed users working with the prototype, we identified new needs and added features to meet those needs.

User Design Study with Prototype

Our user design study consisted of nine interviews with participants ranging from floating-point novices to experts. Most participants were graduate students working on floating-point-related research with at least two years of experience. We spaced these interviews out and iteratively added features to Odyssey, responding to user concerns after each interview. We made the following observations.

First, we found that more experienced users iteratively submitted many hand-written programs to Odyssey. In some cases, users modified a Herbie result, used Odyssey's reported error to confirm that the change didn't harm accuracy, and then used the modified program as a base for further modifications. In other cases, users modified a Herbie result and re-ran Herbie on the modified expression, helping Herbie around a road-block of some kind and achieving a lower error as a result. We also saw users combining pieces of different programs into a single final program. Users described implicit

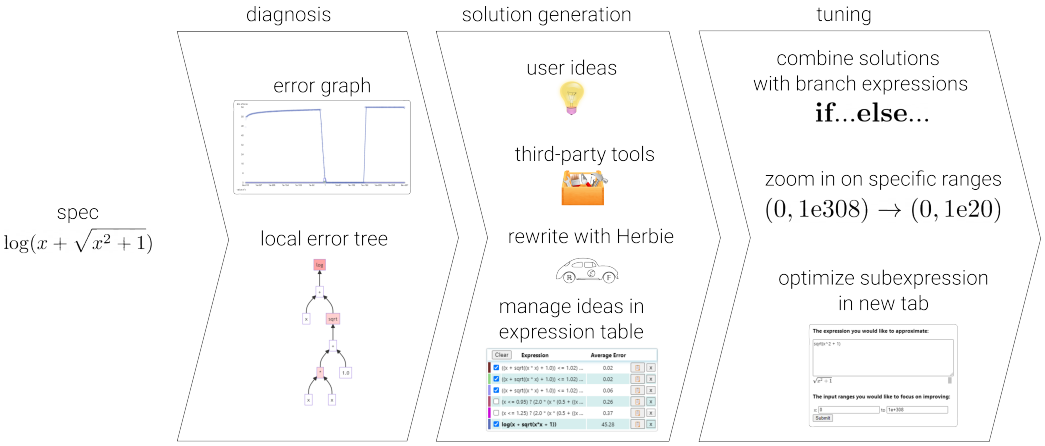


Figure 3.7: The general workflow supported by Odyssey. Odyssey starts with a real-number specification, analyzes sources of error, creates different solutions based on the analysis, and tunes solutions based on user's needs.

trade-offs, for example noting that Herbie's result was very complex, and that deleting certain terms from Herbie's result was less accurate but easier to read.

Second, we noticed that many participants, including both novices and experts, struggled to explain *why* there was error in an expression, even when they could see the error in Odyssey's error plot. For example, in the program $\log(x + \sqrt{x^2+1})$, most users could guess that the error for large x values was caused by overflow, but far fewer participants could identify that error for small x was caused by the log() operation.

In a follow-up conversation with the Herbie developers, we learned that Herbie used a metric called "local error" to identify which operations were likely sources of error. We decided that exposing this metric to the user as a local error "heatmap" (see Figure 3.3D) could help users better understand

floating-point error. Participants immediately began using per-point local error to explain why error occurred for specific inputs to specific programs. In this process, we also discovered that Herbie's local error implementation had a subtle bug on specific, rare inputs, leading to a patch.

Finally, after initially removing derivations (see Figure 3.6A), we realized that they were an important foundation for users' trust in Herbie's results. For example, one participant was surprised when Herbie recommended the expression "1.0" as an "improved" version of some much more complex expression and became skeptical of all of Herbie's other outputs, manually performing derivations to check that those expression had been computed correctly. Adding back support for derivations gave users more trust in Herbie's suggestions.

Through the user design study, we observed the following:

- Experienced users follow an iterative process when rewriting expressions.
- Rapid feedback during expression input helps users catch low-level mistakes.
- Users need help understanding what part of the expression is causing numerical error.
- Users want justification and explanation for the steps of automated tools.

3.6 Expression Rewriting Workflow and Design objectives

Our design study led us to model floating-point error improvement as a well-defined workflow consisting of three main stages: diagnosis, solution generation, and tuning.

First Stage: Diagnosing Problems

In this stage, users identify problematic operations within expressions, determine which problems are relevant to their objectives, and finding starting points for further analysis. For instance, in an expression like $\log(x + \sqrt{x^2 + 1})$, users must determine that the x^2 operation overflows for large values of x, while the logarithm is inaccurate for small values of x. The user then decides whether large values of x are relevant in their environment. If so, they focus on avoiding the overflow in x^2.

We developed two principles to support diagnosis. First, users need ways to focus analysis on the parts of the input range and expression they care about investigating—without losing track of the broader analysis. Second, even experts need tools to help determine which operations cause numerical error without relying on their expertise or resorting to trial-and-error operation replacement.

Second Stage: Generating Solutions

In the second stage, users gather potential rewritings from a variety of sources. The objective is to create a pool of rewritings that the user can evaluate and combine to address the problems identified in the first stage. While existing tools, like Herbie, are a valuable source of ideas and potential rewritings, the user must still track and organize the outputs. Moreover, rewriting ideas may come from many other places: other automated tools, papers, online references, and even the user's own creativity. Users need to collect the available rewritings, keep track of their origin, and organize them for easy evaluation.

We developed three principles to support solution generation. First, there must be a central repository of rewritings drawn from multiple sources. The repository must also store source-specific details, such as Herbie's derivations. Second, since users themselves are a major source of ideas, manual input of rewritings must be supported, with instantaneous feedback to provide low-level error checking. This supports a tight feedback loop and eases iterative exploration. Third, where possible, it should be possible to use user inputs as starting points for additional automated exploration, allowing users to overcome roadblocks faced by automated tools.

Third Stage: Tuning

In the third stage, users test, compare, and tweak rewritings to optimize for their accuracy, performance, and maintainability goals. Often the diagno-

sis and solution generation phases help users identify multiple independent problems and multiple independent rewritings that address them. Users must combine these rewritings to address error. This combination process is itself iterative. Users needing to validate that the combination did not introduce its own error. Moreover, the combination process might itself need tuning. Users may want to adjust the threshold at which they switch from one rewriting to another. Overall, this stage involves iterative refinement and experimentation until the user is satisfied with the result.

We developed two principles to support tuning. First, the user needs ways to compare rewritings for accuracy and get instantaneous feedback as they work. Second, users need explicit support for combining rewritings, whether directly using "if" conditions or indirectly by allowing the user to see multiple rewritings at once.

The order of these stages is not fixed, and users may iterate between them, but we think these principles address explicit user needs during floating-point error improvement with an automated tool.

3.7 Implementation

Odyssey is implemented in two pieces: a "backend" that uses Herbie to dispatch numerical tasks, and a "frontend" implemented using web technologies to present an interactive workbench UI to the user. Odyssey can be used via a web browser or embedded into tools like Visual Studio Code.

3.7.1 "Database Workbench" Architecture

The key to supporting our design principles is Odyssey's "database workbench" architecture. In this architecture, Odyssey stores a list of rewritings that the user is exploring and makes calls to independent analysis, visualization, and generation tools that run on the backend. This architecture stores all of the state on the frontend, allowing direct manipulation by the user. The automated analysis, visualization, and generation tools, meanwhile, are stateless, being invoked by Odyssey on whatever rewritings the user is currently considering. This architecture puts the user at the center at the center of the search.

This architecture also leads to a natural separation of concerns between the frontend and backend. The Odyssey frontend implements all interactions, graphics, and manipulation actions. However, all numerical tasks (sampling, evaluating error, and generating expressions) are the responsibility of the backend. This ensures proper support for low-level operations like enumerating floating-point numbers and other numerical tasks that depend on the user's target environment. While Odyssey currently only invokes Herbie subsystems, the backend is intended to invoke other tools as well.

3.7.2 The Odyssey Frontend

The Odyssey frontend provides a rewritings table and error plot to help users diagnose problems, generate solutions, and tune the results.

The main state is stored in the rewritings table, shown in Figure 3.5. All rewritings the user is considering—including both those generated by Herbie and those entered by the user, are stored here. Each rewriting also shows its average error, for easy comparison. A checkbox allows the user to hide expressions from the error plot and other parts of the UI, which functions as a kind of "archiving" operation so that users can ignore subpar rewritings without an irreversible deletion operation. Additionally, a clipboard button allows users to copy rewritings, which is essential to users modifying or combining rewritings. None of these interactions involve the backend, and are thus instantly responsive to user action.

The input box allows adding rewritings to the table using a natural mathematical syntax backed by a parser from the mathjs library [dJ13]. Odyssey then converts that input both to an instantly-updating LaTeX render (to help users catch mistakes and typos) and to the standard FPCore input format, which Herbie uses to represent rewritings. Herbie is then invoked to analyze the error of the new rewriting, which is then added to the plot. Additionally, rewritings can be added to the table by invoking Herbie to generate suggested rewritings; any rewritings suggested by Herbie are also converted from FPCore back to LaTeX and mathematical syntax so that the user does not have to understand FPCore in order to use Odyssey.

The main visualization is a large error plot. This plot shows the error on all of the sampled inputs, for each of the rewritings in the rewritings table, with colors helping users match each rewriting to its error plot. Because

rewritings often have identical error over some range, the user can click on a rewriting in the table to highlight it in the error plot; users can also use checkboxes in the table to hide expressions from the error plot. By hovering over each point in the error plot, the user can see the exact sampled input, and by clicking on a point, the user can update parts of the UI (such as the local error heatmap) to focus on that specific input. The user can also adjust the input domain using an input range selector below the plot. Changing the input domain causes Odyssey to resample inputs, evaluate each rewriting on the new inputs, and redraw the error plot using the newly-evaluated errors. Once again, besides adjusting the input range, all operations are instantaneous and do not invoke the backend.

On its own, Odyssey does not provide any additional features. However, Odyssey is extensible, and tools invoked by the backend can offer additional visualizations. To see these additional visualizations, the user selects a specific rewrite, and the visualizations are shown beneath the main UI. Selecting the specific rewriting means that different rewritings, which might come from different sources, can provide different kinds of justifications or explanations. Our Herbie backend provides two such visualizations: the local error heatmap and derivations. When Odyssey is extended to support additional backend tools, we expect each tool to provide its own additional visualizations.

3.7.3 The Herbie Backend

Odyssey's Herbie backend is used to sample inputs, evaluate the error of rewritings, and suggest new rewritings to the user. Herbie was originally designed as a batch-mode tool, so part of our work involved adding an HTTP API to expose various internal analysis functions so that they can be invoked by Odyssey. Luckily, the Herbie features that we wanted to expose, including input sampling and error evaluation, were already independently-invocable functions in Herbie.

A key challenge in the backend is dealing with latency. Herbie's initial design as a batch-mode tool means that Herbie typically samples inputs, evaluates error, and suggests rewritings every time it is invoked, even though some of those steps (like sampling inputs) are slow while others (like evaluating error) are fast. To address this, Odyssey's Herbie backend independently caches the outputs of each step (like the sampled inputs). This way, evaluating the error of an expression is done on cached sampled inputs and takes milliseconds instead of resampling the inputs, which would take seconds.

Further, all of Odyssey's invocations of the backend are asynchronous, allowing the user to continue working while Herbie processes their requests.

By keeping the latency of most operations under a second and offering access to previously-inaccessible heuristics like local error (an internal search heuristic) and expression derivations (previously a debugging tool for Herbie developers), Odyssey's "database workbench" architecture allows users to stay in the flow of their work as they solve rewriting problems.

Expert #	Background:
1	Industry, FP hardware + supercomputing (number systems for minimization problems), 45+ years.
2	Professor, FP tools (mixed-precision conversions and program analysis), 10+ years.
3	Grad student, FP hardware (datapath optimization), 5 years.
4	Professor, verification (correctness analysis), 9 years.
5	Industry, FP hardware (interval analysis, transcendental functions), 3 years.

Table 3.1: Five experts from the floating-point community evaluated and suggested future directions for our work.

3.8 Expert Evaluation

The goal of the expert evaluation was to assess the effectiveness of Odyssey in supporting the three-stage workflow we identified: diagnosing problems, generating solutions, and tuning expressions.

3.8.1 Protocol

We conducted an interview study with five experts from the floating-point community (see Table 3.1) to evaluate the effectiveness of Odyssey in supporting the three-stage workflow. We recruited the experts via email through professional networks and communities (e.g., FPBench). Each expert had different levels of experience in academia and industry, ranging from 3 years to

over 45 years, and their backgrounds covered various aspects of floating-point systems, including hardware design, verification, and optimization.

Each interview session was conducted over Zoom, with experts operating the tool via remote control to avoid early issues we experienced with participants on networks with special configurations. Interviews lasted between 60 and 90 minutes and consisted of three parts:

- *Introduction and tutorial.* I briefly introduced Odyssey and the problems it is designed to address. Then, each expert followed a hands-on tutorial demonstrating the usage of Odyssey on a simple example.

- *Seven tasks.* Each expert completed seven tasks, each designed for one of the three workflow stages and aimed at eliciting the experts' reactions to different parts of Odyssey's interface (Table 3.2). If the experts encountered difficulties, the first author provided guidance or reminded them of relevant interface features from the tutorial.

- *Exit survey and discussion.* To conclude, each expert completed a survey (Table 3.3) and participated in a semi-structured interview with the first author, where the experts reflected on their experience with Odyssey and provided feedback on potential improvements and extensions. I specifically asked for experts' opinions on the legitimacy of the workflow we aim to support, its relevance to their work, and the extent to which they felt Odyssey supported each part of the workflow.

Throughout all three parts, the experts' screens and audio were recorded.

The first author also took note of the experts' comments, insights, and responses to the tasks and survey questions. All study materials are provided as supplemental material.

3.8.2 Analysis and Results

We conducted an iterative, thematic analysis of expert solutions and the first author's notes for each stage of the workflow. Below, we discuss the experts' responses to the relevant tasks and survey items for each stage. Through this analysis, we aim to provide a qualitative evaluation of Odyssey's effectiveness in supporting each part of the workflow.

First Stage: Diagnosing Problems

Task 1 required experts to analyze the error in an inverse hyperbolic sine implementation and identify the parts of the expression causing errors, then decide which operation or operations needed to be rewritten in order for the rewriting to correctly handle large inputs. Among the five experts, four successfully completed this task, relying on Odyssey's error visualizations (see Figure 3.3).

P2 explored multiple input ranges in order to identify the two problematic operations:

> "*This is across the entire sample... so I wonder if it's doing something different on this side [clicking a point with a small x*

Task	Description	Targeted part of workflow	Success rate
1	$\log(x + \sqrt{x \cdot x + 1})$ is an expression for the inverse hyperbolic sine. Identify the parts of the expression causing errors for large/small x.	Diagnose troublesome subexpressions and problematic ranges.	4/5
2	Use Odyssey to find a solution for the troublesome square root subexpression from task 1.	Generate solutions for the subexpression and use these to optimize original expression.	4/4
3	Is your solution to task 2 good enough?	Use visualizations to form evaluation criteria for ending analysis.	4/4
4	Identify problems with branch expressions in fully automated solutions for task 1.	Explain important features of expressions and diagnose issues.	3/4
5	Use Odyssey to find and recommend log1p to solve small x.	Nudge an automated tool past roadblocks to generate better solutions.	2/3
6	Evaluate whether the full solution for the expression after tasks 1-5 is trustworthy.	Use Odyssey's feedback on expressions and information about expression soundness to evaluate expressions' trustworthiness and fitness based on personal standards.	2/2
7	Use branch conditions to outperform a fully automated rewriting for the expression $(\exp(x)-2)+\exp(-x)$.	Mix solutions from different sources and tune branch conditions to create stronger solutions.	3/4

Table 3.2: Experts worked through up to seven tasks to exercise the features of Odyssey before a survey-based discussion. Due to time constraints, not all experts completed all tasks.

> *value and looking at the local error graph] So there it's all the log, and over there... [clicks a large x value] it's all the square root. So that's interesting, it's actually coming from different operations.*"

Here, the error plot effectively surfaced the two areas of high error (small and large x values), giving the expert clear places to look for troublesome operations. Then, by switching between inputs in different regions, the expert was able to see that the problematic operation was different between these regions.

In the survey (see item 3 in Table 3.3), all five experts rated the interface's ability to help identify or confirm specific problems with expressions at a 7 out of 7. We attribute this success mainly to the error plot and local error heatmap, which implemented the second principle we identified for a good diagnosis tool. They supported the user in assigning responsibility for error without relying on expertise or resorting to trial and error. As P2 concluded,

> "*Having the graph and being able to click on the different places where error is high is definitely nicer than just looking at output in a text file.*"

Second Stage: Generating Solutions

We designed several tasks to evaluate Odyssey's support for collecting and evaluating new expressions that address the identified problems in floating-

point expressions. Close to all experts who attempted each task succeeded (see Tasks 2 and 5 in Table 3.2).

Task 2 required experts to analyze a troublesome subexpression from Task 1 and find a better rewriting for it. Then, experts needed to bring the solution back to the original analysis and decide if they were happy with it. Four of the five experts who attempted Task 2 successfully completed it, showing that the interface facilitated the collection of solutions and their integration into existing expressions. Of those four, two experts found their own unique approaches to solving the problem identified in Task 1 rather than relying on an automated solution. One expert pulled a factor of x out of the square root, and another expert created a branch that switched to an approximation for large values of x. Both of these approaches showed low error on the error plot, though the experts noted there could be issues with these choices (for example, branching impacts performance, and dividing by x is risky when x could be 0). This showcases the flexibility of Odyssey in allowing users to explore alternative solutions and evaluate their impact on the error plot. (The expert who did not complete Task 2 was our first participant, with whom we lost much of the interview time due to the networking issues mentioned earlier.)

Similarly, Task 5 asked experts to find a more accurate rewriting for a subexpression applicable to small values of x. Three out of the four experts successfully completed this task, further supporting the effectiveness of Odyssey in assisting experts in gathering and evaluating potential solutions.

P3 had the following to say about working through the process up to Task 5:

> "*It feels like quite a natural way you might approach this problem as a human. You're burrowing down into it more precisely and pushing your error around a little bit. I thought the transition of 'we've moved the error from the log into the subtract [using log1p], now I know how to deal with the error in a subtract as well' felt natural, ... since ... once we figure out it was going to be the subtract that was giving us trouble, then [we can use Herbie to rewrite successfully]. It gets there much faster, but it's cool that I also feel that I would have thought about going in a similar direction.*"

In the survey, experts rated the interface's ability to generate ideas for solving specific problems (item 4) with scores ranging from 5 to 7, with an average of 5.8. The interface's effectiveness in evaluating the quality of ideas quickly (item 5) was rated between 5 and 7, with an average of 6.4. These relatively high ratings indicate that the experts found Odyssey helpful in generating and evaluating ideas for improving floating-point expressions.

Users were able to use Odyssey to successfully generate a variety of valid nontrivial new expressions for analysis, both using an automated tool (e.g. the way we expected users to solve Task 2) and by themselves (P5 and P4). This was significantly different from our experience in the earliest parts of the design process. The ability to send rewrites back to Herbie was a vital

part of the solution generation process for the three experts who were able to complete Task 5.

Third Stage: Tuning

The third stage of our proposed workflow involves tuning expressions to further optimize their accuracy and performance. To assess Odyssey's support for this stage, we evaluated Task 7, as well as survey items 6 and 7, which inquired about the interface's support for comparing and mixing different expressions.

Task 7 challenged experts to create a more accurate expression than Herbie's best alternative for a given expression by combining different solutions and fine-tuning the branch point. The task demonstrated that a human can use Odyssey to outperform Herbie's internal heuristics when unique requirements call for a tailored approach. After using the range zoom feature and noticing Herbie's solution was still outperforming their solution on a small region, P2 remarked, "*So in this view, we can see that we don't have quite the right number [for the branch point].*" The expert then adjusted the branch point based on the visual feedback.

In the survey, experts rated the interface's capacity to help them mix expressions from different sources (item 7) with scores ranging from 4 to 7, with an average of 5.4. The interface's support for comparing different expressions (item 6) was rated even more highly, at an average of 6.4 (range from 6 to 7).

