Meta-Information to Support Sensemaking by Developers

Amber Horvath

CMU-HCII-...

August 2024

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Committee

Brad A. Myers, Chair
Laura Dabbish
Aniket Kittur
Elena Glassman, (Harvard University)
Andrew Macvean, (Google Inc.)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Human-Computer Interaction.

This work was supported by the National Science Foundation, under NSF grant CCF-2007482, and by Google. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation or Google.

Keywords: Human-Computer Interaction, Software Engineering, Sensemaking, Code Comprehension, Annotation, Developer Tools, Meta-Information

Abstract

Software development requires developers to juggle and balance many information-seeking and understanding tasks. From understanding how a bug was introduced, to choosing what application programming interface (API) method to use to resolve the bug, to determining how to properly integrate this change, even the smallest implementation tasks can lead to many questions. These questions may range from hard-to-answer questions about the rationale behind the original code to common questions such as how to use an API. Software development, in contrast to other sensemaking domains, has the unique property that many information artifacts are created at different points during the development process (e.g., output data). Once this challenging sensemaking is done, this rich thought history is often lost given the high cost of externalizing these details, despite potentially being useful to future developers.

In this thesis, I explore different systems and methods for authoring and using this rich thought history as *meta-information* about code. Specifically, I have developed systems for *annotating* to support developers' natural sensemaking when understanding information-dense sources such as software documentation and source code. I then demonstrated how this meta-information can be captured and harnessed for new tasks, including for assessing the trustworthiness of documentation, for capturing design rationale and provenance data of code, and for supporting developer implementation tasks such as debugging.

This thesis begins by exploring methods for externalizing developers' thoughts in a form that is lightweight yet contextualized. We explore annotating as a method for simultaneously allowing developer's to offload their mental processes, while presenting that information in-context for later developers to utilize. We developed 2 prototype annotation systems, Adamite and Catseye, which showed the promise of annotating for assisting developers both in overcoming issues with using developer documentation and debugging code. The dynamic nature of code and its connection to annotated materials introduced unique design challenges in which information can quickly become outdated and disconnected, leading both to changes in the Catseye user interface and to the insight that leveraging the (lack of) connectivity between annotation and documentation can support other documentation-related tasks, which inspired Sodalite. The final two systems, Meta-Manager and MMAI, explore capturing other forms of already-authored meta-information, such as edit traces and log data, for question-answering support, with MMAI utilizing large language models to make that querying possible in natural language.

The series of work introduced in this thesis points to the need to treat user-thought and intent as a first-class entity and that meta-information is a way of presenting that information. I show that developers' thought histories can be represented in the form of code-related meta-information and, through proper tooling, can be used by later developers to accelerate their sensemaking of code.

Acknowledgements

Completing this dissertation was a marathon, not a sprint. And, during this marathon, I have had the tremendous opportunity to grow as a researcher, thanks in part to the endless support from my network of friends, family, academic and industry partners, and collaborators.

Firstly, I would not be here without my amazing advisor, Brad Myers. Brad has been with me through this whole journey, fostering my academic growth through all of the ups and downs that come with the completing a PhD. There were times when I wanted to quit, but his belief, support, and resilience, even when I could not muster up such feelings, kept me here and allowed me to reach this point – this achievement is as much his as it is mine. I am proud to be a node on his famous advisee tree.

I am also very fortunate to have a large thesis committee of brilliant researchers to further assist in my academic growth: Laura Dabbish, Aniket (Niki) Kittur, Elena Glassman, and Andrew Macvean. Their varied expertise has lead to thoughtful questions and lines of inquiry I would not have otherwise considered. They, along with Brad, have acted as prime models of how to be an excellent researcher and collaborator – lessons I will take with me moving forward.

Beyond Brad and my committee members, I have been privileged over the course of my PhD to have opportunities to receive mentorship from the wider HCI, Software Engineering, and Carnegie Mellon University communities. I would like to thank Margaret Burnett who advised me at Oregon State University and introduced me to this wonderful, chaotic world of academic research I am now a part of – her mentorship has been invaluable. I would also like to thank Emerson Murphy-Hill, my internship host at Google in 2019, who has continued to provide me support and guidance over the last few years. Special thanks also to Andrew Begel, Patrick Carrington, Jess Hammer, Ken Holstein, Geoff Kaufmann, Queenie Kravitz, Bogdan Vasilescu, and countless others with whom I had the pleasure of learning from during this journey.

I am eternally grateful to my collaborators without whom this research would not be possible. I would like to give special thanks to my amazing lab mates, Michael Coblenz, Matthew Davis, Mary Beth Kery, Toby Li, Jenny Liang, Michael Xieyang Liu, and Daye Nam for their support and contributions to this work. I also had the fortunate opportunity to work with undergraduate and masters students, all of whom helped tremendously with this work – thank you to Shannon Bonet, Kazi Jawad, Emma Paterson, Imtiaz Rahman, Connor Shannon, Matthew Shu, and Lai Wei.