#	Survey Questions:	Results:	Average:
1	"The workflow made sense to me and I was able to follow it."	5, 5, 6, 7, 7	6/7
2	"This workflow matches my experience approaching real numerical analysis problems."	4, 6, 6, 6, 6	5.6/7
3	"The interface helped me identify or confirm specific problems with expressions."	7, 7, 7, 7, 7	7/7
4	"The interface allowed me to generate ideas for solving a specific problem."	5, 5, 6, 6, 7	5.8/7
5	"The interface let me evaluate the quality of ideas for rewritings quickly."	5, 6, 7, 7, 7	6.4/7
6	"It was easy to compare expressions in the interface."	6, 6, 6, 7, 7	6.4/7
7	"It was easy to mix together expressions from different sources in the interface."	4, 5, 5, 6, 7	5.4/7
8	"The interface let me focus on thinking about the problem at a high level."	5, 6, 7, 7, 7	6.4/7
9	"I can think of ways to extend this workflow + interface to address numerical analysis problems that I have worked on."	5, 6, 7, 7, 7	6.4/7

Table 3.3: After completing the seven tasks, experts were asked to evaluate different aspects of the tool on a scale of 1 to 7.

As we can see in the example above, the especially high rating for comparison was likely a result of combining the ability to plot the error for different expressions together with zooming to focus on getting feedback on specific regions. A couple experts (P4, P5) mentioned wanting more support for combining expressions, especially around conditional branches. P4 explained that an automated tool might be able to add guard conditions where appropriate.

Finally, the experts appreciated the potential power of mixing human and automated solutions, with P3 commenting that suggesting `log1p` and `hypot` to Herbie felt similar to proof assistant tools where "*if you just add in an additional step on the way or an additional lemma... then it can actually nudge it over that threshold.*"

In summary, the results from Task 7, along with the survey responses for items 6 and 7, provide evidence that Odyssey effectively supports tuning expressions for optimal accuracy and performance. The interface enables users to mix expressions and adjust coefficients while offering real-time feedback, streamlining the tuning process and enhancing the overall quality of floating-point expressions.

3.9 Discussion

As the first expression rewriting workbench for the numerics community, Odyssey demonstrates how to build useful expert tools that enable users to

more effectively search a design space. Below, we discuss three insights that were key to Odyssey's design. These insights serve as design principles that generalize to expert tools in other domains where users want to navigate a design space.

Expose heuristics, not states: First, we found that exposing the internal exploration-focusing heuristics of the tool, rather than just the search states—for Herbie, mainly the local error—helped users significantly, beyond its use in Herbie alone. By connecting this heuristic to other simpler metrics (like the input error plot), users developed *explanations* of the heuristic's value that helped them understand what was relevant about the search state—for expression search, what subexpression was probably causing the error. By comparing the heuristic and their explanation across expressions, users could check if the issue was solved, even if the expression shape was too complicated for an automated tool to recognize.

Give access to intermediate representations: Second, we found that giving the user ownership over intermediate parts of the search made the tool much more useful. Doing so even allowed us to catch a bug in the underlying tool. A widely held belief among the HCI community is that higher levels of abstraction are more desirable for end-users. Therefore, in an automated expert tool, it can seem natural to hide the middle of a search from the user to keep them working at a high level. However, in our study, we found that users wanted to be able to see and control the search process. Experts

were particularly eager to introduce their own ideas and test assumptions. In Odyssey, without building any separate tooling except for a table that tracks candidates and synchronizes with visualizations of existing automated analyses, users are able to explore a much broader space of possible solutions in a way that was not possible with the original tool, simply by letting a human manage search candidates.

Test expert workflows with relative novices: Finally, we were able to identify the appropriate level of abstraction in Odyssey because of our own iterative design process that involved novices and experts. Involving novices sensitized us to the foundational cognitive burdens experts had developed workarounds for. We realized that if our tool could not help a novice at least understand basic issues, it was likely too opaque for experts to use productively. The local error plot, a key feature we would not have included without involving novices, ended up being the most praised feature by experts.

Applications to other domains with user-driven design space search: While this paper focuses on floating-point analysis, the above key insights and findings suggest generalizable principles for user-driven design space search. The tool wrapped by Odyssey, Herbie, works in a way that should be familiar to anyone who has worked with a design space exploration tool or classical AI search: it identifies a troublesome part of an expression, applies algebraic rewrites or approximations to that part of the expression to obtain new expressions, tests those expressions to see if they are worth exploring,

and finally merges the best options.

The shape of this process matches the workflow we describe for an analyst identifying and solving problems with an expression step-by-step while tracking possible rewriting directions. This search shape is used in tools across many domains, including in automated theorem provers, carpentry compilers, machining systems, and ASIC design space exploration tools. Yet, expert tools in these domains do not apply the above three principles. As a result, the tools remain difficult to use and error-prone. We hypothesize that applying the principles will improve expert tools in other domains where users search a design space.

3.9.1 Limitations and Future Work

A major limitation of our design process was the tight design loop we had to maintain during development. While this was necessary to ensure we were building a system that would be useful to users, this meant we had to compromise on the polish of some features and altogether avoid others which would take too long to implement or require disturbing many parts of the interface. With more time, we would like to further improve the interface's layout and provide more structured expression editing support.

Note that the main future work for Odyssey is to incorporate more analyses and sources of rewritings, including ideas like operation cost analyses and hardware-specific rewrites that were mentioned by the experts in our study. Tools like PRECiSA [TFMM18] that already have an HTML-based analysis

interface may be a good starting point for testing these integrations.

Floating-point experts were very appreciative of our work, and saw a variety of ways it could be extended to further support their particular areas of expertise. These included ideas like adding support for multi-precision rewritings, incorporating operation cost analyses from Herbie and other tools, adding ways of helping human users simultaneously optimize at least 3 variables, and increasing support for splitting expressions into subregions and subexpressions based on domain-specific heuristics.

Odyssey also has clear potential application in floating-point education. Several of our tasks asked users to explain to the interviewer potential problems with an expression using the interface, and both the experts and the novices in our formative study were able to point out areas of high error, select points, and zoom in to get a better look at problem regions to support diagnostic claims. Odyssey has the potential to thrive in a classroom setting; it could be used by an instructor to show off how expression rewriting makes expressions more accurate or by students to explore and diagnose error sources an expression and try fixing them. We plan to try applying Odyssey in an undergraduate class covering floating-point representations soon.

We are also excited by the explanatory potential offered by the incorporation of large language models (LLMs) like GPT. We have found that available language models can, in fact, offer rewritings and generate plausible explanations for users, but they are prone to "hallucinating" and incorporating nonsensical logic, so their output must be validated before it is used. With

access to Odyssey's calculation and validation tools, an LLM might be able to avoid these issues.

Finally, a major possible extension was brought up independently by two different participants, who commented that they would be very interested in plugging in additional visualizations showing actual output effects of errors for each expression. For example, one participant has worked with expressions representing geometrical ellipses, and wanted to see how different kinds of error could lead to distortion of the ellipses. Allowing for additional visualizations would be a major possible improvement, since it will help users understand whether the error they see on the error plot matters when code is compiled and run in practice. If (as with ellipses) the output space can be mapped back to specific input values, combining output visualization with the error graph heatmap will let experts relate points with noticeable error in the actual output to the particular mathematical operation causing that error.

Overall, we are excited to see what floating-point experts and novices end up doing with Odyssey and look forward to improving our support for their work in the future.

Chapter 4

Magic Markup

This chapter is adapted from my paper:

[MTT24] Edward Misback, Zachary Tatlock, and Steven L. Tanimoto. "Magic Markup: Maintaining Document-External Markup with an LLM." In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming*, pp. 22-31. 2024.

4.1 Overview

After a developer has worked through an analysis with a tool like Odyssey, is it possible to keep that analysis with the code it describes? This chapter addresses this fundamental challenge: how to persistently attach contextual information to documents as they evolve. Traditional anchoring methods that rely on string matching or document structure fail when documents un-

dergo significant changes. Magic Markup introduces a semantic anchoring approach that leverages large language models (LLMs) to understand what a document section means rather than just how it appears. Through benchmark development and evaluation, this chapter demonstrates that LLMs can effectively maintain markup positions across document changes that would break traditional anchoring methods. This capability is essential for implementing any system that seeks to maintain external context for evolving code, setting the stage for the more comprehensive system described in Chapter 5.

4.2 Introduction

4.2.1 Problem Statement

Document markup allows a number of powerful behaviors related to *tagging* text with metadata. As an example of the impact of markup, Hypertext Markup Language (HTML) is famously the backbone of the World Wide Web, which is formed by hyperlink annotations that tag some piece of text in one document, called *anchor text*, with a link to a related document[BLC90]. A curious reader might wonder: has markup led to changes of a similar scale in the realm of software engineering?

In software engineering, code comments are a simple way of marking a document with miscellaneous helpful information. More extended systems for attaching data to code also exist–systems in the domain of *literate programming* integrate information about code with the text of the code itself,

and rely on markup to accomplish this[Knu84]. Perhaps the JSON files backing Jupyter notebooks can also be considered a form of markup document, if the code that runs is taken as "the document."

However, if we think of markup as a basic primitive for working with and referring to text, and think further about the universality of text as an interface in programming, we should be surprised to find that this primitive is almost universally unsupported, even in advanced live programming environments. For code in particular, there is no standard for attaching metadata like code review history and example data to a particular point in the text. We attempt to explain why below.

A principal challenge in designing a document markup system is maintaining the correct positions of text tags when the underlying content that the markup refers to is edited. This is called *annotation anchoring*. The simplest solution to this problem is to include the tags in the document text itself, as in HTML or standard code comments. This requires no special programming tools, but it burdens the document's reader (whether human or computer) with distinguishing content from metadata, so it isn't suitable for documents with many layers of extensive annotation–imagine a line of code with comments left by 10 different people for completely different purposes. The second-simplest solution to this problem is to manage document edits through a special program like a *"What You See Is What You Get" (WYSIWYG) editor* that shows only the document content (with the effects of markup metadata) while managing the positions of tags behind the scenes.

The Microsoft Word document system is an example of this second solution, and also shows its downsides: the cost of building and maintaining a special editor that acts the way a human expects is significant, and it also locks the programmer into always using a particular editor. Very few editors can operate on Microsoft Word documents.

This paper advances a third solution to the problem. Prior authors have considered the idea of attaching *external annotations* to programming systems with *text anchoring*–methods that relate metadata kept in a separate file to a point in the code using text similarity. These systems are useful, but fail after the text has changed enough (for example, when the names of variables in a program have changed, or when a loop has been vectorized), even when a human could still find a reasonable new position for the annotation using their understanding of the text's syntax and semantics. As such, serious infrastructure to support ubiquitous external annotations has not been feasible.

This third solution is the only fully general solution for programming systems and other systems with restrictions on the structure of text in the file. For example, many configuration files are stored as JSON, which simply does not support comments at all in its formal standard. Further, all files presented through a plain text editor have implicit restrictions based on what can reasonably fit in the editor's viewable buffer.

To address this issue, we propose *magic markup*: markup maintained separately from the document by a semantics-aware system that "magically"

handles re-tagging after document updates. What would it mean to be able to keep markup off of a document, and what would it mean to be able to mark up code?

4.2.2 User Story

To illustrate how we imagine external annotations being used, we present a user story involving two programmers.

Barbara is a senior data scientist who has received an informal code review request from Alex, a junior developer in the same company. Alex has updated the code of a particular function in the production code base responsible for an image classification task. This code includes an image transformation pipeline with subparts that are known to be performance-intensive.

Barbara begins by pulling Alex's changes and opening the updated file in her editor. The code review request appears in Barbara's editor as a set of annotations next to the file. Alex intends these annotations as review requests for Barbara only. These are *user-directed comments* stored in a separate database (that was pulled with the code) and attached to the document via an external annotation system. As they are not part of the file, unlike normal code comments, they can be hidden by default or even locked for everyone but Barbara. As Barbara begins to edit the document to address issues she sees in the changes, the comments remain attached to the entities she would expect. Even when she vectorizes a loop that Alex mentioned he was uncertain about, completely changing its text, the document's tag

maintainer—an intelligent agent backed by a language model—knows the vectorized code is intended to replace the loop, and the comment remains in the right place until she marks the concern as completed. She knows Alex will easily locate her update through the resolved comment, even though she has also moved the vectorized code into a separate function.

She notices that Alex added a new nested lambda function that introduces additional image processing. She isn't familiar with the method Alex used, but fortunately, Alex annotated this part of the pipeline with some *example data*, and she looks at the cached output image *visualization* annotation for a moment before it updates with the new output from the *out-of-context execution* of her own system on just the lambda function with Alex's example data. Barbara remembers the days when she would have had to copy this code out into a REPL or a notebook and synthesize example data herself just to check its behavior and throw it all away afterward. The behavior of this section is still a little unclear for quick reading, though, so she selects the section and asks a language model to generate a short clear *dynamic explanation* of its function that will continue to apply as the code base evolves.

Barbara looks at the next part of the new code–a loop body with a performance concern. She decides that she will have to check on this part herself, so she selects the section and asks her editor to run just that section with the output from Alex's section and time its performance across 5 executions. The performance is not as good as it should be, and she realizes Alex's output image is unnecessarily high-resolution. She quickly fixes this, and in response,

the output image updates and the execution time drops. She asks her editor to *warn* the programmer if this example execution time ever goes above 20 milliseconds, as that probably would have helped Alex. This annotation automatically becomes part of the document's *performance overlay*, which is different from the *presentation overlay* she uses when walking new developers through the code. Again, she remembers how troublesome it would have previously been to unit test this loop body.

Barbara also notices an edit to a data structure that she realizes it would be best to mark as off-limits to the junior developers on the team. She adds a note that all of the junior developers will see when opening that section of the code.

Alex's change introduces a new option for the image processor, and Barbara follows a new *documentation link* on the option to the relevant documentation section Alex added for this. The documentation links back to particular code blocks Alex added when referencing implementation details.

Having a last quick look through the file, Barbara finally fixes the first minor issue she noticed–there was a typo that broke parsing at the top of the file. *This didn't matter for any of her other interactions with the file*, since it wasn't in the annotated sections and didn't affect the segment-specific executions.

All of Barbara's notes and tests from the review can be consulted by any team member later, with the new annotations laid out on a timeline that reconstructs Barbara's thought process as she worked with the code.

Even if someone edits the file in an unsupported editor, Barbara's team trusts the tag maintainer to correctly re-tag the document afterward without any issues, flagging any seriously ambiguous re-taggings for their review. The platform-independence of the tag system lets them forget about the tags when they don't need the extra information, and even lets them maintain annotations on the source code of an independent code base for one of their dependencies that they only have read access for.

Barbara's rapid, high-level code review is the product of tools whose foundation is a highly reliable tag maintenance system with the following properties:

- Annotation anchors are updated each time the code is modified.
- Annotations are stored separately from the base document to avoid breaking editing and execution for typical text editors and program interpreters and to keep the annotations of different tools independent.

4.2.3 Contributions

The above vision leads us to seek answers to the following questions:

1. How capable are current language models as "tag maintainers" for a document? Are they reliable, fast, and cheap enough to build on top of?
2. What kinds of documents or edits make tags hard to maintain? For source code in particular, what kinds of edits exist at the semantic

level, and in what cases does simply moving tags or noting they have been "orphaned" fail to capture that meaning?

To answer these questions, we construct an LLM-based re-tagging system. To evaluate our re-tagging system and promote further progress on this problem, we synthesize a benchmark suite representing 90 code updates across 5 programming languages in which a tagged entity is relocated or altered. We also provide the code for benchmark generation.

Our contributions include the following:

1. a formal vocabulary for the problem space that introduces the notion of annotation intent

2. adaptable code for generating empirical test and training data for the re-tagging task

3. the synthetic Tagged Code Updates benchmark dataset, generated with the above and cleaned

4. an LLM-based re-tagging system

5. an evaluation of our system's performance on the benchmarks using OpenAI's GPT-4 Turbo[1] model

[1] gpt-4-0125-preview

4.3 Basic Definitions

While the remainder of this paper presents an early exploration into the power of LLMs to update annotations in the context of evolving code bases, this and future work can benefit from a clarification of the terms and concepts involved in this research. This section both addresses this need and suggests a longer-term trajectory of work that takes account of explicit notions of the intent of annotations while maintaining them.

Due to the wide variety of uses for annotations in documents and the implications of usage contexts for automatic maintenance of annotations during document editing, we propose terminology to clarify some of the otherwise ambiguous notions on this topic. We start with the simplest concept of "text point" and work through "annotation" and finally "mapped annotation".

A *text point* TP is the character index (an integer, zero-indexed) used to designate a position in some (any) text string. The text point is independent of any text, except to the extent that the text be long enough to have position corresponding to the text point. For example, the text point 4 refers to the position of "C" in "ABRACADABRA" and to the position of the second "d" in "Aladdin" but does not refer to anything meaningful in "Fun". We'll say that text point 4 is compatible with "ABRACADABRA" and "Aladdin" but incompatible with "Fun".

A *text segment* S of a document D is a part of D specified by a starting text point $TPstart$ and an ending text point $TPend$, where both text points

are compatible with D. The substring of D that starts at $TPStart$ and ends right before $TPend$ is considered part of S. Thus S = [1, 4, "lad"] is a text segment of "Aladdin" but not of "ABRACADABRA" and not even of "The boy Aladdin".

An *annotation* A of a document D consists of a text segment S together with two additional pieces of information:
(i) contents. We can assume this is text or hypermedia represented textually (e.g., with HTML).
(ii) intent. Though often unknown or unspecified, this is a kind of metadata associated with the contents and the text segment that can be important in the accurate maintenance of the annotation as the underlying document D goes through edits or other transformations.

The text segment S of annotation A is known as its *anchor* [BBGC01]. The substring of D in S is known as the *anchor text* of A. If the substring is of length 0, then the anchor is called a *point anchor*. Otherwise, it is called a *range anchor*.

A *document view* V consists of a document D and a set Z of annotations.

Given a document view $V = (D, Z)$ and a transformed (e.g., edited) version of the document D', the view-mapping problem is to update Z to obtain that Z' which best respects the intents of annotations in Z. We call Z' the mapped annotations from V, and V' is the mapped view of V.

As an example, let D = "boy alad." Let $Z = \{A_1\}$ where A_1=([4 , 8, "alad"], content:"The boy's name", intent:'TRACK NAMES'). Further let D'

= "The young boy Aladdin wandered out." We may expect that $Z' = \{A'_1\}$ where $A'_1 = ([14, 21, \text{"Aladdin"}]$, content:"The boy's name", intent:'TRACK NAMES').

The incorporation of intent as a component of an annotation is not customary in computer technology at this time. For example, highlighting tools in editors do not require indication of intent or directly offer any specific affordance for expressing the intent of an annotation. Microsoft Word and Powerpoint do offer alternatives to highlighting: strikeout, underline, change of font, font size, or color, etc. But these do not enforce any clear intent communication either.

However, intent is actually a fundamental aspect of annotations as defined outside of the computing-tool context. Here is the Merriam-Webster definition:

> "annotation: (noun) a note added by way of comment or explanation."

To unpack this definition, we can identify three essential aspects of an annotation. First, the "note" aspect is its content, some text or similar representation of the annotator's idea. The word "added" indicates that the note is a new component of something existing – that would be the base document or the base document already joined with other annotations. Finally, there is something about the purpose of this added note: "by way of comment or explanation". The phrase "by way of" can be interpreted here as "based on"

or "associated with", and the objects that might be associated are "comment" and "explanation". These can be considered purposes, motivations, or reasons for the annotation. They are communicative modes or roles for the annotation. They are examples of what we are naming, in this paper, "intents".

Intents of annotations are very important inputs to any process of maintaining annotations when the underlying document for it changes. Without any explicit information about intents, any algorithm will have to guess or embody a programmer's guess about intent of annotations in a given application. A typical yellow-marker highlight in a text is an annotation with a range anchor and anchor text, but no content. A good guess at the intent in such an annotation is to express "I think this anchor text is relatively important in this document." This may be enough to guide a smart annotation mapper to a good result, because it implies that there is some semantic integrity of the anchor text that should be preserved in the mapped annotation, and the location (of the anchor text segment) should be updated in such as way as to maintain the same relationship between the segment and the surrounding context before and after the transformations.

A very different intent of an annotation is the marking of line numbers, which although not present in the original document, can support an editorial process or programmer interaction with a debugger. When the program is edited, we do not expect that the line numbers will follow the syntactic or lexical contexts in which they originally occurred, but will reflect the new line

Figure 4.1: An example synthetic benchmark. On the left, a language model has produced an original program for displaying the price of drinks, and another model has selected and delimited a segment (the "menuItems" constant) with black Unicode star characters (⋆). On the right, a language model has synthesized updates to the program while keeping the segment in place. Our re-tagging system predicts the position of the segment in an *unmarked* version of the file on the right.

structure post-editing. Another example of an annotation that should not be updated according to surrounding context is a point-anchored annotation with content "5000 words to here." Without either explicit intent or correct inference or guessing of intent, an automatic annotation mapper would be at a loss to do the best thing.