I would additionally like to thank the sponsors of this research, my paper reviewers, and my study participants – all of whom served as the backbone of this work. Specifically, I thank the National Science Foundation (CCF-2007482) and Google for the provided financial support.

My heartfelt love and appreciation extends to all of the amazing friends I have made over the course of the last six years: Karan Ahuja, Lea Albaugh, Mark Buttweiler, Alex Cabrera, Julia Cambre, Tianying Chen, Erica Cruz, Wesley Deng, Jonathan Dinu, Morgan Evans, Will Epperson, Matt Ho, Hyeonsu Kang, Pranav Khadpe, Rushil Khurana, Andrew Kuznetsov, Lynn Kirabo, Tom Magelinski, Courtney Miller, Steven Moore, Huy Nguyen, Wode "Nimo" Ni, Roger Iyengar, Sam Reig, Melrose Roderick, Jaemarie Solyst, Jordan Taylor, Stephanie Valencia Valencia, Danny Weitekamp, Kristin Williams, Vivian Shen, Sung Jang, Franceska Xhakaj, Nur Yildirim, and many more. Thank you all for the laughs and good memories. Special thanks to my dear friends Sarah and Gareth Baldrica-Franklin for our weekly-ish movie nights that helped me stay sane during COVID and beyond. And, of course, Emily Miller, who has been like a sister to me for years.

I would also like to take a moment to acknowledge and appreciate a valued member of our community who we lost far too soon: Sujeath Pareddy. As part of our 2018 cohort of PhD friends, his loss has felt like a giant, gaping hole as our group moves towards graduation – I would like to dedicate this dissertation to him. I remember you, Sujeath, and wish you were still here with us.

Lastly, my deepest gratitude goes to my family – Diane Pearson, Steve Horvath, and Katelyn Horvath. They have stood by me through every up and down, providing immeasurable emotional support. I could not have done it without them. I would also like to thank my extended family and my "bonus" family, Suzanne Robinson, Casey Keating, Renate Woods, and Brittany Thackeray who stood by and joined in the celebrations. I would also like to thank my pet family, Ringo the Pomeranian, Eevee the cat, and Pichu the cat – I know they cannot read, but their cuddles and love has gotten me through many exhausting paper deadlines. Last, but certainly not least, I am forever indebted to my fiancé, River Hendriksen, who has truly been everything to me over the course of this dissertation and beyond – collaborator, friend, shoulder to cry on, co-author and partner. He helped me carve out the life here in Pittsburgh that I have grown to love so much, including happy hours at our local Primanti Bros., so many Pittsburgh Penguins games, and nights at Dependable Drive-In. I love you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Developing software requires developers to keep track of many types of information while performing various interleaved tasks. For example, to debug some code, a developer must first navigate through the code to determine what part or parts of the code are responsible for the bug [126, 139, 195], run the code and diagnose the output to understand how and when the code fails [128, 139, 195], write code and possibly research solutions online to fix the bug [29, 144, 259], then test the code again to ensure the solution worked [70]. Typically, the developer's mental model of this rapidly-evolving problem space is not externalized [33, 148, 189], which can be problematic as working memory is limited [96, 157, 240]. Further, development processes are often iterative [123], can be interrupted [187, 189], and may just be one task amongst many that the developer is handling [124], leading to further cognitive strain.

Given this significant cognitive cost, many research projects have enumerated the challenges in both writing and understanding code. Just some of the challenges include maintaining task awareness [156, 187, 189], making sense of external libraries and their documentation [4, 101, 172] (with software documentation, itself, having its own set of well-known problems [4, 140, 178, 210, 246]), understanding the rationale behind the current code [124, 135, 156, 223], determining what part or parts of the code are relevant to the change the developer is introducing [123, 129], and designing that change and implementing it [129, 144]. Completing these smaller information-intensive tasks often results in the creation of knowledge that is lost, either because it is not externalized by the progenitor of the information or because it is not logged.

Some qualities of programming make tracking this information especially challenging in comparison to other complex sensemaking domains. For one, a developer's end-state is not often reaching an answer or decision to these implicit questions, but is to utilize that information to complete some programming-related task, such as fixing a bug. Secondly, the information landscape is often changing as the developer introduces new code, which can produce new runtime behaviors and introduce new questions. This means that:

- the developer does not normally have the mental bandwidth to externalize this information beyond, perhaps, a short note [104, 148]

- this information is implicitly associated with the corresponding code that stems from the gained knowledge [104, 144]
- as the code changes, the information the developer learns is highly contextualized to the time in which it was discovered or created [17].

We call this information created as a byproduct of authoring or making sense of developer materials (e.g., code, web resources, documentation, etc.) *meta-information*.

In software development, one of the most ubiquitous and well-known meta-information types are code comments. Often created as a way of keeping track of bugs [237] and open to-do items [234, 235], documenting code [218], or informal versioning of code [116], code comments can serve as rich information about what a developer is doing when managed with tooling [244]. Other forms of meta-information about code include Git commit messages, code documentation, code versions, and so on. The act of developing software produces a vast and dense information landscape comprised of these various forms of meta-information. Meta-information also exists in other information authoring domains such as Microsoft Word or Google Docs comments.