4.4 Tagged Code Updates Benchmark Suite

To define the targets for a re-tagging system empirically, benchmarks are needed. While real-world codebases like the Java files tracked by Reiss et al. [Rei08] should be the gold standard for such benchmarks, language models

present an opportunity to rapidly construct synthetic benchmarks with significant detail about the intent behind refactorings. We create the Tagged Code Updates benchmark suite as an example.

Our benchmark suite and generation system are available on Observable and Github.

4.4.1 Code Generation System

Our system generates examples of code that has undergone a single edit or refactoring (Figure 4.1). Examples are generated through a series of queries to language models of varying capability. We note that steps involving creative generation can be performed by any moderately creative and attentive model, but we required one of the largest available models (GPT-4 Turbo as of January 25, 2024) to reliably perform steps involving correctness.

For each example, we begin with this prompt:

> Briefly describe an intermediate-level $LANGUAGE programming problem including at least one $SNIPPET_TYPE that can be solved in a single file. Use a creative, real-world framing. Describe steps to solve this problem. Do not provide code yet.

This step creates a high-level frame for the code that will be produced. A full example output can be seen in the appendix (section 4.9.1).

Next, we generate initial code to solve the described problem. We found that this typically results in small, easily-understood programs that might

be used as applications examples in beginner and end-user programming contexts, like a simple calculator for a pizza restaurant or a grocery store inventory manager; perhaps the most interesting program we saw was a genetic algorithm for optimizing the delivery routes of an e-commerce driver (benchmark 64).

After this, a short delimiter string that the model will not confuse with the text already in the file is chosen to delineate the sections of code that are being annotated. We hardcoded this value as a Unicode star character (U+2605, ⋆) for our benchmarks, since it did not occur in any of the generated code. An extended system might have to dynamically handle a variety of possible character sets.

At this point, the model is asked to describe a snippet in the program matching the SNIPPET_TYPE. This again creates a high-level frame for the next step, where the model is asked to rewrite the code with the described snippet delimited using the chosen delimiter string. Adding the delimiters was the first time we found it helpful to use the largest available model, in order to make sure the language of the snippet description and the actually delimited segment matched our expectation as programmers as closely as possible. (In retrospect, we find the snippet descriptions in our benchmarks still vague. Generating the snippet description itself with a larger model and more detailed instructions about connecting the snippet with the code would probably create a closer match.) An example of a generated snippet can be seen in the appendix (section 4.9.1).

Next, we ask the model to describe "an interesting change or refactoring of this code that a real-world programmer might apply." We provide a flag that optionally asks the model to "[d]escribe a state where this code change has only been partially applied" to try to obtain more realistic examples of incomplete edits or refactorings. An example of an update description can be seen in the appendix (section 4.9.1).

Finally, we generate the updated code, specifying that the delimiter position must be preserved, with their contents "functionally identical in the new version of the code." Figure 4.1 shows an example output benchmark; see the appendix for the full prompt (section 4.9.1).

4.4.2 Benchmark Suite Description

Parameters

We initially generated 101 program/program update pairs for our benchmarks. The suite targets 5 languages: Python, Javascript, JSX, Racket, and C. We wondered if different languages and syntaxes might present different levels of difficulty for the model's re-tagging attempts.

We asked the model to choose from 6 kinds of snippets: constants, subexpressions, variable assignments, loop bodies or code blocks, loop conditions, and function calls. These represent a slightly-diverse sample of common simple code structures, but leave out many other possibilities, like selecting keywords, parts of comments, or multiline sections of code.

We also generated an additional 10 examples for a "training set" used for prompt tuning, described in section 4.5.1.

Benchmarks filtered out of the test set

After generation, we manually reviewed all benchmarks and excluded 11 from the final set. 8 of these were excluded for generation mistakes like missing delimiters in the initial or output code.[2] We also found 2 other kinds of outputs:

1. The segment is completely missing in the updated code. For example, a loop condition is gone after the loop has been refactored into a reduce() call. (2 benchmarks)

2. Multiple independent segments could be delimited due to ambiguity after code duplication (1 benchmark).

2 and 3 represent more complicated cases that we decided were out of scope for this benchmark suite. Other benchmarks designed to target those particular issues are needed to explore them properly.

This left us with a test set of 90 examples, with 28 written in Python, 17 in Javascript, 17 in Racket, 16 in JSX, and 12 in C. The types for the targeted snippets were as follows: constants (17), subexpressions (19), variable assignments (12), loop bodies or code blocks (12), loop conditions (13), and function calls (17).

[2]We did later test those with proper initial delimiters separately from our evaluation below to see if our system had any problem with re-tagging these examples, but it did not.

Benchmark suite generation costs

Each benchmark required around 1800 input tokens and 700 output tokens from a "large" model like GPT-4 and 1300 input tokens and 1000 output tokens from a "small" model like GPT-3.5. This cost us roughly $.04 per benchmark through OpenAI's platform. Generation time for each benchmark was on the order of tens of seconds due to the relatively large number of output tokens required from a large model.

4.5 Prototype Re-tagging System

We created a re-tagging system[3] to measure the capability of current language models on the Tagged Code Updates benchmark suite. Our system submits a single prompt to obtain the text and line numbers in the file of the updated segment, then matches the text points for the beginning and ending of the segment in the updated file.

4.5.1 Re-tagging Prompt

An example of an application of our re-tagging prompt template can be seen in the appendix (section 4.9.2). To allow the model to reference line numbers reliably, we prepend the line numbers for the original and updated files. In case the tag's contents are repeated in the section indicated by the line numbers, for example in a single line like $a = \star a \star + a$, we also request the

[3] Available on Github and Observable.

index of the correct match's occurrence in the section. We use the OpenAI JSON response format to constrain the model to only generate valid JSON in its response.

Prompt hand-tuning

Before settling on this prompt for our evaluation, we used the 10 examples in our training set to check variations, including variations that were more successful for less powerful models (gpt-3.5-turbo-0125, Gemini Pro, and Mistral8x7b).[4] These variations included:

- Prompts that simply generate the full re-tagged file. These prompts were initially promising, especially when run on a large model, but we found that the output was not faithful enough to the text of the updated file to trust. For example, the output might be missing comments, or have whitespace differences, or a single typo somewhere in a long file. Furthermore, the time required for a large model to copy an entire file is significant.

- Prompts that generate plain English responses. This sometimes helped smaller models in our experience. However, parsing the generated output introduces an additional point of failure.

- Prompts that first ask the model to focus on a smaller section of the file. This significantly helped smaller models in our experience. In

[4]Although it was trained on code in particular, CodeLlama-70b-Instruct-hf did not follow instructions well enough to be considered.

particular, asking the model to reprint the general section of the file it was focusing on seemed to help it locate finer-grained segments of that section.

- Prompts that ask the model to reevaluate previous answers. This did not seem to help smaller models when the previous answer was "known" to be the model's own output. A proper setup might ask the same prompt several times and try to average or take the best idea across the results.

- Prompts that ask the model for a confidence rating or ask the model to "put yourself in another programmer's shoes" and think about whether someone else might answer differently. This did not seem to have any impact on results, but we noticed that smaller models seemed to express the same levels of confidence (around 90%) in correct and incorrect or ambiguously correct decisions. gpt-4-turbo-0125 expressed nearly complete confidence in both its correct and ambiguously correct answers, but usually correctly identified alternatives to ambiguously correct answers. (An example of an ambiguously correct answer is re-tagging $a = \star b \star$ as $a = \star b + c \star$, since $a = \star b \star + c$ is arguably also valid.)

Even on our very small tuning set, the smaller models occasionally made serious mistakes, like moving an annotation on a function call to the function definition. They also struggled more with keeping the contents of annotations consistent at the character level: an annotation like "for ⋆line in lines:⋆"

was likely to become "for ⋆line in lines⋆:", dropping the colon. Since breaking the problem down into smaller steps helped but did not fully eliminate these issues, there may be a fundamental difficulty for these models with the complexity and multi-step nature of the task.

The model used for our evaluation (gpt-4-turbo-0125) reliably obtained a perfect score on the tuning set with the selected prompt.

4.5.2 Text Point Matching

After obtaining the text and line numbers of the updated segment, our system attempts a whitespace-normalized exact match with the text in that subsection of the updated file. This naive approach gave us a perfect score on the tuning set with outputs from the model used for our evaluation. For the outputs of smaller models, we attempted to configure fuzzy matching, but did not pursue this in the system evaluated. Asking for a regular expression matching the section or the start or end of the section also did not reliably result in a correct match.

We discuss possible failures of this naive setup in section 4.6.1.

4.6 Evaluation

We ran the re-tagging system on the benchmarks using gpt-4-turbo-0125 as the language model and report the results here.

4.6.1 Results

Accuracy and points of failure

79 of the 90 tags in our test set (˜88%) were placed with no difference at all from the "correct" output.

Two of the differences were incorrect matches resulting from tags beginning with whitespace, which our system was not designed to preserve. Ignoring this issue gives a "true" accuracy of 90%.

Two more differences were incorrect matches due to the model stating the wrong occurence index. In our test, the model never stated any occurrence index other than 1. This seems to indicate a lack of understanding of this part of the prompt.

No match was found for seven of the benchmarks. In five of the match failures, the text was correctly identified, but the model misidentified the starting or ending line of the segment. All of these differences were off by one in the direction of starting or terminating the section of the segment early. Five of these (5, 77, 80, 82; not 20) involved what appears to be an issue with mismatched nested parentheses (see Figure 4.1). Expanding the lines being searched once after a failure to match would solve this issue and give an accuracy of ˜96% on this benchmark suite. However, a new test set would be needed after making such a change. Expansion also allows incorrect matches.

```
4: ⋆const PropertyListing = ({ title, address, price
   , bedrooms, bathrooms, image }) => {
5:  // Input validation can be implemented here if
   needed for additional logic
6:  return (
    ...
14:      </div>
15:   );
16: }⋆
```

Listing 4.1: Code that led to an ending line number error. gpt-4-turbo-0125 incorrectly chose line 15 for the final line of the segment after correctly stating the full text of the segment, including the brace on line 16.

In two of the match failures, the model failed to correctly copy the text of the updated segment. One of these instances (the response for benchmark 30) omitted a comment from the updated text, and another (for 63) copied the snippet from the original file rather than the text from the updated file. 63 was the only observed misunderstanding of the primary task.

Latency

Output from our prompt was about 30 tokens plus the number of tokens in the segment, which may be arbitrarily long. The average generation time for

our system over the benchmarks using the gpt-4-turbo-0125 endpoint was 4.4 seconds.

4.7 Discussion

Here we discuss the implications of these results and the challenges faced by an LLM-maintained, document-external annotation system.

4.7.1 Capability of Current Language Models

Accuracy, latency, and cost tradeoffs

Putting aside the off-by-one line number issue, gpt-4-turbo-0125's responses other than the response to benchmark 63 matched human expectations. For the kinds of re-taggings in our benchmark suite, there is no question that an existing language model is capable of the task in a vacuum, or that the accuracy could be pushed arbitrarily high by resampling. However, the model's high cost and average response time are problematic. If cost is not a concern, since tag positions are independent, re-tagging can occur in parallel for all tags on a document, but full parallelism still requires an LLM instance for each tag. As the speed and availability of LLMs increases, this problem may diminish, but base performance will still need to increase considerably, or more creative prompts that process tags together will be needed, to scale to documents with hundreds of tags.

How else might this issue be overcome? One method we attempted was

to support smaller, faster models. Mixtral-8x7B-Instruct-v0.1, a fast open source mixture-of-experts (MoE) model, usually responded in around one fifth the time of gpt-4-turbo-0125, with hosting costs around one fortieth despite performance on par with gpt-3.5-turbo-01.25. Initial tests with Mixtral were promising, but we ultimately found significant enough issues with reliability during prompt tuning (discussed in section 4.5.1) that we did not proceed to evaluation. This process was repeated with Gemini Pro and gpt-3.5-turbo-0125. This may indicate a fundamental lack of ability in smaller models, but more testing is needed; it may still be possible that a system with enough checks could at least solve the accuracy issue for small models, though it might require many more requests to the language model.

A more promising direction we did not explore would be to fine-tune a small model on examples collected from our code generation system (or a derived system representing a greater variety of cases). Depending on the level of success in this, fast and accurate re-tagging across many platforms could be achieved relatively soon. Another approach might use fine-tuned models for likely easier triage and validation steps to avoid calling an expensive model in most cases.

Threats to validity

As a synthesized benchmark suite, our evaluation faces obvious threats to external validity. At present, our suite represents a very low bar which any re-tagging system should clear without issue, and does not test the reliability

of our system on an actual code base. At best, it establishes that gpt-4-turbo-0125 is capable of following the movement of entities in refactorings performed by the model itself. However, the failures of the smaller models reveal that this result has some value.

4.7.2 Common Difficulties

Our system and evaluation reveal at least 3 problems a semantic re-tagging system must address.

"Whitespace" and tag matching

When building our system, we were surprised to find that the text output by the language model often differed from the input text due to whitespace. With the right prompt or model tuning this issue may go away, but without a solid solution, matching model output to the code in the buffer is a considerable challenge. Even though the system's focus is on semantics, possibly-brittle syntactic bookkeeping remains necessary.

Annotation orphaning and duplication, and intent

As described in Section 4.4.2, our system does not attempt to handle annotations whose anchor text is simply removed from the file, and we excluded several of these cases from our benchmark. Duplicated anchor text was also excluded. These represent very common real-world cases that must be handled in practice: code is often entirely deleted or copied from one place to

another. A robust system must detect these cases and may be able to leverage an intelligent agent to decide whether the anchor text vanishing or multiplying means that the annotation itself should vanish or multiply. This is a more complicated problem than it may seem. For example, in an excluded benchmark, a tag on a loop condition vanished because the loop became a reduce() call. Should the tag have been re-applied to the reduce() call? Knowing that the tag referred to the *length* of the array being looped over, maybe one would say it should not, since the reduce call has no reference to the array's length. However, if we suppose the annotation represented a comment on the loop body noting that "the loop will execute *length* times, so the programmer must be careful not to pass in very long arrays," we would probably conclude that the warning is still valid and should find a new home, perhaps on the array itself. In other words, knowing the intent of the annotation allows it to be removed or reapplied appropriately.

For these reasons, our definitions in section 4.3 include a notion of intent. However, the complexity introduced by intent led us to avoid it while testing this initial system. We hope to see an expanded set of benchmarks in the future that include full information about the content and intent of annotations in order to allow addressing cases like these.

Truly ambiguous re-taggings

Even with intent, a system will still fall short in cases where even a human would not know what the programmer wants. A robust system should

detect these cases and offer suggestions for reasonable options. In limited tests on simple examples, we found gpt-4-turbo-0125 limited its responses to correct alternatives (sometimes missing alternatives), while smaller models occasionally suggested far-fetched (arguably incorrect) re-taggings.

4.8 Conclusion

In conclusion, this paper presents a novel approach to leveraging language models for the automated re-tagging and maintenance of document-external markup, highlighting the potential of LLMs to significantly enhance code annotation processes. Together, our formal description of annotations, adaptable code for benchmark generation, the synthetic Tagged Code Updates benchmark dataset, and our example implementation provide starting points for research on this technique. Our evaluation measures the viability of current language models for accurately re-tagging code in response to modifications. The gpt-4-turbo-0125 model indeed handles cases that traditional methods cannot with promising accuracy rates, but challenges such as properly matching the model's output to the file and handling annotation ambiguities remain.

Our research underscores the importance of intent and semantics in annotations and suggests a future direction for automated code maintenance tools. Ultimately, our findings lay the groundwork for further exploration into the integration of programming languages, markup systems, and artifi-

cial intelligence.

4.9 Appendix

4.9.1 Benchmark Generation

Example of a high-level problem description generated by gpt-3.5-turbo-0125.

Problem: A local coffee shop wants to create a dynamic menu on their website that displays the prices of different coffee beverages based on the size (small, medium, large) selected by the user.

Constant:

- menuItems: An array of objects containing information about each coffee beverage (name, small price, medium price, large price).

Steps to Solve:

1. Create an array of objects called menuItems, where each object represents a different coffee beverage and stores its name, small price, medium price, and large price.

2. Create a component that displays the menuItems in a visually appealing way, with buttons for selecting small, medium, or large sizes for each coffee beverage.

3. Add logic to the component that updates the displayed price based on the size selected by the user.

4. Test your component by rendering it on a webpage and verifying that the prices update correctly when different sizes are selected.

Example of a high-level snippet description for SNIPPET_TYPE "constant" generated by gpt-3.5-turbo-0125.

One particular constant in this program is 'menuItems'. It is an array of objects that represent different items on a coffee menu, each object containing the name of the item, along with the prices for small, medium, and large sizes of that item. This constant is used to display the menu items and their prices on the screen.

Example of a high-level update description generated by gpt-3.5-turbo-0125.

One potential change a real-world programmer might consider is refactoring the way the menuItems data is structured. Instead of using separate properties for smallPrice, mediumPrice, and largePrice, they might consider grouping the prices together under a single prices object for each item. This

could make the data structure more flexible and easier to work with, especially if there are plans to add more size options in the future.

In this partially applied state, the programmer may have created a new data structure for menuItems where each item now has a prices object that contains the prices for different sizes (small, medium, large). They may have updated the CoffeeMenu component to work with this new data structure for some items, but not for all items yet. As a result, some items may still be using the old separate properties for prices while others are utilizing the new prices object.

This halfway refactored codebase may demonstrate a transitional phase where the programmer is in the process of updating the data structure and component logic to be more scalable and maintainable.

Prompt to get the updated code with the snippet still properly delimited.

Consider the following problem:

```
<problemDescription>
${problemDescription}
</problemDescription>
```

Now consider this code that tries to solve the problem:

```
<program>
```

```
${codeWithSnippetDelimited}
</program>
```

Note that a snippet from the code has been marked with a "${delimiter}" on both sides. This snippet is described as follows:

```
<snippetDescription>
(${snippetType})
${snippetDescription}
</snippetDescription>
```

Now consider the following description of an update to the program:

```
<updateDescription>
${updateDescription}
<updateDescription>
```

Apply this update to the code as described. Your response should be purely code without any external discussion, and should fully copy any relevant sections of the original program. In order to obtain credit, you MUST maintain the "${delimiter}" marks on the snippet or its updated version. The contents of the snippet should be functionally identical in the new version of the code.

Again, the updated version of the code MUST have a SINGLE pair of "${delimiter}" marks referring to the same snippet in its new position or form.

4.9.2 Annotation update system

Prompt with an example program and update

Consider the following file:

```
<INPUT>
1:
2:#include <stdio.h>
3:
4:int main() {
5:    int numItems;
6:    float totalAmount = 0;
7:    float discountedAmount = 0;
8:
9:    printf("Enter the number of items in the cart: ");
10:    scanf("%d", &numItems);
11:
12:    for (int i = 1; i <= *numItems*; i++) {
13:        float price;
14:        printf("Enter the price of item %d: ", i);
15:        scanf("%f", &price);
16:
17:        totalAmount += price;
18:    }
19:
20:    if (numItems >= 5) {
21:        discountedAmount = 0.1 * totalAmount;
```

```
22:         totalAmount -= discountedAmount;
23:     }
24:
25:     printf("\nTotal amount: $%.2f\n", totalAmount);
26:     printf("Discounted amount: $%.2f\n", discountedAmount)
    ;
27:
28:     return 0;
29:}
</INPUT>
```

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

```
<SEGMENT>
numItems
</SEGMENT>
```

Next, consider the following updated file:

```
<UPDATED>
1:#include <stdio.h>
2:
3:int main() {
4:     int numItems;
5:     float totalAmount = 0;
6:     float discountedAmount = 0;
7:
8:     printf("Enter the number of items in the cart: ");
9:     scanf("%d", &numItems);
```

```
10:
11:     float prices[numItems];  // Introduce an array to
    store the prices of the items
12:
13:     for (int i = 1; i <= numItems; i++) {
14:         float price;
15:         printf("Enter the price of item %d: ", i);
16:         scanf("%f", &price);
17:
18:         prices[i - 1] = price;  // Store the price in the
    array
19:
20:         totalAmount += price;
21:     }
22:
23:     if (numItems >= 5) {
24:         discountedAmount = 0.1 * totalAmount;
25:         totalAmount -= discountedAmount;
26:     }
27:
28:     printf("\nTotal amount: $%.2f\n", totalAmount);
29:     printf("Discounted amount: $%.2f\n", discountedAmount)
    ;
30:
31:     return 0;
32:}
</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.

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:

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.

Chapter 5

Codetations

This chapter is adapted from my paper:

[MVTT25] Edward Misback, Erik Vank, Zachary Tatlock, and Steven L. Tanimoto. "Codetations: Intelligent, Persistent Notes and UIs for Programs and Other Documents." *arXiv preprint arXiv:2504.18702* (2025).

5.1 Overview

The culmination of this research applies the semantic anchoring techniques from Magic Markup to create a complete system for attaching rich, interactive context to code. Codetations[1] extends beyond simple annotations to support dynamic, interactive UIs that can execute code, visualize data, and respond to changes in the document. Through user studies and worked exam-

[1]Available through the Visual Studio Marketplace, with a public repository at https://github.com/elmisback/magic-markup/tree/main/vscode-extension.

ples, this chapter demonstrates how document-external annotations address key pain points in current documentation practices while enabling novel forms of programmer assistance. The system reveals the potential for a new programming paradigm where contextual information—both human-authored and tool-generated—persistently augments code without compromising its clarity or structure. This final chapter also explores how such context benefits not only human developers but also automated agents like LLMs, bringing the dissertation full circle by showing how robust context management enhances all forms of code manipulation.

5.2 Introduction

Software development is inherently contextual. Developers maintain mental models that encompass far more information than exists in source files, including design decision rationales, implementation trade-offs, edge cases, maintenance processes, and debugging workflows; these conceptual models rarely find a permanent home in the codebase itself. With the rise of large language models (LLMs) and other programmatic agents and tools that enhance development workflows, the problem of missing context surfaces not only when code is handed over to new maintainers (or when programmers forget their own past work) but whenever any agent is invoked without sufficient context, contributing to well-known problems like hallucination [MIT23].

The problem of missing context cannot be solely attributed to over-

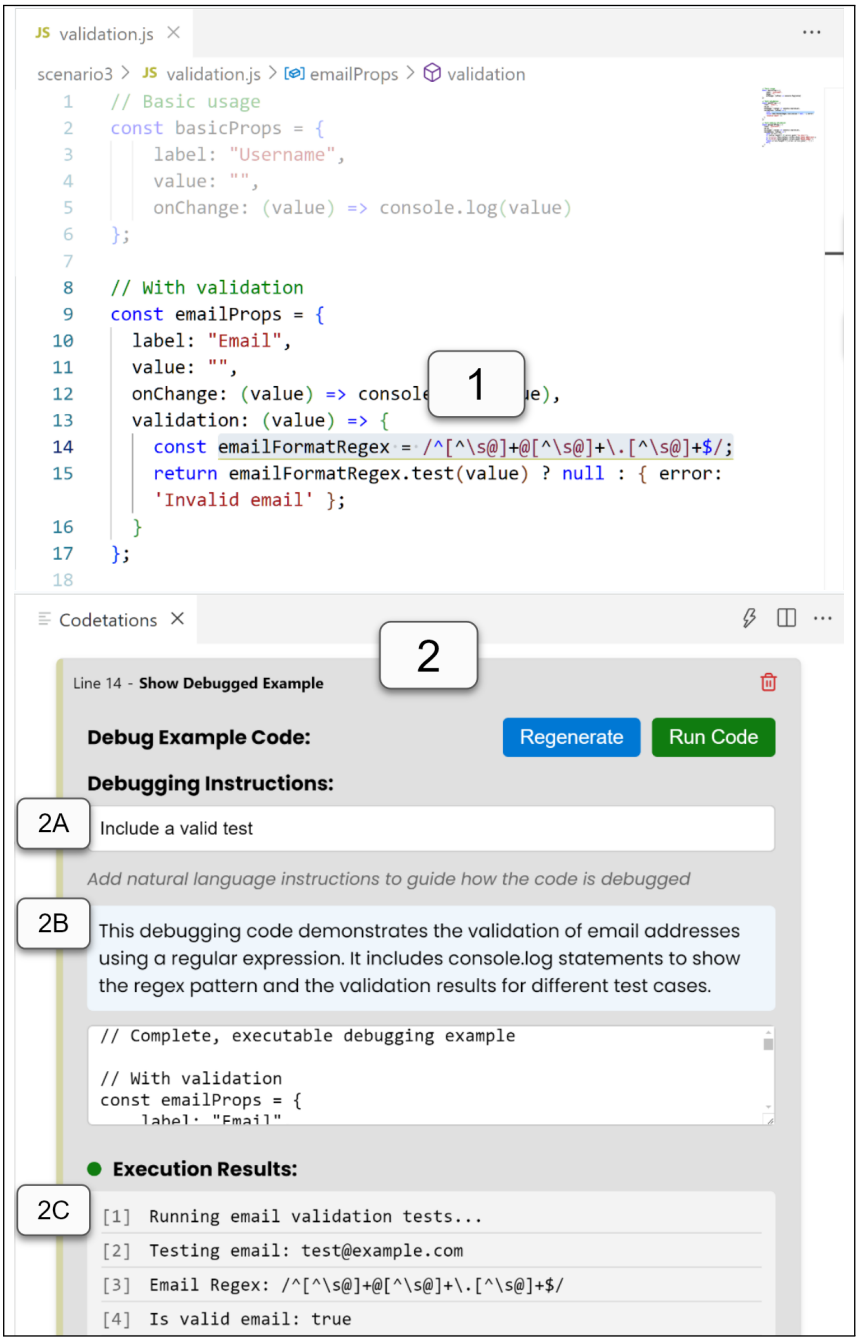


Figure 5.1: Codetations lets users attach rich, living annotations to any text file. Here, (1) the behavior of highlighted code is illustrated by (2) a "Show Debugged Example" annotation through (2A) user-customizable debugging instructions, (2B) LLM-generated test code, and (2C) live execution results. We added this annotation type to our system in just a few minutes using the prompts shown in Appendix 5.10.3. Unlike traditional comments and notebooks, Codetations does not modify file content but instead uses a hybrid editor/LLM anchoring system robust to both online and offline edits.

whelmed developers who lack time to explain their assumptions. Our need-finding work (Section 5.5) highlights that developers *do* seek ways to maintain development context as a natural part of the programming process: they sketch solutions by hand, maintain personal notes in separate documents, trace through example code executions in REPLs, discuss approaches with teammates, or use other long-form methods for code modeling. Developer attempts to creatively structure this context as code comments can lead to "messy" code and hard-to-manipulate representations, such as ASCII art diagrams [HHC+24]. The reality we find is that *many kinds of context are simply ill-suited to representation within a program's code,* forcing developers to keep it elsewhere and recall when and where it is relevant.

Context is a language-independent property of every codebase, but no widespread, editor-level approach attaches rich context to specific parts of code and documentation. We posit that LLMs may both finally bring this problem to a head and become part of its solution.

We propose that editors assist programmers in linking individual code spans to external documents, executions, or diagrams, including live diagrams. Of course, such systems have been proposed before—but they suffer from a key limitation: how to keep the contextual annotations in sync with code as it changes. Not only does that mean tracking code spans as the code evolves (usually called *anchoring*), but it also means updating the external documents, executions, and diagrams as the code changes (which we call *syncing*). Our insight is that modern LLMs, with their rich semantic under-

standing of code and documents, allow us to finally surmount this challenge and automatically keep external contextual information in sync with code as the code changes.

Guided by early need-finding and prototype work, we structure our investigation around three questions:

1. Do semantically anchored, richly interactive annotations tangibly help programmers—and which parts of the concept matter most?

2. What contextual information do today's programmers need to capture and retrieve, and how do they expect those needs to grow?

3. How much does explicit context improve an LLM's reasoning about code repair issues that it could, in principle, already solve?

As a design probe, we present Codetations, a VSCode extension that hosts interactive annotations and keeps them attached and synced for any code repository. We solve the anchoring problem by tracking code edits in real-time through the editor and employing LLMs to resolve offline updates. We solve the semantic synchronization issue by offering annotation developers an easy-to-use API that lets annotations access host system resources, including, for our implementation, the document text, other annotation data, and editor APIs, including an LLM API.

Our proposed Codetations system allows a programmer to work with a variety of custom UIs for recording, manipulating, and displaying context during a programming workflow; we call these custom UIs *annotation types*.

The Codetations API is designed to make new annotation types easy to build: we found that a single request to an LLM with examples of the API's usage was sufficient to immediately add powerful new tools to the system, including the Show Debugged Example annotation (Figure 5.1).

In an interview-based user study (N=9) we conducted with experienced programmers to investigate their needs with respect to maintaining code context and responses to a working prototype system (Section 5.5), participants expressed enthusiasm about the system's features, with all study participants spontaneously suggesting or agreeing that document external annotations would permit greater storage of context with their code and imagining unique ways they could use Codetations-style notes in their own work.

Finally, we found that annotation data created by tools like ours was helpful to an LLM that repairs bugs in code (Section 5.7).

In sum, this paper's contributions include:

- **Codetations**, a system for keeping dynamic, interactive annotations on semantic entities as documents evolve (Section 5.4).
- Results of a **qualitative user study** exploring developer needs with respect to maintaining code context and developer responses to a working prototype (Section 5.5).
- **Recommendations** for new annotation types, including examples of rapid annotation generation with LLMs, and specific **design takeaways** for researchers conducting future work in this area (Section

5.6).

- Results of a **worked example** of LLM performance for debugging tasks with and without attached contextual information (Section 5.7).

5.3 Our Design Process

We now describe our initial motivations and questions regarding system design, discuss early prototypes we discarded, and finally present five key design decisions we made and applied to build our prototype system.

5.3.1 Design Motivation and Research Questions

We have noted that software development involves managing many kinds of contextual information. A one-size-fits-all approach to representing this diverse context is inadequate given the variety of possible use cases and stakeholders (including different kinds of programmer-users and programming agents). We considered the following research questions:

RQ1. Is there any contextual utility in semantically attached, automatically contextualized notes with rich and interactive interfaces? How important are the individual parts of this hypothesized solution?

RQ2. What are the needs of programmer-users today when creating and managing context? What contextual needs do programmer-users anticipate having in the future?

RQ3. How important is context for LLM reasoning? To what extent do current LLMs rely on contextual hints to solve problems whose answers they are capable of reasoning over (e.g., can model A always solve a programming problem designed by model A)?

At a high level, RQ1 probes a possible solution for the challenge of context management, and RQ2 and RQ3 focus on understanding needs in order to further evaluate that solution. We reflect on these questions in Section 5.8.

5.3.2 Early Explorations

We began our investigation by discussing ideas for the system with colleagues but found it difficult to conceptualize and explain the system without a tangible demonstration, so we created a rough working prototype (shown in Appendix Figure 5.5). This early implementation was fully editor-independent and browser-based. The interface, while functional, had significant usability challenges, but even beyond these, after explaining the system's purpose and operation, pilot users still had difficulty understanding the system's purpose and imagining realistic workflows using it.

This invaluable experience with our initial prototype prompted us to conduct two rounds of user studies with the current prototype, as described in Section 5.5. We consider that characterizing the shortcomings in both our early and current implementation constitutes perhaps our most valuable contribution for other annotation system designers.

The primary lesson we learned was that programmers have existing men-

tal models for how annotations should look and work, and our initial system did not meet its users' expectations. To elicit meaningful feedback on our research questions, we decided to focus on identifying existing user workflows and mental models and ensuring that our system provided obvious value within those constraints before asking users to suggest additional possibilities and consider future systems.

As we navigated the design space, we encountered key design questions (below) where this lesson helped to constrain our responses, and we explore these choices in the following sections.

5.3.3 Five Key Design Questions

Should annotations be hosted through the editor interface and belong to the editor conceptually, or should they reside elsewhere?

This question addresses our highest-level design choice. Based on the difference in users' immediate understanding of the system and ability to imagine further capabilities after we rewrote it as an editor extension, we unequivocally find that *users* consider annotations to be an editor feature. But should designers? In this case, the preference of users runs counter to the software design principle of separation of concerns, which tries to keep features independent to allow systems more degrees of freedom. Annotation interfaces for read-only code could certainly be hosted in applications that are not editors, e.g., a browser extension could maintain annotations for Github reposito-

ries, and if an annotation interface were fully independent, it could service documents across many different user applications, like Reiss's Field system [Rei90].

For our implementation, however, we decided that editor integration provides clear benefits: annotations can (1) appear on the code to which they are attached with their (2) content rendered nearby and (3) respond to edit actions with low latency, matching the mental model that annotation users have formed from document editors like Microsoft Word and Google Docs.

What kind and customization of annotation types should we host in our prototype system to communicate the system's capabilities to a typical programmer?

Since contextual information is useful in most parts of the programming process, we first discussed the range of possible annotation data types and workflows that would use them. These choices inform the API a system needs to offer for annotations to work, e.g., a file-writing API is needed for annotation types that modify the region where the annotation is attached. Section 5.6 describes the kinds of annotations we created or proposed and those suggested by participants in our user study.

How should we connect users' incremental or major file edits to updates in the anchor position and to the content of annotations?

Broadly, there are trade-offs between correct responses to changes and latency. For example, using an LLM to obtain a more semantically correct update of the position or annotation content is currently relatively expensive in terms of both time and computational energy. Further, a question arises about whether updates from an LLM should wait for user permission to run in case the results would violate user expectations and require intervention.

Of course, the issue of anchoring could be solved entirely by embedding markers directly in code as comments. However, doing so means mixing the annotation layer into the document, which requires all parties using the document to agree on an acceptable level of code clutter and to avoid accidental damage to markers when documents are edited; it also cannot work in environments without comment support, like JSON files, and requires write access to annotated documents. We think these issues rule out this method for use in an annotation system intended for many kinds of codebases and documents.

Our prototype system tries to strike a balance between these options by responding to online edits that occur through normal editor actions with simple automatic positional updates and reserving the slow, semantically correct, and potentially surprising reattachment method for cases where user permission has been received after an offline update (e.g., when a remote user merges a branch).

To users, what are system-wide capabilities for working with any annotation vs capabilities for a specific kind of annotation?

Users may request many powerful features for designers to consider. These choices chiefly affect the information that is stored and must be kept accurate for every annotation. Annotation anchor information (which describes the document part the annotation refers to) and actual annotation data must always be included. Beyond these essentials, other information can be considered, such as globally unique IDs, order or positions within the set of annotations, author information, timestamps, content hashes, or explicit cross-references between annotations.

Our current system supports only an ID; a single annotation type name, which it uses to render the annotation as a React component; and a component-controlled data field for saving state. This approach still lets annotation types host their own complex behaviors (e.g., cross-referencing could be supported by reading other annotations' data), but system-level behaviors remain relatively basic. Multiple users in our study wanted more than this; in particular, they saw value in annotations that could point to multiple code sections (see Section 5), which would require the formal definition to be expanded to permit multiple anchors.

Where is the user data stored?

Though this may seem like an implementation question, the location of users' data directly affects and unifies many user-facing concerns about collabora-

tion, sharing, and version control workflows, and we maintain that users should always know where their data is being kept when using a system.

When annotation data is stored along with code in the repository, teams immediately gain shared context, and annotations naturally follow version control history. However, such sharing also means that annotations become part of the codebase's footprint and could become a source of merge conflicts. This decision also affects the basic user experience for teams: repository-stored annotations are simple but require commits for other users to see annotation changes, while database solutions offer real-time collaborative features but introduce authentication complexity. Different storage models also enable different annotation features; some kinds of annotation may need to persist sensitive information separately from public repositories.

Our prototype uses a JSON file in the repository for simplicity, but using a mixed, local-first approach could let users opt in to private and shared storage, better serving diverse user needs.

5.4 System Design

Codetations is intended to probe in a structured way the design space issues identified by the questions in Section 5.3.3. The interface is shown in Figure 5.1. At the highest level, Codetations allows a programmer working in an editor to add persistent, self-updating, interactive annotations to selected text. An annotation view for displaying and modifying annotations is shown

to users in the editor adjacent to (or perhaps interleaved with) the file buffer.

Appendix 5.10.1 describes basic features any annotation system should have. Here we distinguish novel features Codetations provides, examine its architecture, and provide details about the system version we implemented for our user study.

5.4.1 Novel Codetations Feature Set

Document-external annotations

No changes are made to the text of the file when an annotation is added. Instead, the specified anchor points for the annotation are noted in a separate annotation data file.

Intuitive annotation movement through semantic anchoring

Broadly, the system tries to keep annotations attached to the same semantic entity as the text evolves. As the programmer edits the code, the system updates the anchor points using simple heuristics that match the behavior of comments in popular document editors like Microsoft Word. However, for edits not initiated by the programmer, e.g. when the code is updated on disk by a remote collaborator or a different editing program, Codetations notifies the user of any annotations that have become detached and requests permission to update the anchor points. If the user confirms, the anchor points for each annotation are re-attached to semantically similar points in

the new document using the best available method (in our implementation, the method from Chapter 4). This protects the annotation data from the unavoidable real-world case where remote developers are not running Codetations. Additional measures to ensure anchor consistency, such as periodically checking whether anchor points need to be adjusted, can be considered but were not implemented for our prototype.

Automatic annotation and file content updates through context-aware annotations

The annotation system itself does not include domain knowledge or heuristics to decide whether annotation or file content is up to date; instead, it delegates this responsibility to the annotation implementations. Annotations receive updates about the document's contents and the section of the document they are attached to, and they are provided with an API (that could be extended with a permissions model) for writing to the document, as needed.

Interactive "code"-tations

To probe the full space of possible annotation behaviors mentioned in Section 5.3.3, even beyond the file access necessary for 5.4.1, Codetations is designed to host dynamic, interactive annotations with programmatic behaviors. It also offers an API through which the programmatic power of a host system is offered to annotations.

5.4.2 Codetations System Architecture

Annotation definition

Each annotation has a *tag* (i.e., an annotation attachment record) stored independently from the document. A Codetations annotation record has the following fields:

1. A universally unique identifier for the tag
2. The start position of the annotation in a document version D
3. The end position of the annotation in D
4. A method for obtaining D
5. An *annotation type* to use for rendering
6. Data for the annotation

Elements 2 through 4 anchor the annotation to provide the external anchoring from Section 5.4.1; any alternative anchoring method should be able to produce these elements. List element 5, *annotation types*, are an implementation-dependent mechanism (described in Section 5.4.3) for allowing the system to provide the content updates and programmatic behaviors described in Sections 5.4.1 and 5.4.1 above.

Host and rendering processes

Codetations consists of a *host or editor-related process* and an *annotation rendering process* (that may render in the editor or elsewhere). The host process provides access to the document or buffer, the annotation data, and any APIs or other system or editor functions required by the rendering process. In our implementation, this host process is a NodeJS VSCode extension. The rendering process gives annotation types access to the functions exposed by the host. In our implementation, the rendering process is a VSCode Webview that can render components written with HTML and JavaScript.

5.4.3 Implementation Details

In this section, we describe the main implementation-dependent features of our prototype. For other implementation details, see Appendix 5.10.2.

Annotation types

Annotation types are defined in our implementation as React function components that receive as arguments (1) the annotation data and (2) an API for getting and setting the document text and annotation data field from the Codetations host.

API for annotation types

Our implementation gives annotations access to the following annotation type API:

1. Functions for reading and writing the annotation's data
2. Functions for reading and writing the document text
3. A function for calling the VSCode Language Model API[Mic24]

This API could be extended with additional host access, e.g., to permit use of the host's interpreters or compilers for executing annotated code.

We found that providing a description of this API to a language model allowed us to **generate complex annotation types with a single prompt**, including the Show Debugged Example annotation type (Section 5.6.1, prompt in Appendix 5.10.3) and the LM Unit Test (Section 5.6.2). This capability could be offered to end-users to allow them to create new annotation types for a codebase on-the-fly.

5.4.4 Annotation data

We store tag records in a JSON file that contains a simple array of tag records. Tags are fully independent, i.e., they can be updated in parallel, and they will not be affected if other tags become damaged. This setup also allows annotation types to be written as a function of a single tag record (see Section 5.4.3).

We place this JSON file in a lazily generated tree stored under a hidden ".Codetations" directory kept at the root of its git repository. We noticed that study participants immediately understood how to find the raw annotation data and understood that it would be synced with the rest of their repository. As Section 5.3.3 notes, keeping annotation data in version control involves significant trade-offs.

5.5 User Study

To evaluate Codetations and understand developers' contextual information needs, we conducted a user study with a small convenience sample of 9 participants from varied backgrounds including academic researchers, industry developers, and students, with programming experience ranging from 5 to 30 years across domains such as web development, scientific computing, compilers, and SQL database programming. Participant backgrounds are shown in Appendix Table 5.1.

5.5.1 Study Design

Sessions were conducted remotely or in-person, lasted 60-90 minutes, and followed a three-part structure, with about 20 minutes allocated for each part: a pre-demo need-finding discussion and survey focused on documentation practices (see Appendix 5.10.8); a hands-on demonstration of Codetations, described below; and a post-demo discussion and survey to gather immediate

reactions to our system and perspectives on its future (see Appendix 5.10.11). We iterated on the demo and surveys once during the course of the study, dividing the study into 2 rounds.

Participants installed our VSCode extension on their system and cloned our study repository, then completed 6 annotation tasks focused on various code examples and annotation types, shown in Appendix 5.10.10. Other than specific exceptions that we attribute to easily-addressed issues in current LLMs and LLM APIs (described in Appendix 5.10.6), all of the demo tasks were completed by round 2 participants without significant assistance from the interviewers.

5.5.2 Need-finding Results

Our need-finding pre-demo survey and discussion revealed several consistent pain points in managing code-related contextual information.

Documentation Fragmentation

Participants consistently described challenges with contextual information being scattered across multiple disconnected systems. P6 and P9 both discussed using shared and private drives for storing documentation, with P9 admitting that "a really hard problem was that all of our docs were just like scattered in people's own Google drives." In other words, this useful documentation was effectively privately stored and inaccessible to other team members. P9 also summarized the consequences of this kind of fragmenta-

tion: "[I]f 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." P6 echoed this, spontaneously saying that "the documentation tends to be pretty far from the line by line parts of the code." Speaking about what they would want from documentation before seeing our system, P7 said that "it would be nice at times to have some sort of link or embedded document, but writing out links in code is kind of ugly." This last point helps to explain why fragmentation happens: injecting external links or the documentation itself into code was perceived by participants as impacting the code's quality, as we expand on in Section 5.5.2.

Users felt that Codetations did a good job of meeting the challenge of fragmentation. P6 felt that the organization of information through Codetations helpfully reduces the distance between relevant information, saying it would allow them to "just go to the part of the code where the information I want is relevant, and then ... just go through this small set of comments that I've made over time." P8 called "documentation ... that doesn't get lost in [their employer's] giant database of docs because it's attached to the code" one of a number of "killer features" that Codetations could enable.

Documentation Generation and Maintenance

Participants generally recognized the importance of documentation, but pointed to practical challenges. Participants noted that documentation is often not prioritized or is "undervalued" in development workflows. Even P5, a profes-

sor who teaches software engineering, admitted skipping documentation when busy despite knowing "the documentation will save me and others time and money in the future." Participants called initially creating documentation "the hardest part" of the documentation process and also consistently rated keeping documentation up to date with code changes as difficult.

While participants like P8 praised the system for enabling annotations to update and show continuous integration-like feedback as the code changes (see Section 5.3), we noticed that the problem of initially creating documentation remained largely unaddressed. We discuss this limitation in Section 5.8.3.

Media Limitations

Many participants expressed frustration with the limited expressiveness and forced trade-offs of current documentation tools. P6 complained that code comments "constrain them to just words" and said they use a physical notebook to draw things out, both for their own understanding and to communicate with other team members; they also later noted these notebooks eventually get thrown away (which makes sense based on Section 5.5.2—the value of the notes drops as soon as it is hard to relate them to the code). Participants familiar with computational notebook systems wanted embedded images, rendered equations, and diagrams for code. P8 called this issue a "pain point" and described substitutes like ASCII art as "a crutch, where you'd probably just want an SVG or a real image editing tool."

As P8 put it, hosting "arbitrary widgets that do cool stuff" through our system could address this issue.

5.5.3 Participant Responses to the Demo

After interacting with our prototype, participants shared their reactions and insights about its utility and potential applications.

Non-Intrusive Annotations Encourage Keeping More Context

Most critically, *all participants quickly understood the ability to maintain rich contextual information alongside code without modifying the source files as increasing possible context for a codebase*, usually bringing it up without the interviewer even mentioning the point. P6 described the system as letting them "write down all these ideas and just have them around, but they're not imposing on the actual code." P3 saw the scalability advantage compared to inline comments: "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." P1 also described extended inline comments as "annoying" in comparison. P9 pointed to new opportunities to add extra information to codebases, like developers' provenance stories about how code got added to the codebase.

Robust Code Tracking