In this thesis, I[1] explore developing software systems to capture and present programming-related meta-information to assist in overcoming known challenges in developing software. This work is predicated on the knowledge that many forms of meta-information are generated at different times during the software development life cycle and I hypothesize that this knowledge, when captured and presented in a comprehensible manner that leverages the code or other source information within the developer's working context, can be useful. By associating this meta-information with code, the connection between what the developers care about (i.e., the code) and all of this other information that normally is not captured but can actually answer developers' implicit questions about code (e.g., visited web pages for *where* some code came from, code edit history for *when* some change occurred, added and removed and commented out code for reasoning about *what* a developer has tried, etc.) is an effective way to keep the developers within their working context, while providing them the information they need to answer their questions.

My thesis aims to investigate the following claim:

> *Developer-generated code meta-information created as a byproduct of writing or understanding some code, if properly contextualized given its relationship to some code or documentation, can be both useful to the initial developer and later readers.*

For the initial author, I show that externalizing their thoughts and questions as meta-information can help with keeping track of information while completing cognitively-demanding software engineering tasks. For later readers, these traces of

[1]The projects discussed in this thesis were lead by myself but were developed in collaboration with other researchers – out of respect for their contributions, I will predominately use "we" when discussing the various projects and papers that comprise this thesis.

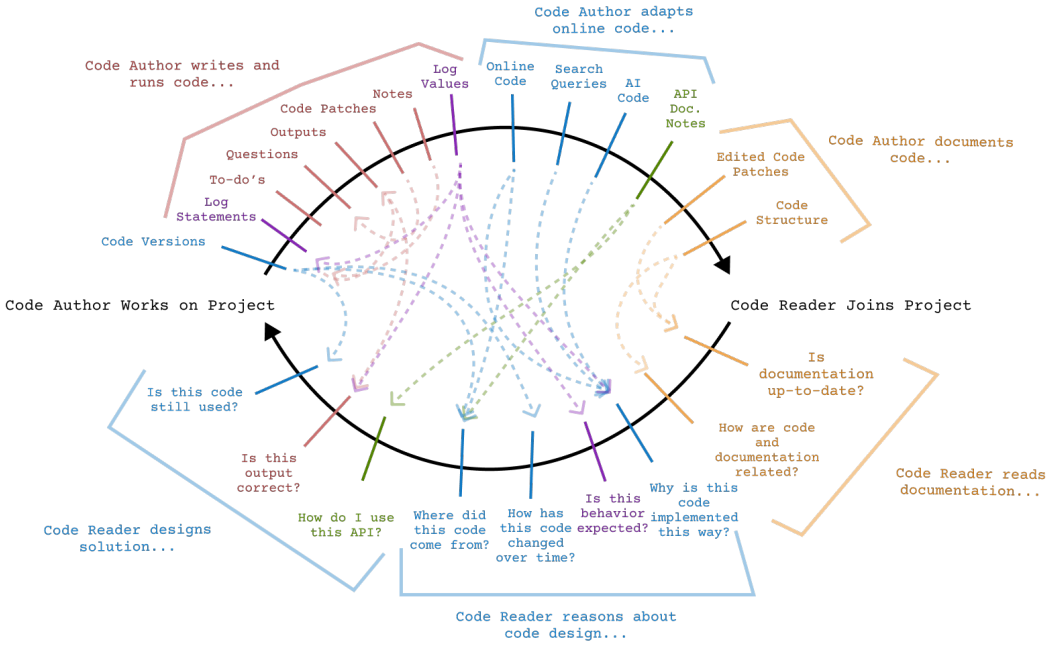

FIGURE 1.1: The "virtuous cycle" of code meta-information generation during software development (top arc) and how that meta-information may be used by later readers (bottom arc). The various forms of meta-information each of my systems generates and exemplar questions that that information may answer are color-coded: Adamite (chapter 3) is shown in green, Catseye (chapter 4) is shown in red, Sodalite (chapter 6) is shown in yellow, Meta-Manager (chapter 7) is shown in blue, and MMAI (chapter 9) is shown in purple.

the context in which the initial development occurred can help with answering otherwise unanswerable questions about code and for better understanding unfamiliar code and learning resources. I also show that code meta-information can be used, not only for reasoning about some code, but also for evaluating written materials about code, given the connection between meta-information and its corresponding code. An overarching goal of this thesis is to treat information about code as a first-class entity, not unlike the code itself, given that the rich history of code and its development is often the *answer* to developers' questions, but is not typically captured in a systematic or easily explorable manner. I envision this meta-information creation and usage as a virtuous cycle for question-asking and answering, as shown in my research framework diagram (Figure 1.1).

1.1 Overview

There exists a vast space of meta-information about code, with varying levels of current tooling support. For example, both large commercial projects and smaller-scale academic studies have extensively explored code versioning, one form of code meta-information, including enterprise-level products like GitHub [162] and research projects such as Variolite [116] or Chronicler [257]. Other areas of code meta-information

have been less explored, including developers' notes written about code, despite prior research confirming that this is an activity that developers sometimes perform [148, 154, 156].

To begin exploring to what extent meta-information can be used to help software developers overcome their sensemaking barriers, I began by exploring *annotations* as a vehicle for associating meta-information with some source information, thus establishing a shared context between author and later reader. To this end, I designed the Adamite system, a web browser-based annotation tool specifically designed for annotating software documentation with features designed to combat known documentation usage barriers. We chose to focus on documentation, given the large body of software engineering research discussing particular pain points that we expected meta-level notes to address. For example, some prior literature has discussed the challenges in finding pertinent information in documentation and how this information may be fragmented across multiple locations within the documentation [4, 52, 246] – to address this barrier, Adamite allows developers to add multiple text anchor points to their annotation, such that all relevant parts of the documentation may be connected to one note. Adamite demonstrated the efficacy of utilizing developer-authored meta-information presented as annotations to assist later developers in their own programming tasks – developers using Adamite annotations performed significantly better on a programming task.

Adamite was successful in helping later developers utilize meta-information about code to overcome their information barriers – however, we found that the initial authors did not experience a significant effect, despite appreciating the annotation tool. Given the need for more support for the initial author in utilizing their meta-information and the increased need for information tracking support when actively writing code in an integrated development environment (IDE), I developed Catseye, a Visual Studio Code [164] extension for annotating code. Catseye not only supported adding free-form text to code that was abstracted away from the original source code (as opposed to code comments) but also supported other developer information-tracking activities including collecting output data and lightweight versioning of code, with the option to annotate these other forms of meta-information. Developers using Catseye performed better when attempting to keep track of information while debugging some purposefully-confusing code.

In active usage of Catseye, it became clear that the extremely mutable nature of code can lead to information within the annotations quickly becoming out-of-date. Keeping annotations attached to the correct code is surprisingly non-trivial – while some simple text edits can be easily interpreted as small arithmetic changes to the line number and offsets along the bounds of an annotation's anchor, for large scale or batch changes, such as a Git merge, the question of what, exactly to do with the annotation and where, if at all, should the annotation be re-attached becomes more complex. Further, given that the nature of these annotations is often, but not always, ephemeral, therein lies a need to support developers in actively managing

their meta-information. This lead us to wonder how we could support developers in *curating* their annotations, given developers' perceived value of the annotations and to explore tooling approaches within Catseye that would allow developers to perform this curation, including algorithmically re-anchoring annotations after significant file changes and batch-level operations to act on annotations for, e.g., removing large sets of out-dated annotations.

Our exploration into how annotations go out-of-date lead to the insight that the relationship between annotations and their anchor points can be a signal of out-of-dateness. Considering out-of-dateness is a large problem in documentation as well, the insight that the connection between code and text is both always changing and the accuracy of the link can be used as a signal for the trustworthiness of the document led to us extending Catseye for long-form documentation with the system Sodalite. Long-form documentation can be a useful information source, given that it co-evolves with the code, but practice shows that does not always happen [140]. Sodalite uses the connection between code and text and its ability to connect the code to the text as a metric for how "healthy" the document is in the user's current programming context. This system showcases how meta-information can be used for new activities beyond simply displaying data.

Adamite, Catseye, and Sodalite were all successful in supporting developers in authoring their own meta-information, but, in their designs, were reliant upon the programmer to actually write the information. Further, what developers have to say about code is only one type of meta-information and is incapable of answering some of the types of questions developers may have, such as *when* a particular change was introduced. To expand the types of questions we could support answering with meta-information and to lower the cost of authoring useful information, we developed the Meta-Manager, a Visual Studio Code extension with a supplementary Google Chrome extension, that automatically listens for and captures editing events of interest in both the editor and the browser. Some editing events of interest occur *within* the editor including copy-pastes between code patches, which may inform where some code came from and when, and code commenting, which may show the author trying different solutions or the author adding or changing documentation. Other edits of interest occur in the *web browser* – we are particularly interested in copy-pastes from the browser (e.g., code snippets taken from GitHub, ChatGPT code snippets, or answers from Stack Overflow posts) into the editor, which may inform *why* some code is written the way it is, given relevant Google Chrome searches and/or visited web page(s). AI-generated code from ChatGPT, in particular, contains a rich collection of interesting additional meta-information about the development of the code, such as the initial query and chat title, which may inform what the developer's original intent was. Indeed, in our study developers were able to use the Meta-Manager to answer otherwise unanswerable questions about code using this rich version history, with information about AI-generated code presented as code meta-information being particularly valued by participants.