The system's ability to track annotations predictably as code is edited and through significant code changes was also valued by participants. P1 saw the advantage over traditional code comments, saying that "when you're working with code, you don't want to have all those [traditional] comments everywhere because if you're moving stuff around, things get jumbled up." P3 imagined a system tracking the movement of annotations across files (which is not yet supported by our prototype) using a similar kind of semantic tracking. For the most part, it appeared that participants expected this aspect of the system to work out of the box, with P1 reacting to the LLM repositioning prompt after a complex update to the code by saying repositioning should happen "in the background without me even having to think about it."

Interactive Features and Testing

Several participants highlighted the potential of stateful interactive annotations. The Show Debugged Example annotation type shown in Figure 5.1 and described in Section 5.6.1 was created based on recommendations from P2 and P3, who imagined annotations that show the data and execution process for code. P1 and P6 also recommended tools that permit a user to describe what they want a section of code to do at a high level and receive suggestions for how to do it; these recommendations inspired us to create the LM Unit Test annotation type we discuss in Section 5.6.2. Our round 2 participants praised both of these annotation types, as we note in the relevant

sections below.

5.5.4 Future Vision

In the final part of the post-demo discussion, participants shared their perspective on the role of systems like ours in the future of programming. P2 and P7 recognized the growing importance of connecting domain expertise with code, especially for applications in business and the sciences. P9 also envisioned *annotations as facilitating a shift in programming abstraction and satisfying a new need for distinctions between human and model-generated information in documents*, speculating that

> Maybe people will write pseudocode and then that will be linked in some way to the [model generated] code implementation of the pseudocode ... In that world, sorting through the noise will become harder... That seems like a place where annotation can be useful, like, here's a pile of code, and then here's some evidence that there's a person who has thought about it.

Overall, our user study revealed significant challenges with current approaches to code documentation and strong potential interest in non-intrusive, robust, and interactive annotation systems like Codetations, especially as AI plays an increasing role in software development workflows.

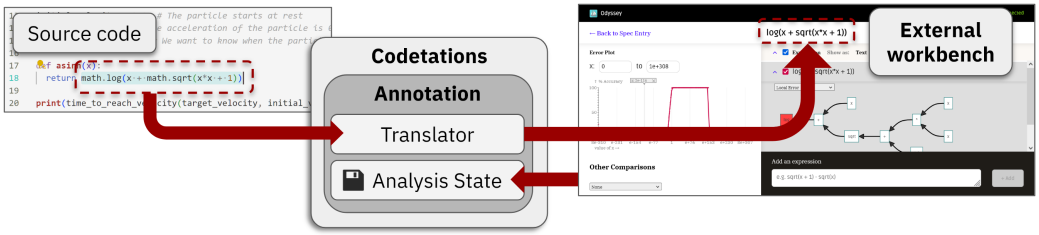


Figure 5.2: (right) Codetations connects the external Odyssey floating-point workbench, hosted on the web or in an editor, to (left) source code. An Analyze Floating-point Expression annotation uses an LLM to translate the Python expression shown on the left, math.log(x + math.sqrt(x*x + 1)), into log(x + sqrt(x*x + 1)), an expression in Odyssey's input language. This operation works for all input and output languages the LLM understands, immediately bringing Odyssey's analyses to a variety of languages that use floating-point operations. The user's work in Odyssey can be preserved in the annotation state.

5.6 Applications

We now describe the annotation types we implemented or planned as test applications for Codetations. We targeted applications involving different roles for annotations and different interaction patterns between the user, the document, and the annotation. (More applications are described in Appendix 5.10.12.)

5.6.1 Program Data and Live Execution

Programming languages provide a high level of abstraction, helping developers handle varied cases with generic constructs. However, this abstraction complicates efforts to understand code behavior in specific instances or contexts. The Babylonian programming system of Rauch et al. [RRR+19] solves

this issue by annotating programs with example data to illustrate how code behaves with concrete inputs. Other systems that also implement ways of showing execution examples, such as interactive notebooks, resemble the system of Rauch et al. by generally requiring developers to adopt a particular development environment or toolchain when targeting production code.

Using Codetations, we rapidly developed the Show Debugged Example annotation type depicted in Figure 5.1 to assess whether it could address this problem (see Appendix 5.10.3). This annotation type lets users explore and record the runtime behavior of any subsection of their program by attaching an annotation to it (for an example, see the summary and debugging code in Section 2B and execution results in Section 2C of Figure 5.1). Users can write debugging code manually or use a language model to generate the code based on anchor text, file context, and user-provided high-level guidance (Section 2A of Figure 5.1). If the file changes, users can ask an LLM to regenerate the debugging code while preserving as much of the existing debugging structure as possible (via the blue "Regenerate" button in Figure 5.1).

Users in our study responded very positively to this annotation type. P2 observed, "If you could run the code to get example output, that would...be useful for me," and P3 was enthusiastic about the possibility of both input and output value annotations for code (Section 5.5.3). After seeing the tool we built, P9 noted that "I didn't have to construct a whole input-output pair for some larger function. I could just say, 'I want to test this line.' That's definitely useful." Thus, the increased *addressability* of a Codetations-

annotated codebase (the ability to work with individual code subsections, such as lines, as introduced by Basman et al. [BLC18]) provides value relative to users' typical testing experience, partly by simplifying the testing process. P7 appreciated that "having [annotations with test cases] right next to the code is really nice."

If given more access to a host's resources, the Show Debugged Example tool could run with different code execution engines to help programmers with any language. We could also foresee extending it with "fuzzy execution," stepping through a program operation-by-operation using an LLM's notional machine rather than an engine, similar to a programmer carefully reading code during a debugging session to understand its function. Although a notional machine might initially be faulty and less reliable than a true engine, this is not necessarily a downside; we can view a debugging process as not just correcting the code, but also correcting the notional machine of the agent that generated it. Externalizing and aggregating critical corrections to a default notional machine over many annotations could ultimately be of lasting benefit to both programmers and models working with a particular codebase.

5.6.2 Document Observers and Agents: Program Validation, Certification, and Summarization

In complex software systems, it is often desirable to consistently check code for certain properties. These properties might be formal proofs of behavior or dynamic checks for performance, or even developer-noted checks for whether the code meets some loose natural language specifications, like ensuring the documentation for a function is current. Validating these properties is vital to ensure the correctness and performance of critical libraries and applications. Maintaining a historical record of these validations can facilitate monitoring the evolution of code quality and compliance over time.

Unfortunately, these properties often describe code sections on the scale of individual expressions or loop bodies, where the developer would like to check for very specific execution concerns. This low-level logic is not readily exposed at the function level, so it is difficult to unit test.

Codetations enables the attachment of 'document observer' annotations directly to code snippets, solving the problem. Such observers could monitor high- or low-level properties and save their state over time. Below, we discuss several examples of observers.

Analyzing floating-point expressions

Floating-point expressions often suffer from inaccuracies that can propagate significant errors throughout a software application. Odyssey, the web-based

computational workbench discussed in Chapter 3, lets developers working with floating-point expressions explore and optimize expressions by rewriting them while viewing their error. Unfortunately, such tools for analyzing floating point issues are scarce, and developers must locate and manually run them on particular expressions from a codebase even if their use is required for program validation. For library developers who rely on such checks to keep their libraries accurate and performing well, Codetations offers the immediate benefit of bringing the tool to the code.

Integrating an external tool like Odyssey into diverse development environments requires a way to translate between many possible input languages and the tool's input format. Odyssey expects expressions to be input in the MathJS format, which differs from the native formats used in most programming projects. Our "Analyze Floating-point Expression" annotation type finds a natural fit for this problem through the VSCode Language Model API, which we expose to annotations through the Annotation Type API in Codetations (see Section 5.4.3). An extension could call this API to transpile arbitrary floating-point code into the MathJS format required by Odyssey, as shown in Figure 5.2.

This approach leverages Odyssey's human-readable input format, which is easy for a language model to produce. For popular programming languages, this translation streamlines the use of Odyssey and maintains a high degree of translation accuracy. Although more testing is needed, this approach demonstrates a clear pattern for integrating any tool with a well-defined input

representation as a type of annotation. We envision the Analyze Floating-point Expression annotation type as just one example from an entire category of possible annotation types that wrap around existing analysis utilities with a well-defined intermediate representation used for input and output.

Study participants generally found this annotation type of interest for exposing a powerful analysis tool. According to P5, "I loved the tool that helped me figure out how to improve my floating-point arithmetic. You don't just point out the problem or give a list of options. It gives me a framework for evaluating the options [accuracy vs performance trade-off analyses], which is great." Participants easily discovered Odyssey and applied it to a codebase through our system, and future users of the same code could have found Odyssey output attached to the code to justify or certify their implementation choices. Significantly, if an implementation were to change, the same annotation would be immediately available to validate new choices.

The LM Unit Test annotation type

Consistently maintaining code documentation is a challenge in software development. Inline comments and documentation often become outdated or misaligned with actual code logic due to updates and modifications in the codebase. This misalignment can lead to confusion, errors, and increased maintenance overhead.

By attaching an annotation that uses an LLM to observe code changes, we can dynamically check and update code sections for specific properties,

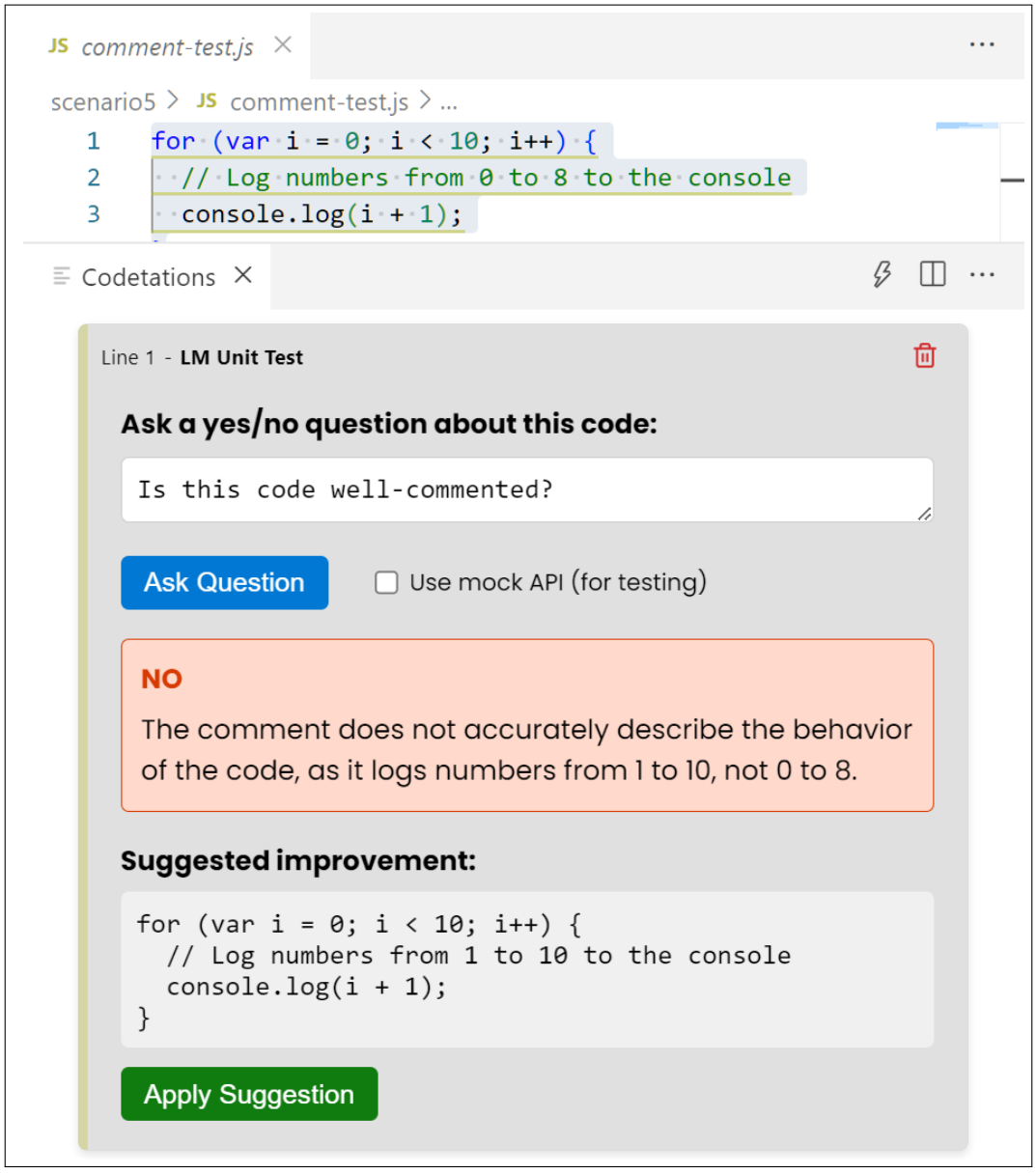


Figure 5.3: The LM Unit Test annotation type lets users ask yes-or-no questions about the code and suggests changes if the answer is "no." Suggestions can be applied to the buffer. The question stays with the code and can be asked again after the code is updated. This annotation type was generated in a few minutes by an LLM (see Appendix 5.10.3).

including summaries or explanations. Our LM Unit Test, shown in Figure 5.3, uses the editor's LLM API exposed by Codetations to provide a yes/no answer to a user's question about a code section; a 'no' answer causes it to suggest updates that would change the answer to a yes.

This annotation type has a variety of applications. Our study demonstrated the very simple case where a comment written by a beginner documenting the behavior of a loop is incorrect. The programmer might use the LM Unit Test in this case by asking if the loop is well-documented; the test will fail and suggest a correct comment. After the comment is corrected, the test will pass. P8 immediately perceived the value of this example: "Now if someone changes this code, like, refactors it, but forgets to change the comments, which happens all the time, then this test will fail in CI." P9 mentioned a real-world analogue: "Rust has mechanisms to run your documentation as tests... every piece of documentation should have a test embedded in it, and when that test fails, you go back and fix the documentation."

Consider a more realistic scenario where a developer has written a method such that it has no side effects. To check that there are no unexpected side effects, they add an LM Unit Test that asks, "Does this code have no side effects?" to compare their understanding to the LLM's. Later, another programmer refactors the method to improve performance but accidentally adds side effects. The LM Unit Test annotation automatically detects these changes, warns the programmer, and suggests updates to the method to preserve the property of no side effects.

Our implementation of this annotation type requires the user to manually request the LLM check, but we believe that automation of this process would likely reduce the developer burden involved with manually checking documentation. Compared to asking an LLM if any comments in the codebase are out of date, this targeted documentation-checking approach would also likely increase the chance that specific known issues would be noticed and thereby reach the developer's attention.

In general, annotation types like the LM Unit Test could reduce the manual effort required to keep documentation aligned with code changes, enhancing maintainability. We can envision similar annotation types that provide high-level summaries of the code they are attached to and manage the process of updating that documentation as the code evolves.

Other application domains

Keeping automatically-checked records with code has many other potential applications. On showing our system to a browser cryptography expert working on the Verified Software Toolchain project [App11], the problem of keeping proofs of subparts of cryptographic functions attached to the correct places in those functions as libraries evolve was immediately suggested, and the expert claimed that other critical systems like this also occasionally rely on manually updated attachments between certifications and certified code. While the risk of unpredictable output from an LLM's inclusion in a critical system should not be ignored, well-designed automated systems may make

it much easier to identify certification issues as libraries evolve.

Live-updated visual representations of values that live next to the code, like the representation of floating-point expressions in Odyssey, are a basic annotation type that have already established their popularity in interactive notebook systems. When they are interactive, such representations can also become *bidirectional* editors: changing the code changes the visual representation, and manipulating the visual representation leads to corresponding updates in the code. Hempel et al.'s Sketch-n-Sketch [HLC19] exemplifies this technique. Bidirectional editors are a kind of *structured editing* (first described as syntax-directed programming by [TR81]), where a user is presented with an interface whose affordances let them generate valid programs or objects only by manipulating the program's semantic structure, not its text. Gobert and Beaudouin-Lafon [GBL23] refer to these "alternative interactive representations of specific fragments of code" as *projections*. Although purely syntax-based attachment tools (that match all instances of a syntax or regular expression pattern) may better fit some kinds of values, they cannot save state for individual projection instances (e.g., history or an undo chain) outside the code, and they might be augmented by a stateful Codetations data layer. Codetations also permits quick, intuitive use of the projection technique for larger or less well-defined code sections, as Odyssey demonstrates for floating-point expressions.

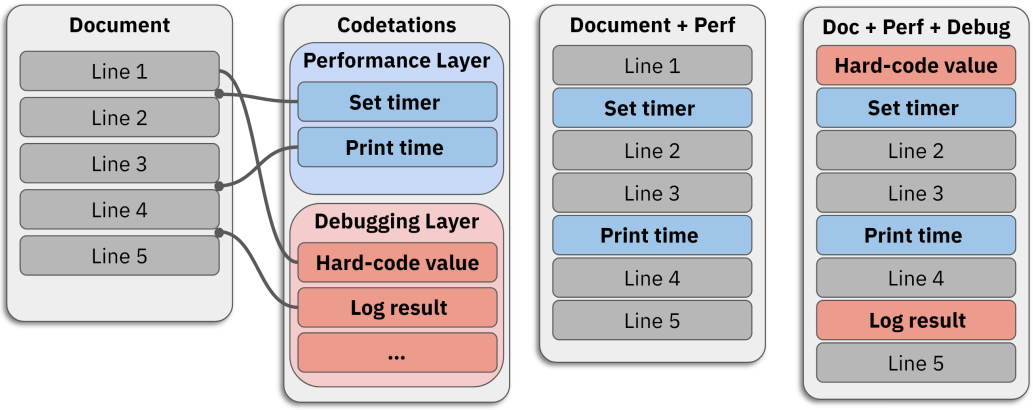


Figure 5.4: The document-external annotations enabled by our system could facilitate the separation of documents into layers that help with specific tasks, such as instrumenting code, without cluttering the main document.

5.6.3 Layered Documents

In digital design tools like the popular Adobe Photoshop program, the concept of "layers" helps users manage multiple overlapping elements that contribute to a final composite image. However, this concept has not traditionally been applied to textual electronic documents or codebases, which present content as a single consolidated file. The notion of layers in documents can offer a new way to separate and manage different types of content or functionality in a document.

Layered documents represent a novel user-document interaction pattern that a system like Codetations could enable.

The Add Layer annotation type

Software development often involves auxiliary code segments that are necessary for tasks such as debugging, performance testing, or conditionally executed features (like feature flags). These segments, though useful during development or testing, can clutter the main codebase and complicate maintenance and readability. Using the concept of layers, we could overlay auxiliary code on a main codebase without permanently integrating it into the main files, keeping the primary code clean and focused.

Our proposed "Add Layer" annotation type (not implemented for our user study), shown in Figure 5.4, aims to enable developers to attach, manage, and toggle sets of auxiliary code segments as layers over their main codebase. The tool facilitates a clean separation of concerns by allowing developers to activate or deactivate different layers depending on the current need, such as toggling debugging annotations or performance metrics without altering the underlying source code.

To use the Add Layer annotation type, a developer would attach the annotation in the position where they would like to insert new code. Then, they would specify the code to be inserted at that position and the name of the layer to which the code belongs. Next, an "ApplyLayers" command-line tool could be used to read the codebase's annotation data files and obtain an alternate version of the codebase with a specified set of layers applied.

Consider a scenario where a developer needs to add extensive logging to troubleshoot a specific issue. Using the Add Layer type, they can create a

new "debugging" layer where all logging code is added. This layer can be activated to gather necessary data and then easily deactivated once the issue is resolved, without leaving residual code in the production codebase. If the issue arises again after the code has changed, the layer can be reactivated to resume debugging. This approach keeps the codebase clean and formalizes the management of temporary or context-specific code modifications.

Layers could also be used by annotation types like the Show Debugged Example tool (Section 5.6.1) to automatically instrument code to gather runtime data.

5.7 LLM Evaluation

There's no question that context is required for any agent to resolve issues in a domain it knows nothing at all about. However, considering the amount of training data shown to modern LLMs and advances in their intelligence, is context still necessary for a domain like program repair? To investigate RQ3 and explore the impact of context on LLM reasoning (see Section 5.3.1), besides consulting the work of authors like Martino et al. on hallucination [MIT23], we also evaluated a few examples to understand to what extent context is necessary for current LLMs (ca. 03/2025) to recall and apply knowledge *known to be latent in the model*, especially for problems involving program repair. We also wanted to explore the effect of irrelevant information.

To guarantee that the information was latent in the model, we began by generating example debugging problems with an LLM without tool calling. Each debugging problem consisted of a program requiring repair and an explanation of the bug. Next, in each case, we constructed context that simulates real-world cues: partial documentation, debug outputs, and other JSON-formatted annotations based on those output by Codetations, with a hint at the true explanation mixed into a set of extraneous information. These sets of contextual information mimic the kinds of fragmentary artifacts one might find on a voluntarily-annotated codebase. Finally, we observed output for *the same* LLM when prompted to solve each debugging problem (1) without and (2) with context.

Our null hypothesis was that even in the zero-context condition, an LLM with excellent recall might be able to guess solutions, since it was able to think of the problems. If this were the case, the value of keeping particular notes on the code for problems an LLM knows about would diminish. However, our results support the view that, for current LLMs at least, even when extraneous information is included, useful context still significantly improves performance, particularly in scenarios where failures stem from missing external references or real-world assumptions. The following section examines one failure mode in detail, with a second case shown in Appendix 5.10.13.

5.7.1 External References

LLMs, like human programmers, rely on external references, such as inline comments, documentation, or APIs, to bridge gaps in their internal knowledge. When these references are omitted or misremembered, both humans and LLMs make predictable errors.

Consider a bug involving the GitHub API: the endpoint `https://api.github.com/users/{username}/repos` returns only public repositories unless an authentication token is supplied but returns 200 OK in both authenticated and unauthenticated cases. We prompted GPT-4o with this setup and a description of the bug: "This code returns no repositories for some users." Without context, the model recognized that the issue was related to access control but misattributed it to using an incorrect endpoint (and switched the code to using a different, incorrect endpoint). In reality, the endpoint was correct: the problem was missing authentication.

When we provided annotations, including distractor documentation and data from a Show Debugged Example annotation demonstrating the unauthenticated response, the model immediately diagnosed the issue and inserted the correct token logic. However, we also note that when the model was prompted with only unrelated context, such as documentation about pagination limits, it confidently produced incorrect diagnoses *citing the unrelated context*.

This simple case shows that a system intending to leverage context to help LLMs should consider the risk of unrelated context being abused by the

model to create convincing model hallucinations, especially in the case that a correct solution is missing from the context. Determining relevance is likely much more challenging than simple "needle-in-a-haystack" retrieval, since many statements might falsely seem relevant. Proper benchmarks around this challenge are called for, with the assignment of probabilities for the relevance of individual items of context as a target.

Overall, this reflects a broader phenomenon: LLMs, like human developers, benefit from context that simulates embedded team knowledge, and shared representations are valuable; however, models can also be led astray in ways that humans usually are not, and humans also have their own limitations. While sharing context may benefit both humans and LLMs, systems like Codetations may need additional actor-specialized scaffolds around the use of that context to safely support different kinds of actors in a codebase.

5.8 Discussion

This section reflects on what we learned from building and studying Codetations and offers design takeaways for researchers and tool builders who want to treat annotations as rich and evolving artifacts rather than static comments.

5.8.1 Reflections on our Research Questions

RQ1: Utility of semantically anchored, interactive notes.

Hands-on tasks showed that developers valued (i) robust re-anchoring during edits, (ii) live executable widgets such as the Show Debugged Example annotation type, and (iii) flexible testing tools like LM Unit Test. Participants called documentation that stays next to related code a "killer feature" and could think of use cases in their own work for the interactive components we showed.

RQ2: Evolving contextual needs.

Interviews revealed that developers struggle with *documentation fragmentation*, *maintenance effort*, and the *expressiveness limits* of plain comments. Participants pointed to missing links between code and scattered artefacts while expressing concern about documentation methods that increase code clutter. For future programming systems, they envisioned an increased need for links between code and high-level documentation.

RQ3: Impact of context on LLM reasoning.

Our controlled experiments demonstrate that GPT-4o solved repair tasks reliably only when the relevant slice of annotation context was present; without it, the model offered incorrect fixes despite having the knowledge in its parameters. Thus, curated, local context still materially boosts state-of-the-

art models and can guard against confident error, but irrelevant context can also mislead models. This highlights the need for both context filtering and provenance tracking.

5.8.2 Other Key Design Takeaways

Direct integration beats side-car tooling. Pilot users struggled with an early browser prototype located outside their editor; embedding the system as a VSCode extension immediately improved comprehension and adoption. The lesson echoes decades of IDE research: if an augmentation must be consulted continuously, it needs to share the developer's focal surface.

Rapid creation is practical. Multiple powerful annotation types were generated with a single prompt once the API was demonstrated, confirming that modern LLMs make the long-standing vision of end-user-programmable IDE widgets realistic. Practically, this means that a small team can seed a rich ecosystem; the barrier is no longer engineering effort, but surfacing needs.

5.8.3 Limitations and Future Work

Our evaluation relied on a small convenience sample of nine participants drawn mainly from research environments. Industry teams with entrenched workflows may surface different pain points or adoption barriers (or offer faster paths to adoption).

The system also assumes that developers will invest effort to create annotations, although **users told us that time is their chief obstacle**. Our system might make adding context a more natural part of the development process, but using it is still far from passive; users must select text and interact with the UI to provide context. We think that addressing users' documentation fatigue will probably require a more automated approach that collects information passively as part of a normal development workflow, or even actively prompts the user (without being intrusive) to collect context on decisions.

Participants also asked for many additional features that we did not have time to build, such as annotations that span multiple code regions, possibly across files, or annotations that apply to the whole file. These would require changes in our annotation definition, but they seem otherwise feasible in the system we described.

Finally, our implementation of Codetations remains a prototype, and revision and more extensive user testing, especially with teams of users, is required to fully understand the impact of a Codetations-like system on real-world development workflows.

5.9 Conclusion

Codetations demonstrates that lightweight, LLM-aware annotations can externalize rich context, survive code evolution, and even act on their surround-

ings without modifying source files. A VSCode prototype, a qualitative user study, and targeted LLM experiments show that the approach is both feasible and valued by developers. The path forward lies in reducing the effort to add notes, scaling anchoring across languages, and exploring how active annotations can mediate ever closer human-LLM collaboration.

5.10 Appendix

5.10.1 Basic Annotation Affordances

Connecting the buffer and annotation view

When the cursor enters the region of some anchor text, an associated annotation is shown and highlighted in the annotation view; if an annotation is clicked on, the editor scrolls to the related anchor text and highlights it. Annotations are also assigned customizable colors that outline their left edge and underline the anchor text. They are also marked with their line number.

Adding and moving annotations

New annotations are added by selecting text in the file and then, in a dialog that appears in the annotation view during text selection, selecting an annotation type and clicking a button to confirm. A selected annotation can be moved to another section of the text by selecting the annotation, selecting the new anchor text, and clicking on the "Move" button in the selected annotation or on the editor's context menu.

5.10.2 Extra Implementation Details

Observing edit actions

As a VSCode extension, our host process responds immediately to user edits of the buffer by updating relevant annotation positions, as described in

Section 5.4.1. An earlier implementation used a WebSocket-based document server to monitor the disk for changes, but we found that the latency on annotation position updates using this method—essentially, waiting for the user or editor to save the document before updating underlined anchor text—was confusing for users, who expected annotations to move as they typed.

5.10.3 Show Debugged Example Annotation Type Generation Prompts

Here we show prompts used to rapidly generate the tool shown in Figure 5.1.

The prompts were given to Claude 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 5.1 except for the natural language end-user customization field at the top of the component.

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 5.1.

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.

5.10.4 LLM-Generated Code Challenges

GitHub API

Summary: As discussed in Section 5.7.1, this example demonstrates how external references can significantly improve a language model's reasoning about code. The GitHub API endpoint for listing a user's repositories returns only public repositories unless an authentication token is provided. When presented with Python code that performs an unauthenticated request, language models frequently suggest using a different endpoint—misidentifying the source of the problem. However, when supplemented with relevant API

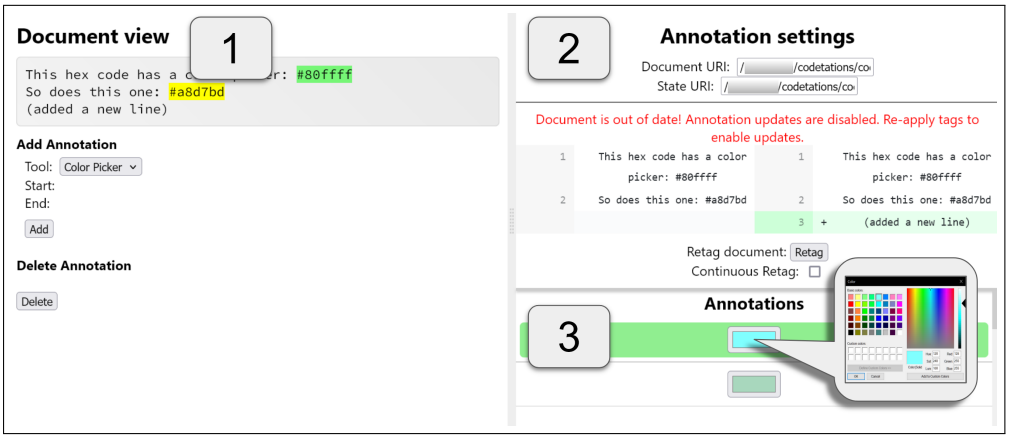


Figure 5.5: An early prototype of our system was fully editor-independent and ran in a browser. It featured (1) a document view to allow users to select annotation targets, (2) an annotation view with (user-irrelevant) configuration and document attachment information, and (3) a list of annotations. Even after explaining the system's function, users struggled to imagine workflows involving file annotations hosted in a browser separate from their editor.

documentation (alongside some extraneous, distractor context), the models accurately and efficiently diagnose the issue, correctly identifying missing authentication as the root cause.

Example code:

```
import requests


def list_repos(username):
    url = f"https://api.github.com/users/{username}/
        repos"
    response = requests.get(url)
```

```
if response.status_code == 200:
    return response.json()
else:
    raise Exception("Failed to list repositories")
```

LLM Prompt: Help debug this code. It returns no repositories for some users.

Context:

- GitHub's REST API endpoint `/users/:username/repos` only returns public repositories. Access to private repositories requires authentication via a personal access token with `repo` scope.
- API responses are paginated. By default, only 30 repositories are returned. Use `?per_page=` and pagination headers to retrieve more.
- GitHub usernames are case-insensitive, may include alphanumeric characters or hyphens, and cannot begin or end with a hyphen.
- GraphQL is also available in GitHub's API for more efficient and flexible queries.
- Code before and after debugging:

```
# Original version:
def list_repos(username):
    url = f"https://api.github.com/users/
```

```
            {username}/repos"
        response = requests.get(url)
        if response.status_code == 200:
            return response.json()
        else:
            raise Exception("Failed to list
            repositories")


    # Debugged version:
    def list_repos(username, token=None):
        url = f"https://api.github.com/users/
            {username}/repos"
        headers = {}
        if token:
            headers['Authorization'] = f'token
                {token}'
        response = requests.get(url, headers=headers)
        data = response.json()
        if response.status_code == 200:
            if isinstance(data, dict) and 'message' in
                data:
                raise Exception(f"GitHub API error:
                {data['message']}")
```

```
            return data
        else:
            raise Exception("Failed to list
            repositories")
```

- Example input: `username = "octocat"`
- Output (before): public repos only
- Output (after): public and private repos (with token)

LLM Response Without Context: Language models usually suggest using a different GitHub API endpoint or point out that the 200 result could signal a "partial success."

LLM Response With Context: Language models typically point out how authentication is required and add boilerplate code to provide the authentication.

NFC Chip with Adafruit

Summary: As mentioned in Section 5.10.13, hardware-specific constraints often reduce LLM performance when generating or reasoning about code. The PN532 NFC reader, when configured in UART mode, requires an ACK byte after every frame. When given Python code that initializes the reader in this mode without handling ACKs, language models frequently misdiagnose the issue—often inserting unnecessary sleep delays under the assumption that

the problem lies in timing. In reality, the correct fix is to either explicitly handle the ACKs or switch to high-speed UART mode, which removes the requirement altogether. When provided with PN532 documentation, models reliably identify the correct resolution. Notably, some reasoning-augmented models are able to infer the need for ACK handling even without additional context.

Example code:

```
def read_nfc_tags(nfc_uart):
    nfc_uart.write(b'\x00\x00\xFF\x02
        \xFE\xD4\x4A\x01\x00
        \xE1\x00')  # InListPassiveTarget
    response = nfc_uart.read(16)
    return response
```

LLM Prompt: Users report that sometimes the NFC reader returns partial or no data when attempting to scan tags. This occurs inconsistently, and restarting the module temporarily fixes it. Please debug this function.

Context:

- The PN532 supports multiple host interfaces including I2C, SPI, and UART. It is compliant with ISO/IEC 14443 Type A/B and FeliCa and supports peer-to-peer communication using NFC Forum standards.
- According to the PN532 user manual (UM0701), in Normal UART mode the host must send an ACK byte after every frame. If ACKs are

missing, the chip enters a discard-and-recover state. Switching to HSU mode disables this requirement and enables continuous polling without dropped frames.

- Code comparison:

```
# Without mode switch (normal UART mode):
nfc = PN532UART()
nfc.write(b'\x00\x00\xFF\x02\xFE\xD4
    \x4A\x01\x00\xE1\x00')
print(nfc.read(16))