With the rise of large language models in recent years, meta-information is only going to become more valuable, not only as a means for potentially training these models, but also as a mechanism for potentially accelerating a developer in their individual sensemaking journey. Code meta-information can be used as a dataset to pull from when utilizing LLMs for software development tasks as a version of retrieval augmented generation (RAG). A context in which this ability to revisit and reason about code changes may be particularly valuable is print debugging, such that developers can use that history to understand how or why some code no longer works and compare that code to the current code. In order to support that use case, I extended the capabilities of the Meta-Manager to allow for direct querying on its historical dataset while expanding the type of information it captures, including outputs produced by `console.log`. I performed a small qualitative needs-finding interview and survey to determine what questions developers would most like to ask their code and output history and designed functionalities within the new Meta-Manager (hereafter referred to as "MMAI") to support those tasks. With this new architecture in place, I showcase that utilizing meta-information for RAG can accelerate otherwise onerous tasks and assist in reasoning when debugging.

In conclusion, I reflect on the contributions made and the insights gained through this body of work, and discuss what outstanding questions and subjects exist for future research (Chapter 9).

Chapter 2

Background and Related Work

Developers are tasked with managing many different types of information when both authoring and making sense of code. In the following sections, I discuss prior literature about how developers make sense of code, tooling support for that process, meta-information about code (including documentation), and prior systems to support the creation and management of some of these types of meta-information.

2.1 Making Sense of Code

Researchers in both Human-Computer Interaction (HCI) and Software Engineering have extensively studied how developers make sense of or comprehend code [27, 29, 57, 63, 124, 126, 127, 128, 131, 156, 185, 211, 212, 220]. These studies are typically situated in the context of understanding unfamiliar code [57, 124, 131, 156, 212], which is a common and significant challenge for developers [172]. For example, developers need to understand unfamiliar code when learning a new API [56, 60, 63, 105, 148, 172, 174, 210], joining a new code base [18, 51, 110, 208, 232, 245], adapting code from online sources [12, 29, 148, 175], taking over a code base from a departed coworker [176, 207, 208], and so on. Some of the challenges developers must overcome when understanding unfamiliar code include maintaining their task awareness [154, 168, 187, 189, 235]; developing mental model of the code [27]; questions, hypotheses, and facts learned about the code [75, 135, 223]; the codes' different versions and outputs [116, 139]; and "working set" of code patches [27]. How developers actually complete these difficult information tracking tasks is further confounded by developers' varying experience levels [61, 130, 137, 142, 156], learning styles and preferences [16, 49, 61, 142, 161], and the nature of their tasks [195]. Even when an implementation task is small or the code is more familiar, developers must continually keep track of these varied information types for maintaining software to varying degrees of success [123].

2.1.1 Developer Tools

Given the diversity of types of information developers must keep track of, a number of research tools have focused on supporting these different activities. One challenge developers experience when making sense of code is navigating through code and

finding and relating code patches of interest (sometimes referred to as the "working set") [27, 45]. Researchers have investigated various solutions to ease navigation costs including clustering code patches as "bubbles" [27] or in a dedicated workspace [1, 45] and augmenting code comments to serve as navigational way points [86, 234, 235, 244]. Other projects have attempted to ease navigation through suggesting code patches to support various tasks including completing a pull request [76, 115], fault localization [70, 194, 195] and for helping newcomers to code projects find where to begin [17, 51, 97]. In all of these cases, the code patches have a higher-level semantic meaning as to why they are relevant to the developer – a relationship which is only sometimes made clear given the tool. All of my tools attempt to make the relationship between this meta-information and its corresponding code traceable and bi-directional to assist in comprehension and navigation.

Relevant to my work are projects that have focused on leveraging the large amount of already-written code and developer resources that exist online to assist in overcoming cognitive barriers incurred when programming. These projects commonly suggest code snippets taken from documentation [28, 159, 180], open source repositories [242], or question-answer forums [107, 269], given a developer's query. The underlying rationale for this design, namely that developers care about code first and foremost, also drives my design approaches, but my systems capture other forms of information beyond just code since code, alone, cannot answer every type of question. Other systems lower the barrier of foraging for programming-related information on the web through integrating web-based functionalities (e.g., web search) into the IDE [47, 80, 89, 186]. My systems, especially the Meta-Manager, attempt to also better connect the activities that are naturally happening both in the editor and the web browser, but do not require the user to adopt a new browsing or development environment. Meta-Manager also has an advantage over these prior tools through not only connecting web activity and the IDE but contextualizing these activities to particular code versions and supporting search to support foraging [230] through queries [215], such that the developer can understand not only what happened with some code, but when and why.