# With HSU mode enabled:
nfc.reset()
nfc.write(b'\x00\x00\xFF\x02\xFE\xD4
    \x14\x01\x17\x00')  # HSU switch command
nfc.set_baud(115200)
nfc.write(b'\x00\x00\xFF\x02\xFE
    \xD4\x4A\x01\x00\xE1\x00')
print(nfc.read(16))
```

- Response comparison:

  - Without mode switch: `[]`

- With mode switch: `[0x00, 0x00, 0xFF, 0x03, 0xFD, 0xD5, 0x4B, 0x01, 0x00, ...]`

LLM Response Without Context: Usually, OpenAI 4o and Claude 3.5 Sonnet suggest adding a short waiting period after every 16 bytes. Reasoning models suggest sending an ACK byte after each read.

LLM Response With Context: Most types of language models switch the module to high-speed UART mode on initialization, which would solve the problem as an ACK byte is not required anymore.

5.10.5 Annotation Types Participants Wanted

Testing and Execution

- **Test Generation**: P9 was "most likely to want model generated code for" test generation, asking "Is my test coverage sufficient? What edge cases am I missing?"
- **Inline Tests**: P2, P7, and P3 all wanted inline test capabilities. P7 specifically requested "tests that are just in line... you can like run the test."
- **Input/Output Examples**: P3 wanted to "show in quotation, like, what's this input? This is the output."
- **Execution Results**: P2 requested "If you could do stuff where you could run the code to get example output, that would also be useful

for me."

- **Conditional Execution Tracking**: P9 suggested "Some way to have an annotation that's conditionally executing parts of your code and tracking the results for many different configurations."

Visual Content

- **Images**: P1 wanted "Comments, images, have a snapshot of what the code running would look like."
- **Diagrams**: P6 mentioned "I could put in diagrams and stuff."
- **Mathematical Formulas**: P7 specifically requested mathematical formula visualization: "some annotation that you highlight a line of code and indicate this is the mathematical formula that this is implementing"
- **LaTeX formulas**: P3 wanted to ensure "latex format formula matches what you prove in the code"
- **Data Structure Visualizations**: P8 desired visualizations for data structures: "if every single cargo test I had that involved an egraph had a quotation that let me explore the egraph and interact with it"
- **Tables**: P4 requested both "Rich text editor - insert tables" and "Rendering of the table that's produced by the code"

Documentation and References

- **Document-wide Annotations**: P2 suggested "Instead of having it on just some texts, you could just add like a document annotation"
- **Ownership Information**: P2 wanted "information that's document wide" including "who owns the code"
- **Research Paper Links**: P7 requested embedding research papers: "annotate the code in such a way that I can embed some... diagram or a paper or the entire paper, a snippet of the paper"
- **Data Provenance**: P7 wanted "annotations that would be sort of provenance, both in terms of what experiment loads this data, and also... where did I get these constants?"

AI-Powered Features

- **AI Suggestions**: P1 thought "AI suggestions would be cool—if there's something confusing, you hover over it, and it gives you a suggestion."
- **Code Refactoring**: P2 suggested managing code refactoring through an annotation.
- **Intent-Based Code Fixing**: P6 suggested AI could "see what [the code is] doing now and read about what you want it to do and just tweak it"

- **Pseudocode Links**: P9 envisioned "pseudocode and then that will be linked in some way to the code implementation"

Code Quality and Analysis

- **Performance/Optimization Comments**: P3 mentioned needing to explain "optimized code vs. easy-to-read code"
- **Coherency Checks**: P3 wanted "Coherency between machine proof and pen-and-paper"
- **Live Preview**: P4 praised and wanted more HTML preview functionality with live rendering capabilities
- **Side-by-side Comparisons**: P8 wanted "the original file and the produced output side by side in VS Code"

5.10.6 LLM-related User Study Issues

One participant (P5) was unable to activate GitHub Copilot, which is currently required for the VSCode Language Model API that backs several features of our system, including the offline annotation position updates. This highlights a need to include basic free or local alternative APIs as fallbacks in a full system. We note that P5 was still able to complete parts 1, 2, and 4 of the demonstration, showing that the basic system and some annotations can still operate in the absence of a language model.

Table 5.1: Participant backgrounds for our user study.

ID	Round	Background	Prog. Exp.
P1	1	Undergrad, maintains a documentation generation extension	5 years
P2	1	Senior Research Specialist, chem. engr. and synth. bio.	10 years
P3	1	Postdoc, proof engineering and system programming	18 years
P4	1	MBA, SQL development	25 years
P5	2	Senior Lecturer, user-facing formal methods and ed.; tax software	30 years
P6	1	Undergrad RA, DSL/IR research	7 years
P7	2	Grad. RA, soft. engr. & PL for computing ed.; comp. synth. bio and robotics	12 years
P8	2	Grad. RA + intern, compilers, verification; game dev.	9 years
P9	2	Grad. RA + intern, compilers, verif.; maps app, cloud security, blocks prog.	10 years

Table 5.2: Quantitative responses from round 2 of our user study. (*P7 works with computational notebooks and reported less trouble with maintaining documentation.) Our round 2 pre- and post-surveys are shown in Appendices 5.10.8 and 5.10.11.

ID	Difficulty of Doc. (1-7)	Difficulty of Doc. Sync. (1-7)	Likelihood of Using Our Tool (1-7)
P5	6	7	5
P7*	2	4	6
P8	6	5	7
P9	6	7	6

We also note that the offline position update using a language model[2] failed a couple of times during our study for the regular expression annotation. While this was unfortunate, it wasn't unexpected: Chapter 4 notes that current language models still struggle with outputting exact copies of some kinds of text, and acknowledges that the Magic Markup position update system could stand further improvement in that it requires the model to output text that matches the new annotated section exactly. We did not spend study time innovating on that basic method, since it has already improved during the course of our study with improvements in the underlying models and could be further boosted with a properly-implemented fuzzy match. We did observe that the participants presented with this failure asked if they could move the annotation manually, and were able to do so, indicating again that the system's ability to guarantee basic annotation functions even in the absence of a language model has value.

Overall, we were satisfied that basic annotation features could continue operating even when bleeding-edge features fail. While we think it's important to address these issues in future versions of Codetations, we don't view them as insurmountable problems for the idea of semantic annotation attachment.

[2]GPT-4 Turbo and GPT-4o during our study

5.10.7 User Study: Significant Quotes

User-Identified Needs

Our interviews revealed several consistent pain points in managing code-related contextual information:

Documentation Fragmentation. Participants consistently described challenges with contextual information being scattered across multiple disconnected systems:

> "Documentation in a shared drive, the documentation tends to be pretty far from the line by line parts of the code." (P6)

> "I don't think there's any way for a person looking at my code to know, 'oh, there's this Excel or Word document that you should be looking at."' (P4)

This disconnection creates significant barriers to code comprehension. P9 elaborated:

> "All of our docs were just like scattered in people's own Google drives. And so if you're looking at a piece of code and had a question about it, it was hard to know whether there existed a doc that would give some useful information." (P9)

Documentation Generation and Maintenance. Participants overwhelmingly rated keeping documentation synchronized with code changes as fairly difficult, with most rating it 5-7 on a 7-point scale:

> "Look, when you have a deadline, what's more important: the code or the documentation? I teach software engineering, and I struggle with this... Surely I'll do the documentation afterward.... Spoiler: I won't; there's another crisis." (P5)

P2 summarized: "The hardest part is just doing it. And just being consistent with it, and taking the time to do it." Several participants noted that documentation is often not prioritized in development workflows:

> "I think there are too many demands on developer time and documentation tends to be undervalued by developer teams... it's theoretically my job, but like not actually given time in the sprint plan... We'll ship the feature without it. And so we never come around and get back to the documentation." (P9)

Media Limitations. Many participants expressed frustration with the limited expressiveness of current documentation tools:

> "[Asked about problems with code comments as documentation:] I like to use the notebook [a physical notebook], because I can draw things out other than words, which I think is very helpful to understand what's going on and to communicate what's going on. So being constrained to just words." (P6)

P2 desired embedded visualizations: "If there were a way to just embed [an image] into the code... like, here's it before the function, here's it after

the function, that would be useful." P9 observed that developers attempt workarounds with ASCII diagrams, but the effort required discourages wider use:

> "I've seen people do like kind of ASCII art level diagrams as comments... I think that kind of thing would be more widely done if it were easier. Like if you could not do it as ASCII art... because ASCII art is kind of a pain... people only do it if it's like really complicated." (P9)

User Responses to the System

After interacting with our prototype, participants shared their reactions and insights about its utility and potential applications.

Value of Non-Intrusive Annotations. Participants appreciated maintaining rich contextual information alongside code without modifying the source files:

> "I feel like I could, in [Codetations], kind of like write down all these ideas and just have them around, but they're not imposing on the actual code." (P6)

P3 recognized the scalability advantage of external annotations compared to inline comments:

> "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, and that's not going to be very helpful." (P3)

P9 reflected on the distinction between inline comments and annotations:

> "A code comment feels more like it should be about the syntax as it stands in some way, like the code, not about the functionality... And then we'd want to use an annotation [in Codetations] to say something like, 'we did this for this product launch on this day,' things that are a little higher level than the code itself." (P9)

Robust Code Tracking. The system's ability to track annotations through significant code changes using reattachment based on document semantics was valued:

> "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)[3]

P1 noted the potential benefit of automatic retagging: "That would be nice if it just does that in the background without me even having to think about it."

Utility for Different User Roles. Participants identified value for various roles within software development teams:

[3]Note the participant here was imagining a future feature of our system, which only handles re-tagging within a file at present.

> "For the original code author, it does help [them] to be more aware of... to be able to communicate more efficiently about the intent of the code." (P3)

> "For the code maintainer... it's easier for them to get that code context information." (P3)

And for code testing:

> "It would be nice to see which part of the code works or doesn't work... [it] would help with the QA process." (P4)

Interactive Features and Testing. Several other participants highlighted the potential of the system's interactive capabilities.

P3's idea for illustrating code behavior were later adapted into the Show Debugged Example tool shown in Figure 5.1:

> "It would be immensely intuitive if I can just show in Codetations, like, what's this input? This is the output." (P3)

P9 later showed excitement at the prospect of debugging with the Show Debugged Example tool:

> "I think the test generation is like the thing that for me, I would be most likely to want model generated code for... generate some test inputs for this. Is my test coverage sufficient? What edge cases am I missing?" (P9)

P8 contemplated using annotations for in-context exploration of data structures: "I do work with a lot of data structures that have visualizations. So if every single cargo test I had that involved an e-graph had [an annotation] that let me explore the e-graph and interact with it or build new tests, that would be pretty cool."

Future Vision

Participants shared their perspectives on how systems like Codetations might shape the future of programming.

AI and Annotation Integration. Many participants envisioned deeper integration between annotations and AI assistance:

> "Maybe if you knew that a part of your code wasn't working how you wanted it to you could comment on it and say like this is what I describe what you want it to be doing and then the AI could like put the code into the context of the system... see what it's doing now and read about what you want it to do and just tweak it." (P6)

P9 also envisioned a shift in programming abstraction:

> "Maybe people will write pseudocode and then that will be linked in some way to the code implementation of the pseudocode... the model is going to generate the actual code that runs in production. But the thing that we're all looking at and talking about is

> a pseudocode model of it. And then we need links between the various points." (P9)

Domain Knowledge Representation. Participants recognized the growing importance of connecting domain expertise with code:

> "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)

P9 noted the specific value for AI-generated code:

> "I think the future of programming includes more and more computer generated code... in that world, sorting through the noise will become harder if you're just like more and more low to medium quality code, or like, at least unknowable quality code... that seems like a place where annotation can be useful is like, here's a pile of code. And then here's some evidence that there's a person who has thought about it." (P9)

5.10.8 Pre-demo Surveys

Participants were asked the following questions in our pre-demo survey. The questions mainly served as a starting point for discussion, and most participants skipped one or more questions due to time constraints.

Round 1

Question: What is your education level and job title (if applicable)?

Question: How many years of programming experience do you have?

Question: Briefly, in what kinds of positions or research domains have you worked as a programmer?

Question: What is your primary development environment (e.g., VSCode, IntelliJ, other)?

Question: Do you have any experience using Copilot or another code generation/programming assistance platform?

Question: How do you currently maintain code documentation?

Question: When working with a team, how do you handle sharing and storing contextual information about code?

Question: How do you currently handle attaching supplementary information (like examples, visualizations, or analysis results) to specific sections of code?

Question: What kinds of problems have you faced with code documentation systems? For example, have you faced challenges in keeping documentation synchronized with code changes?

Round 2

Question: Education level + job titles if applicable

Question: Years of programming experience

Question: Briefly, in what kinds of positions or research domains have you worked as a programmer?

Question: Primary development environment (VSCode/IntelliJ/other):

Question: Do you have any experience using Copilot or another code generation/programming assistance platform? If so, when prompting a language model to assist with your work, what kinds of information do you usually provide?

Question: Please skim this list of types and examples of contextual information that developers may use while working with code [Appendix 5.10.9]. Briefly, how do you currently maintain code documentation and contextual information for projects that you work on?

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?

Question: What specific problems have you faced with maintaining code documentation and contextual information?

Question: Describe a situation where better code annotations would have been valuable to you or your team.

Question: Do you currently attach long-form contextual information (e.g., examples, tutorials, notes, diagrams, or videos) to specific code sections? If so, how? If not, why not?

5.10.9 Types of Development Context

The following lists were provided to participants in round 2 of our user study to help them reflect on their current context maintenance processes.

- **General types**
  - **Design & Architecture Information**
    - Architecture diagrams and system flow charts
    - Design patterns used and their rationale
    - Component relationships and dependencies
    - System constraints and design trade-offs
    - API contracts and interface specifications
    - Architectural decisions and their justifications
  - **Historical & Process Information**
    - Reasons for implementation choices
    - Alternative approaches considered and rejected
    - Change history beyond version control commits
    - Stakeholder requirements that influenced the code
    - Evolution of the code over time
    - Original author's intent and mental model
  - **Performance & Optimization Context**
    - Performance characteristics and benchmarks

    - Profiling data and bottlenecks
    - Memory usage patterns
    - Optimization opportunities
    - Resource constraints
    - Performance regressions over time
  - **Business & Domain Knowledge**
    - Business rules encoded in the algorithm
    - Domain-specific terminology explanations
    - Regulatory requirements implemented
    - Business process mappings
    - Customer use cases addressed
    - Domain expert insights
  - **Testing & Quality Context**
    - Test coverage information
    - Known edge cases
    - Difficult-to-test scenarios
    - Bug history and previous failures
    - Validation approaches
    - Quality metrics
  - **Runtime & Environmental Context**
    - Expected input/output examples

    - Runtime behavior visualizations
    - Environment dependencies
    - Configuration requirements
    - Deployment constraints
    - Platform-specific considerations
- **Security & Compliance Information**
    - Security considerations
    - Privacy implications
    - Compliance requirements
    - Authentication/authorization context
    - Data sensitivity information
    - Security review history
- **Maintenance & Support Context**
    - Known issues and workarounds
    - Debugging tips
    - Common maintenance tasks
    - Support ticket references
    - Troubleshooting guides
    - Upgrade/migration considerations
- **Educational & Onboarding Context**
    - Learning resources for new developers

    - Step-by-step explanations
    - Code walkthroughs
    - Glossary of terms
    - Interactive examples
    - Conceptual models
- **Collaborative Context**
    - Code review comments and discussion
    - Team member questions and answers
    - Knowledge transfer notes
    - Pair programming insights
    - Cross-team coordination information
    - Meeting notes relevant to the code
- **Future Development Context**
    - Planned enhancements
    - Technical debt notes
    - Roadmap items
    - Future-proofing considerations
    - Extensibility points
    - Deprecation plans
- **Visual Context**
    - Data flow visualizations

        * UI component renderings
        * Color mappings and design assets
        * Algorithm visualizations
        * State machine diagrams
        * Timeline visualizations
    - **External Relationships**
        * Links to external documentation
        * References to research papers or standards
        * Connections to ticketing systems
        * References to related codebases
        * External API documentation
        * Third-party library usage context

- **Examples in Real-World Software Development**
    - **Comments & Documentation**
        * Code comments explaining complex logic or edge cases
        * Function and class documentation
        * README files and getting started guides
        * API documentation and usage examples
    - **"Outside the Code" References**
        * Stack Overflow answers bookmarked or copied

    - Google/web search results you relied on
    - Blog posts that helped solve a problem
    - YouTube tutorials or conference talks
    - Personal notes in text files or notebooks
  - **Visual Thinking**
    - Whiteboard sketches you took photos of
    - Diagrams drawn on paper or in tools like Miro
    - Screenshots of working (or broken) features
    - Napkin drawings that never made it into docs
    - UI mockups and wireframes
  - **Knowledge from Others**
    - Slack/Discord/Teams conversations with teammates
    - Email threads explaining design decisions
    - Code review comments that explained rationales
    - Tribal knowledge shared verbally in meetings
    - Pairing session insights
  - **Project Management Artifacts**
    - Jira/Trello ticket details and requirements
    - Pull request descriptions and discussions
    - Sprint planning documents
    - User stories and acceptance criteria

    - Bug reports and their symptoms

- **The "Why" Behind Code**
    - Reasons for weird-looking code ("this looks strange but...")
    - Business requirements that dictated unusual logic
    - Workarounds for bugs in dependencies
    - Performance hacks and their rationale
    - Security considerations that affected design
- **Testing Context**
    - Test cases that drove implementation choices
    - Edge cases discovered during testing
    - Difficult-to-reproduce bugs
    - User feedback that influenced the code
    - Performance testing results
- **"Remember This" Information**
    - TODOs and FIXMEs
    - Gotchas and pitfalls to watch out for
    - Areas known to be fragile or in need of refactoring
    - Reminders about deployment considerations
    - Configuration requirements
- **Environment & Runtime Details**
    - Local setup instructions

    - Environment variables needed
    - Database schema and relationships
    - API endpoints and auth requirements
    - Infrastructure dependencies
- **Historical Context**
    - "Why did we build it this way?"
    - Previous approaches that failed
    - Migration paths from legacy systems
    - Compatibility requirements
    - Technical debt explanations
- **Future Plans**
    - Planned refactoring ideas
    - Feature roadmap items
    - Performance improvement possibilities
    - Scaling considerations for the future
    - Ideas for better approaches
- **Examples & Sample Data**
    - Sample inputs and expected outputs
    - Test data sets
    - Example configurations
    - Demo scenarios

- User journey examples

5.10.10 Demonstration Tasks

During the round 2 demo, participants were shown the following material and asked to complete annotation tasks:

1. An HTML file pre-annotated with a basic comment annotation type and another type that rendered a preview of a section of the HTML. All round 2 participants quickly confirmed at this stage that they understood what the annotations meant and how they worked.

2. Another HTML file, where all round 2 participants quickly showed that they could add an annotation to the file without interviewer help.

3. A JavaScript quicksort algorithm annotated with an animated GIF visualizing quicksort; no task was assigned for this example.

4. A JavaScript regular expression (shown in Figure 5.1 on the left); participants were asked to add and use a Show Debugged Example annotation (see Section 5.6.1) to validate the regex. The extension's on- and offline annotation position updating capabilities were also demonstrated at this point.

5. A Python program including an expression with floating-point error. Participants were asked to analyze the error and find a better expression

using an *Analyze Floating Point Expression* annotation type (Section 5.6.2).

6. A JavaScript loop body with an incorrect comment; participants were asked to add a unit test for the comment (see Section 5.6.2).

5.10.11 Post-demo Surveys

Participants were asked the following questions in our post-demo survey. The questions mainly served as a starting point for discussion, and most participants skipped one or more questions due to time constraints.

Round 1.

Question: How do the features currently provided by this system compare to your current tools for code documentation? Are there particular projects you have worked on where you would like to have used the features of a tool like Codetations?

Question: What features and guarantees would the system have to provide before you would recommend the system to a team you were working with?

Question: [Discuss only] In the future, how would you expect this system to integrate with:

- AI code generation
- Version control

- Code review processes
- Team collaboration
- Continuous integration
- Other

Question: Orphaned annotations are annotations that no longer fit with the current document. In your opinion, how should the system handle re-attaching annotations that don't quite fit with the new document? Consider domains you have worked in and the types of changes to documents.

Question: For each annotation type you saw, was it useful as is or would it need more features to be useful? How difficult was it to understand the use of that annotation type?

Question: Which annotation types would you use in your daily work? Why?

Question: Can you imagine additional annotation types? Would you use any of these in your own work?

Question: [comment briefly for each] How do you imagine using a tool like Codetations as someone working in the following roles:

- Original code author
- Code maintainer
- Tester

- Product manager
- API/library user
- Junior programmer
- Other roles you have worked in

Question: How easy would you say it was to learn to use Codetations to annotate code? (1-7) Question: Discuss your idea of the "future of programming". Can you think of ways that annotations and annotation systems might have a role in this vision?

Round 2.

Question: In what ways does this system address pain points in your current processes for working with contextual information? If needed, you can refer to the document on types of contextual information here.

Question: In what ways does this system NOT address pain points in your current processes for working with contextual information?

Question: How do the features of Codetations compare to your current documentation tools? In particular, compare Codetations to typical code comments.

Question: Which annotation types seemed most useful for workflows you are familiar with?

Question: Are there specific projects or tasks where you would use a tool like Codetations?