Over the last few years, a new class of developer tools have risen to prominence for making sense of code – LLM–powered coding assistants. Some of these popular tools include ChatGPT [181] and GitHub Copilot [79]. The allure of these tools and, subsequently, their rapid adoption is perhaps unsurprising – unlike many prior developer tools, they are easily accessible, requiring minimal to no set-up, and can immediately provide a benefit to developers through supporting both code generation and code understanding tasks. Some preliminary reports even suggest that the nature of software development itself may be shifting as these tools become more popular [44], with programmers reporting around 31 percent of their new code comes from AI tools [145], they use these tools for high-level guidance on how to approach a task [119], and they spend more time reviewing AI-generated code [24,

167]. Since the public release of these tools in late 2022, there have been many studies looking at the quality of the generated code [55, 68, 239, 263, 268], novice versus expert usage of these tools and the potential harm they may introduce [177, 200], along with how these tools impact perceived productivity [227, 254]. These tools can also be used to "short-cut" code comprehension through directly answering questions about unfamiliar code [36, 112, 224], including APIs [173], and assisting in cognitively-demanding tasks such as debugging [43]. My work extends this line of inquiry through utilizing developer meta-information as a repository unto which an LLM can query upon to support code sensemaking.

2.2 Meta-Information About Code

2.2.1 Writing About Code

One strategy developers employ when keeping track of information is writing down, either through code comments or external notes, what they want to keep track of. In the case of code comments, research has explored what types of information are in code comments [218, 234, 237, 255], whether the code comments are actually useful [26, 199, 255], and how these comments are later used [71, 202] and cleaned up [237, 238]. An analysis of 2,000 GitHub projects found 12 distinct categories of comments developers write and found that information designed for code authors and users appeared less often than more formal types of documentation, suggesting that developers are not frequently commenting for their own benefit or these comments are removed prior to submitting the code to publicly-viewable repositories [218]. Other meta-information about code that is sometimes included in code comments are links to where some code originated from, in the case that the code was copy-pasted from online [11, 90].

Developers also employ note-taking for tracking information about code [46, 148, 154, 156, 187]. These notes are commonly used as memory-aids [46, 148, 187] both for themselves and for communicating insights to other teammates [39, 156], for keeping track of useful resources [148], and for keeping track of their progress on a development task [154, 187]. Notably, these notes often lose their initial utility given their lack of context, while others are authored with the full expectation that they will be thrown away [148] as a form of "information scrap" [21]. Of particular relevance, particularly to Catseye, are tooling solutions for developers that directly support note-taking to assist in code comprehension, such as Codepad [188] or Connotator [111]. Connotator explores supporting different levels of code annotations to support program comprehension, but does not allow annotating partial selections such as expressions or strings nor does it allow the developer to write free-form content, instead only supporting pre-determined tags [111].

Code development and coordination with a software team commonly results in the creation of other types of written works. This includes version control system

(VCS) commit and pull request messages [146, 151, 243], computational notebooks [91, 117, 196, 251, 252], bug reports [23, 203], and code review messages [9, 84, 132, 170, 171, 214]. Developers may also write about their code with the intent of helping others, whether that be a blog post [182, 183, 190], question-and-answer forum post [8, 258, 272], or tutorial [91, 93, 190, 241, 250]. Most of these written artifacts suffer from the same challenges that other written works about code face in that they are typically abstracted away from the original code context, thus making them difficult to discover or glean answers from.

Documentation

Perhaps the most ubiquitous and studied type of writing about code is software documentation. Much prior literature has investigated what information is written in documentation [94, 155], what the problems are with that information [3, 4, 35, 65, 140, 160, 178, 210, 212, 246], how this documentation is authored [53, 219, 233], and automating documentation processes to offset authoring costs and standardize the information present in documentation [2, 88]. Notably, the majority of this documentation work is in the context of software documentation for end users of a software library, e.g. API documentation [172]. Slightly less work has focused on documentation created for other developers working on the same code base, such as internal documentation about the code base [208, 219] or open source on-boarding files [241], where the primary goal is to help other developers understand and contribute to the code base.

Once some software documentation is made, researchers have studied how developers maintain those documents and use that information. In studies of usage, researchers have identified many problems of documentation that lead to the documentation being less trustworthy [150], including questions about how up-to-date the information is [4, 140, 246] and how complete the information is [210, 212]. Maintaining documentation has also been found to be challenging, with developers reporting on the costly time spent on updating documentation [246] and a lack of clarity on where and how to update the documentation given code changes [73].

2.2.2 Other Meta-Information

Code History

Developers keep track of many other types of information that are not or cannot be written down. One ubiquitous form of meta-information about code is code history, i.e., code versions across time. Most software engineering utilizes a version control system (VCS) (e.g., Git) for keeping track of code releases and to make collaboration across potentially many different versions of code possible. However, in between formal check-ins of code, developers have reported a need for keeping track of their code versions. Whether that be in the context of data science programming where intermittent versions may represent small-scale experiments [92, 116, 117, 221] or

when maintaining a code project and trying different solutions for fixing a bug [265] or adapting a code example [29], developers employ a variety of methods for keeping track of these intermittent versions such as commenting out the code [116]. Some systems (e.g., [184, 252]) have leveraged discussion threads about code and mapped the threads to the relevant code – an approach not dissimilar from Catseye, Meta-Manager, and Sodalite. Both these less formal intermittent code versions and committed code versions can serve as meta-information about the "current" version of the code in service of, e.g., understanding how some code evolved over time.

To support sensemaking of code history, prior research has explored creating visualizations of code history. Most code history visualizations reserve the x-axis for representing time, but the y-axis and presentation of the code and edits varies. Some tools adopt a stream visualization in which each "stream" represents a block of code [257]. The stream expands, retracts, and moves up and down along the y-axis as lines of code are added, removed, and the location of the code moves, respectively. Notably, this visualization approach was inspired by systems visualizing other historical, text-based data including Wikipedia page edit histories [249] and Google Doc histories [253]. Other visualizations reserve the y-axis and data points along the visualization for edits that happened to some code at a particular time [265, 266, 267]. Some visualization approaches remove the timeline-style presentation in order to support other activities, such as Quickpose [204], which uses a node-based graph to support actively modifying and annotating the history. Seesoft [62] and Augur [77] choose to visualize the line of code itself and imbue additional meta-information such as who most edited the code and what part of the code structure is this history a part of. Meta-Manager and MMAI differ through combining the approaches of both aggregating information through a stream visualization and showing lower-level details in an additional widget.

Outputs and Logs

Another form of code meta-information closely related to code versions are outputs generated by code. Outputs may be as simple as a `console.log` message (i.e., log statement and value) or as complex as a full graphical user interface. In either case, developers must often keep track of how these outputs change with respect to their code edits [116]. This may happen when the developer must revert to a prior, functional version after a bug is introduced or to compare differences in outputs with the intention of choosing an optimal version [265]. Some tools have been developed to keep track of outputs with respect to the code versions that generated them [117], while others have focused on making the output itself more useful through additional affordances and improved presentation [109].

Log statements and values, in particular, have been extensively studied [78]. Logs are particularly challenging for developers to manage given the scale of logs often accrued in an enterprise system [217], their presentation in plain text making them hard to work with [109], and their decoupled nature from the code and runtime

context in which they were generated [59]. Further, log statements, not unlike code comments, often suffer from the "information scrap" [22] challenge, in that they can be written for a single, throw-away line of inquiry as part of print debugging [19]; otherwise, they may be meant to exist indefinitely to capture code performance metrics [78], but their intent from the outset is not always apparent. Given these challenges, some prior tools have attempted to mitigate these problems through reducing scale [217] and changing log presentation [109]. Despite the short-comings of logging, they are nonetheless ubiquitous, especially in the context of "print statement debugging" [141, 256]. Given this ubiquity as a form of code meta-information and their applied nature when debugging, we developed MMAI as a way to better manage and make sense of their contents using AI while leveraging the architecture of the Meta-Manager to address challenges of scale and lack of context.

In our work, we design tooling approaches to support the authoring and management of programming-related meta-information for sensemaking. Whether this meta-information is in service of externalizing and tracking some thought about code, an intermittent code version, a piece of useful documentation, or the history of a function, the proposed work will unify all of these forms of normally-siloed and de-contextualized information pieces into a singular system.

Chapter 3

Adamite: Meta-Information as Annotations on Documentation

This chapter is adapted from my paper:

[101] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A. Myers. "Understanding How Programmers Can Use Annotations on Documentation". In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA*. ACM, New York, NY, USA, 16 pages.

3.1 Overview

Application programming interfaces (APIs), including libraries, frameworks, toolkits, and software development kits (SDKs), are used by virtually all code [172]. Programmers at all levels must continually learn and use new APIs in order to complete any project of significant size or complexity [60]. In learning APIs, developers depend upon the documentation, including tutorials, reference documentation, and code examples, along with question-and-answer sites like Stack Overflow [140]. However, developer documentation is known to be problematic with common and significant issues including incompleteness of information, out-of-date information, and fragmented information [3, 4, 210, 246].

In this chapter, we investigate how developer-authored meta-information about documentation, presented as annotations, may help developers overcome some of these barriers. We hypothesized that the highly-contextualized nature of developers notes when anchored to text within the documentation, can help the initial annotator better utilize their notes about documentation. Further, when given proper tooling support, this information can be leveraged by later users of the documentation to overcome their documentation barriers.

To explore the concept of annotations as a way of supporting short notes on documentation that are useful both for the author and for later readers, we started with a preliminary lab study that explored the concept of annotations on documentation using an off-the-shelf Chrome extension, Hypothesis [108], and then we performed a corpus analysis of annotations on documentation created using Hypothesis. Given what we learned from these preliminary analyses, we developed our own documentation-specific annotation tool, Adamite[1]. Next, we ran a two-pass user study where we explored the kinds of annotations developers *authored* when learning a new API and then had another set of developers *read* those annotations while attempting to complete the same API learning task using Adamite. We compared these participants to a *control* condition which had no annotations. From these studies, we provide evidence that annotations are useful in helping developers overcome documentation-related issues.

3.2 Preliminary Studies and Design Goals

3.2.1 Lab Study with Hypothesis

To explore the efficacy of annotations as a useful learning device for API learning tasks using documentation, we ran a preliminary study where people learned an unfamiliar API while using Hypothesis [108]. In summary, we found that developers are able to use annotations in the ways we envisioned, but that annotation authoring and reading could be improved for developers by adding additional tooling features.