Question: After seeing the demo, can you identify a recent situation where contextual information would have improved your work, and explain how this tool might host that information?

Question: What features would the system need before you'd recommend it to your team or use it personally? Are there any basic reasons you would hesitate to use it?

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?

5.10.12 Other Applications

Recording Structured LLM Interactions

LLMs have increasingly become tools of choice for low-code programming, aiding developers of various skill levels in the rapid development of software by generating code snippets from high-level descriptions. However, observations by Liu et al. [LCB+24] highlight user concerns that LLMs may introduce variability in the correctness and quality of the generated code. Bird et al. further note the establishment of code provenance as a key issue in systems that use generated code [BFZ+22], i.e., when code is generated in a chat interface, the conversation that led to the code, including valuable specification data, is typically not attached to the codebase with the resulting code.

We could easily foresee simply recording unstructured chats with an LLM

as annotations, already improving upon current practices. This workflow could be extended to any other tool that generates or validates code. For example, the user's work in Ferdowsi et al.'s LEAP system [FJP+23], which lets users validate AI-generated code with live feedback on the line-level behavior of generated code, could be preserved in a similar way.

5.10.13 More Worked Examples

Here we include other worked examples exploring the effects of additional context on an LLM's program repair performance for debugging problems generated by the model itself.

Real-World Assumptions

Many programming errors stem not from faulty logic within the code itself, but from real-world constraints that lie outside the immediate scope of the code snippet. These constraints — whether hardware-specific quirks, environmental dependencies, or undocumented design choices — are often tacitly understood by human teams but remain opaque to LLMs. Just as human programmers rely on experience and tribal knowledge to navigate such issues, LLMs require similar cues to surface relevant latent knowledge.

Consider the NXP PN532 NFC chip, widely used via Adafruit breakout boards. When configured in UART Normal Mode, the chip expects an ACK byte from the host after every frame. By default, the Adafruit library disables this handshake, leading to intermittent data loss; this is a subtle bug that

eludes detection unless agents are familiar with the underlying hardware protocol. A correct fix involves switching the chip into High-Speed UART (HSU) mode, which bypasses the need for ACKs entirely.

When prompted with a three-line NFC reading function and a vague error report describing occasionally dropped data, GPT-4o responded by dramatically expanding the code: introducing delays, buffer resets, and packet-length guards. Despite these elaborate modifications, the bug remained. However, when the same code was presented alongside data from a Codetations Show Debugged Example tool demonstrating the failure and paired with distractor documentation about the PN532 chip, the model rapidly identified the true issue and recommended the correct configuration change.

Chapter 6

Conclusion

This dissertation has explored a fundamental challenge in software development: how to maintain rich contextual information alongside evolving code. I introduced and defended the thesis that *programming environments can and should now provide semantically anchored, persistent contextual information to enhance program understanding and manipulation for both human developers and AI agents.* Through three interconnected systems—Odyssey, Magic Markup, and Codetations—this work has demonstrated both the technical feasibility of context-enriched programming environments and their tangible benefits.

6.1 Technical Feasibility

As we saw in Chapter 4, programming environments *can* indeed provide semantically anchored, persistent contextual information. The technical foundations established through Magic Markup's approach to semantic anchoring show that it is possible to robustly maintain connections between code and context even as code undergoes significant changes. Codetations further proved that such anchoring can be implemented (albeit as a prototype) in a popular, general-purpose programming environment and extended to support rich, interactive annotations.

The hybrid approach of using editor-based tracking for online edits and LLM-based reanchoring for offline changes provides a practical solution to the longstanding challenge of annotation persistence. Moreover, the annotation type API developed for Codetations shows that these annotations can become far more than static text—they can include dynamic, interactive tools that respond to code changes and provide live feedback.

6.2 Demonstrated Benefits

Beyond technical feasibility, this work has shown why programming environments *should* provide such contextual information. Through user studies and worked examples, I've demonstrated benefits for both human developers and AI agents:

For human developers, the ability to maintain rich, persistent context

alongside code addresses key pain points in current development practices. Participants in our Codetations study highlighted the value of reducing documentation fragmentation, improving media expressiveness, and maintaining robust connections between code and context. The non-intrusive nature of external annotations was particularly valued, as it allowed developers to maintain more contextual information without cluttering the codebase itself.

For AI agents, the worked examples with LLMs showed that contextual information significantly improves their reasoning about code. Even when the underlying model should theoretically have the knowledge to solve a problem, explicit context dramatically improved performance on code repair tasks. This suggests that as AI becomes increasingly involved in software development, contextual information will be crucial for effective human-AI collaboration.

6.3 Broader Implications

The findings from this research have significant implications for the future of programming environments and software development practices.

First, this work suggests a fundamental shift in how we think about code documentation. Rather than treating documentation as either embedded comments or entirely separate documents, we can envision a continuous spectrum of contextual information that maintains semantic connections to code while living outside it. This approach combines the best aspects of both

paradigms—the tight coupling of comments with the rich expressiveness of external documentation.

Second, as programming becomes increasingly collaborative between humans and AI agents, shared representations of context become even more critical. Systems like Codetations can serve as a bridge, allowing both humans and AI to contribute to and benefit from the same contextual information. This is especially important as AI tools like GitHub Copilot become more integrated into development workflows.

Third, this research points toward a new paradigm of programming environments where the environment itself becomes "context-aware." Rather than simply being a text editor for code, the environment can understand the semantic structure of the code and maintain connections to relevant contextual information, presenting it when and where it's needed.

6.4 Future Directions

While this dissertation has established important foundations, several promising directions for future work remain:

Near-term extensions could focus on improving the robustness and performance of semantic anchoring techniques, particularly for complex code transformations like refactorings that span multiple files. The anchoring method developed in Magic Markup could be extended to handle a wider range of document types and transformations, and fine-tuned language models could

make the process faster and more reliable. [1]

Medium-term research could explore new types of context-aware programming tools built on this foundation. For example, future work could develop specialized annotation types for particular domains or development practices, such as security auditing, performance optimization, or accessibility compliance. The ability to rapidly create new annotation types using LLMs, demonstrated in Codetations, suggests a rich ecosystem of specialized tools is possible.

Another medium-term extension would be transitioning from file-based annotation storage to real-time database backends, enabling truly collaborative contextual programming. While Codetations currently stores annotations in repository files, a persistent database system with conflict resolution would allow multiple developers to simultaneously create, view, and modify annotations with immediate synchronization. Such a system could track annotation history separately from code history, implement granular permission models for different annotation types, and support ephemeral "working annotations" that exist only during specific development sessions. This approach would also enable richer cross-repository annotation linking, where context from one codebase could inform work in dependent projects. Implementing this extension would require solving challenges around authentication, offline support, and the seamless integration of local and remote annotations,

[1] At the time of publication, an LLM that can handle retagging at roughly the same quality as the SOTA LLM used in the original Magic Markup publication 1 year prior, without any fine-tuning, runs on an 8-core CPU, though still about ten times slower.

but would significantly enhance Codetations' value for team-based software development where contextual knowledge sharing is most critical.

Longer-term, this work opens up questions about how programming environments might evolve to more deeply integrate contextual information. Could future IDEs automatically suggest relevant contextual information based on what a developer is currently working on? Could they help collect and organize context during the development process, rather than requiring explicit annotation? Could they adapt the presentation of context based on a developer's expertise level or current task?

6.5 Conclusion

This dissertation has shown that programming environments can and should now provide semantically anchored, persistent contextual information. The three systems developed — Odyssey, Magic Markup, and Codetations — demonstrate both the technical feasibility and tangible benefits of such an approach for human developers and AI agents alike.

As software continues to grow in complexity and new programmers seek to take on existing projects—large parts of which may now have been generated without any detailed human consideration whatsoever—I predict that the gap between code and its context, if left open, will become an increasingly significant barrier to effective software development and maintenance. By bridging this gap through intelligent, context-aware programming envi-

ronments, we can enable more effective and accessible programming, improve code quality, and facilitate safer, more transparent collaboration between humans and AI in the software development process.

Bibliography

[ABDF07] Maristella Agosti, Giorgetta Bonfiglio-Dosio, and Nicola Ferro. A historical and contemporary study on annotations to derive key features for systems design. *International Journal on Digital Libraries*, 8:1–19, 2007.

[AF07] Maristella Agosti and Nicola Ferro. A formal model of annotations of digital content. *ACM Transactions on Information Systems (TOIS)*, 26(1):3–es, 2007.

[AGM03] Micah Altman, Jeff Gill, and Michael P. McDonald. *Numerical Issues in Statistical Computing for the Social Scientist*. Springer-Verlag, 2003.

[AM03] Micah Altman and Michael P. McDonald. Replication with attention to numerical accuracy. *Political Analysis*, 11(3):302–307, 2003.

[App11] Andrew W Appel. Verified software toolchain: (invited talk). In *European Symposium on Programming*, pages 1–17. Springer, 2011.

[BBGC01] AJ Bernheim Brush, David Bargeron, Anoop Gupta, and Jonathan J Cadiz. Robust annotation positioning in digital documents. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 285–292, 2001.

[BBKE13] Brian Burg, Richard Bailey, Amy J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484, 2013.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining Coq and Gappa for certifying floating-point programs. In *International Conference on Intelligent Computer Mathematics*, pages 59–74. Springer, 2009.

[BFZ+22] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, 20(6):35–57, 2022.

[BHH12] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 453–462, New York, NY, USA, 2012. ACM.

[BLC90] Tim Berners-Lee and R. Cailliau. Worldwideweb: Proposal for a hypertext project, 1990.

[BLC18] Antranig Basman, Clayton Lewis, and Colin Clark. The open authorial principle: supporting networks of authors in creating externalisable designs. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 29–43, 2018.

[BMDR13] Fabian Beck, Oliver Moseler, Stephan Diehl, and Günter Daniel Rey. In situ understanding of performance bottlenecks through visually augmented code. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 63–72. IEEE, 2013.

[BVLS13] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. POPL '13, 2013.

[CCCJ07] Guillaume Cabanac, Max Chevalier, Claude Chrisment, and Christine Julien. Collective annotation: Perspectives for information retrieval improvement. In *Large Scale Semantic Access to Content (Text, Image, Video, and Sound)*, pages 529–548. 2007.

[CGRS14] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Efficient search for inputs causing high floating-point errors. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 43–52, New York, NY, USA, 2014. Association for Computing Machinery.

[CLB+18] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C Gall. Performancehat: augmenting source code with runtime performance traces in the ide. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 41–44, 2018.

[CMM02] Michael L Collard, Jonathan I Maletic, and Andrian Marcus. Supporting document and data views of source code. In *Proceedings of the 2002 ACM symposium on Document engineering*, pages 34–41, 2002.

[CRDB15] Mihai Codoban, Sruti Srinivasa Ragavan, Danny Dig, and Brian Bailey. Software history under the lens: A study on why and how developers examine it. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–10. IEEE, 2015.

[DBG+20] Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, Sriram Krishnamoorthy, and Pavel Panchekha. Scalable yet rigorous floating-point error analysis. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–14, Los Alamitos, CA, USA, nov 2020. IEEE Computer Society.

[DER10] Ekwa Duala-Ekoko and Martin P Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(1):1–31, 2010.

[dJ13] Jos de Jong. math.js: An extensive math library for JavaScript and Node.js, 2013.

[DK14] Eva Darulova and Viktor Kuncak. Sound compilation of reals. POPL, 2014.

[DM17] Nasrine Damouche and Matthieu Martel. Salsa: An automatic tool to improve the numerical accuracy of programs. AFM, 2017.

[dt] Cursor development team. Cursor – rules.

[Eur98] European Commission. *The introduction of the euro and the rounding of currency amounts*. Euro papers. European Commission, Directorate General II Economic and Financial Affairs, 1998.

[FJP+23] Kasra Ferdowsi, Michael B James, Nadia Polikarpova, Sorin Lerner, et al. Live exploration of ai-generated programs. *arXiv preprint arXiv:2306.09541*, 2023.

[FL79] Michael J Fischer and Richard E Ladner. *Data structures for efficient implementation of sticky pointers in text editors*. Department of Computer Science, University of Washington, 1979.

[GAK23] Fabrizio Gilardi, Meysam Alizadeh, and Maël Kubli. Chatgpt outperforms crowd workers for text-annotation tasks. *Proceedings of the National Academy of Sciences*, 120(30):e2305016120, 2023.

[GBL23] Camille Gobert and Michel Beaudouin-Lafon. Lorgnette: Creating malleable code projections. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–16, 2023.

[GJA23] Ken Gu, Eunice Jun, and Tim Althoff. Understanding and supporting debugging workflows in multiverse analysis. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–19, 2023.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

[GP11] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. VMCAI'11, pages 232–247, 2011.

[Ham87] Richard Hamming. *Numerical Methods for Scientists and Engineers*. Dover Publications, 2nd edition, 1987.

[Han06] Frank Allan Hansen. Ubiquitous annotation systems: technologies and challenges. In *Proceedings of the seventeenth conference on Hypertext and hypermedia*, pages 121–132, 2006.

[HH23a] Joshua Horowitz and Jeffrey Heer. Engraft: An api for live, rich, and composable programming. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–18, 2023.

[HH23b] Joshua Horowitz and Jeffrey Heer. Live, rich, and composable: Qualities for programming beyond static text. *arXiv preprint arXiv:2303.06777*, 2023.

[HHB+19] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[HHC+24] Devamardeep Hayatpur, Brian Hempel, Kathy Chen, William Duan, Philip Guo, and Haijun Xia. Taking ascii drawings seriously: How programmers diagram code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–16, 2024.

[HLC19] Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-sketch: Output-directed programming for svg. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 281–292, 2019.

[HLHF23] Ava Heinonen, Bettina Lehtelä, Arto Hellas, and Fabian Fagerholm. Synthesizing research on programmers' mental models of programs, tasks and concepts—a systematic literature review. *Information and Software Technology*, 164:107300, 2023.

[HMM23] Amber Horvath, Andrew Macvean, and Brad A Myers. Support for long-form documentation authoring and maintenance. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 109–114. IEEE, 2023.

[HMM24] Amber Horvath, Andrew Macvean, and Brad A Myers. Metamanager: A tool for collecting and exploring meta information about code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–17, 2024.

[HMMR22] Amber Horvath, Brad Myers, Andrew Macvean, and Imtiaz Rahman. Using annotations for sensemaking about code. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, pages 1–16, 2022.

[HSH18] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Augmenting code with in situ visualizations to aid program understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2018.

[HVW07] Jeffrey Heer, Fernanda B Viégas, and Martin Wattenberg. Voyagers and voyeurs: supporting asynchronous collaborative information visualization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1029–1038, 2007.

[ID17] Anastasiia Izycheva and Eva Darulova. On sound relative error bounds for floating-point arithmetic. FMCAD, pages 15–22, 2017.

[IEE08] IEEE. IEEE standard for binary floating-point arithmetic. *IEEE Std. 754-2008*, 2008.

[IRKF+21] Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. Guided optimization for image processing pipelines. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–5. IEEE, 2021.

[Juh19] Ján Juhár. Supporting source code annotations with metadata-aware development environment. In *2019 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 411–420. IEEE, 2019.

[KHM17] Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, volume 10, pages 3–025, 2017.

[KM04] Amy J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, 2004.

[KM18] Mary Beth Kery and Brad A Myers. Interactions for untangling messy history in a computational notebook. In *2018 IEEE symposium on visual languages and human-centric computing (VL/HCC)*, pages 147–155. IEEE, 2018.

[Kne17] Ronald T. Kneusel. *Numbers and Computers*. Springer Cham, 2017.

[Knu84] Donald Ervin Knuth. Literate programming. *The computer journal*, 27(2):97–111, 1984.

[KRKP+16] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. Jupyter notebooks–a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*, pages 87–90. IOS press, 2016.

[KSRC09] Sayali Kulkarni, Amit Singh, Ganesh Ramakrishnan, and Soumen Chakrabarti. Collective annotation of wikipedia entities in web text. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 457–466, 2009.

[L+66] Vladimir I Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[LAK+23] Jenny T Liang, Maryam Arab, Minhyuk Ko, Amy J Ko, and Thomas D LaToza. A qualitative study on the implementation design decisions of developers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 435–447. IEEE, 2023.

[LBM14] Tom Lieber, Joel R Brandt, and Rob C Miller. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2481–2490, 2014.

[LCB+24] Yongkun Liu, Jiachi Chen, Tingting Bi, John Grundy, Yanlin Wang, Ting Chen, Yutian Tang, and Zibin Zheng. An empirical study on low code programming using traditional vs large language model support, 2024.

[LDMG20] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. 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)*, pages 1–11. IEEE, 2020.

[Ler20] Sorin Lerner. Projection boxes: On-the-fly reconfigurable visualization for live programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–7, 2020.

[LKAH20] Yang Liu, Alex Kale, Tim Althoff, and Jeffrey Heer. Boba: Authoring and visualizing multiverse analyses. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):1753–1763, 2020.

[LVD06] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006.

[LYUI20] Maria Larsson, Hironori Yoshida, Nobuyuki Umetani, and Takeo Igarashi. Tsugite: Interactive design and fabrication of wood joints. In *UIST*, pages 317–327, 2020.

[MCS+23] Edward Misback, Caleb C Chan, Brett Saiki, Eunice Jun, Zachary Tatlock, and Pavel Panchekha. 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*, pages 1–15, 2023.

[MD23] Paolo Masci and Aaron Dutle. Vscode-precisa, 2023.

[Mic24] Microsoft. Visual studio code language model api, July 2024. First released in the July 2024 (version 1.92) release of Visual Studio Code.

[MIT23] Ariana Martino, Michael Iannelli, and Coleen Truong. Knowledge injection to counter large language model (llm) hallucination. In *European Semantic Web Conference*, pages 182–185. Springer, 2023.

[MLAC20] Glaucia Melo, Edith Law, Paulo Alencar, and Don Cowan. Exploring context-aware conversational agents in software development. *arXiv preprint arXiv:2006.02370*, 2020.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):12:1–12:41, May 2008.

[MTT24] Edward Misback, Zachary Tatlock, and Steven L. Tanimoto. Magic markup: Maintaining document-external markup with an llm. *ArXiv*, abs/2403.03481, 2024.

[MV99] B. D. McCullough and H. D. Vinod. The numerical reliability of econometric software. *Journal of Economic Literature*, 37(2):633–665, 1999.

[MVTT25] Edward Misback, Erik Vank, Zachary Tatlock, and Steven Tanimoto. Codetations: Intelligent, persistent notes and uis for programs and other documents. *arXiv preprint arXiv:2504.18702*, 2025.

[NRJ12] Elena Novak, Rim Razzouk, and Tristan E Johnson. The educational use of social annotation tools in higher education: A literature review. *The Internet and Higher Education*, 15(1):39–49, 2012.

[OAM99] Ilia A Ovsiannikov, Michael A Arbib, and Thomas H McNeill. Annotation technology. *International journal of human-computer studies*, 50(4):329–362, 1999.

[OM09] Stephen Oney and Brad Myers. Firecrystal: Understanding interactive behaviors in dynamic web pages. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 105–108. IEEE, 2009.

[Pan22] Pavel Panchekha. Improving rust with herbie. `https://pavpanchekha.com/blog/herbie-rust.html`, November 2022.

[PSSWT15] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. PLDI, 2015.

[Qui83] Kevin Quinn. Ever had problems rounding off figures? This stock exchange has. *The Wall Street Journal*, page 37, November 8, 1983.

[RBS13] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE software*, 7(4):57–66, 1990.

[Rei08] Steven P Reiss. Tracking source locations. In *Proceedings of the 30th international conference on Software engineering*, pages 11–20, 2008.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and

recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[RRR+19] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. Babylonian-style programming: Design and implementation of an integration of live examples into general-purpose source code. *arXiv preprint arXiv:1902.00549*, 2019.

[SBLH06] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The semantic web revisited. *IEEE intelligent systems*, 21(3):96–101, 2006.

[SFN+21] Brett Saiki, Oliver Flatt, Chandrakana Nandi, Pavel Panchekha, and Zachary Tatlock. Combining precision tuning and rewriting. In *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, 2021.

[SHY+23] Chunmei Sun, Gwo-Jen Hwang, Zhaoyi Yin, Zhonghou Wang, and Zhuo Wang. Trends and issues of social annotation in education: A systematic review from 2000 to 2020. *Journal of Computer Assisted Learning*, 39(2):329–350, 2023.

[SJRG15] Alexey Solovyev, Charlie Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. FM, 2015.

[SP13] Kamran Sedig and Paul Parsons. Interaction design for complex cognitive activities with visual representations: A pattern-based approach. *AIS Transactions on Human-Computer Interaction*, 5(2):84–133, 2013.

[SRS+09] Margaret-Anne Storey, Jody Ryall, Janice Singer, Del Myers, Li-Te Cheng, and Michael Muller. How software developers use tagging to support reminding and refinding. *IEEE Transactions on software engineering*, 35(4):470–483, 2009.

[Tan13] Steven L Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34. IEEE, 2013.

[Tea] The Herbie Development Team. Herbie: Optimize floating-point expressions for accuracy, 2013–.

[TFMM18] Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Munoz. An abstract interpretation framework for the round-off error analysis of floating-point programs. VMCAI, pages 516–537, 2018.

[TR81] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[U.S92] U.S. General Accounting Office. Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia, 1992.

[WCB16] Moritz Wittenhagen, Christian Cherek, and Jan Borchers. Chronicler: Interactive exploration of source code history. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, pages 3522–3532, 2016.

[WERC+07] Andrew Walenstein, Mohammad El-Ramly, James R Cordy, William S Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[WN01] Joanna L Wolfe and Christine M Neuwirth. From the margins to the center: The future of annotation. *Journal of Business and Technical Communication*, 15(3):333–371, 2001.

[Wol02] Joanna Wolfe. Annotation technologies: A software and research review. *Computers and Composition*, 19(4):471–497, 2002.

[WQ22] Junyi Wang and Yue Qi. A multi-user collaborative ar system for industrial applications. *Sensors*, 22(4):1319, 2022.

[WW92] Debora Weber-Wulff. Rounding error changes parliament makeup, 1992.

[YM15] YoungSeok Yoon and Brad A Myers. Supporting selective undo in a code editor. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 223–233. IEEE, 2015.