Study Design

The preliminary study had two distinct phases: the first phase was focused on understanding how developers *author* annotations during an API learning task, while the second phase focused on how developers *read* annotations that are already attached to documentation. In each phase, the 4 participants completed an API learning task adapted from my previous study [100] that required them to forage through API documentation to complete a programming task.

In the *authoring* condition, participants were given the documentation with no annotations, and instructed to add annotations when they learned anything useful, had questions about the content in the documentation, or had any other thoughts about the documentation. In the *reading* condition, participants were given the same documentation, but with annotations added. The annotations included annotations authored by the first author that were designed to be helpful for this task given what developers were confused about in a previous study [100], and other annotations that were designed to be "distracting" to simulate the more realistic case where not

[1] Adamite stands for **A**nnotated **D**ocumentation **A**llows for **M**ore **I**nformation **T**ransfer across **E**ngineers and is a green mineral.

all annotations would be relevant. In total, we had 23 "helpful" annotations and 44 "distractor" annotations, totalling 67 unique annotations.

Results

In the authoring condition, the 4 participants together authored a total of 19 unique annotations. Each participant, on average, authored 4.75 annotations, with annotations averaging 4.41 words. Hypothesis also allows users to simply highlight a piece of text on a web page without adding any text content to the anchor (hereafter referred to as "highlight" annotations) — out of the 19 unique annotations authored, 5 were these simple highlight annotations.

Many of the annotations that participants created showed a part of the documentation that illustrated how to achieve some part of their current task. Other annotations served as task reminders or open questions the author had about the documentation content.

In our analyses, we also found that participants had many questions about the documentation (on average, 10.8 questions per participant) which were not annotated. While trying to answer these questions, participants routinely encountered more confusing information, resulting in them losing track of their original questions. Given this confusion, participants struggled to answer their questions, with only 29% of questions definitely answered. Notably, Hypothesis does not have any way of marking or following up on a question.

From this initial preliminary study, we found evidence that annotations may enhance the original text. Additionally, we found support for different types of information needs that are not directly supported by Hypothesis, such as keeping track of open questions. In the reading condition, we also learned through our "distractor" annotations that annotations need to be easy to skim and relatively short in length and we need more support for discovering annotations, either through more anchor text points or filter and search.

3.2.2 Corpus Analysis of Hypothesis Annotations

In order to supplement our preliminary study, we queried Hypothesis's API to get a list of public annotations which developers have already made on official API documentation including public APIs from Google, Microsoft, Oracle, and Mozilla, along with other developer learning resources including Stack Overflow, W3Schools, and GitHub.

Across these sites, we found 1,995 public annotations. Of the 1,995 annotations, 196 were questions about the content of the documentation, and 995 were highlight type annotations. Of the 1,000 annotations with content, 16 of the annotations were to-do items the author wanted to follow up on, 43 pointed out problematic aspects

of and potential improvements to the documentation, and 79 were created to specifically call out important or useful parts of the documentation[2]. These annotations were authored by 298 unique users (average = 6.694 annotations per user, minimum = 1, maximum = 677) across 1,143 unique web pages. The authored annotations were, on average, 8.79 words long and were anchored to text that averaged 12.35 words.

We believe some of the 1,000 annotations that contained content could benefit other developers with additional tooling support. For example, a Hypothesis user annotated the text "you can pass the path to the serve account key in code" and asked how they can do that. This user then later annotated a code example showing how to achieve this behavior at a different point in the documentation and said "finally found it". While these two annotations depend upon one another in order to make sense and point to different parts of the documentation, Hypothesis does not allow for these annotations to reference one another, suggesting a need for better tooling support for multiple anchors for annotations and keeping track of open questions.

These annotations provide support for our claim that some developers are willing to write annotations and attach them to documentation, as they are already doing this. Moreover, the annotations we found follow some of the patterns we identified in our preliminary study, such as open questions and issues. However, Hypothesis's general-purpose annotation system does not have enough support to effectively utilize these annotations.

3.2.3 Design Goals

Given prior literature and what we discovered in our preliminary studies, we developed the following design goals for our system:

- **Support developer note taking through annotations.** Developers sometimes take notes on what they have learned [148, 154, 156, 187], even when using documentation (Section 3.2.2, [148]).
- **Support developers' question-asking and answering.** Developers have many questions about unfamiliar APIs and their documentation ([60, 223], Sections 3.2.1 and 3.2.2).
- **Support diagnosing documentation issues.** Developers commonly identify documentation issues including obsolete information [4, 246], incorrectness ([4, 246], Section 3.2.2), incompleteness [4, 37, 52, 209, 210, 246, 274] and ambiguities [4, 209, 210, 246] but have no way of sharing that information.

[2]These counts were generated by counting instances of phrases like "incorrect" and "todo" in the annotation content, then manually reviewing all of the annotations that contained those phrases to determine if they were actually e.g., an issue or todo item.

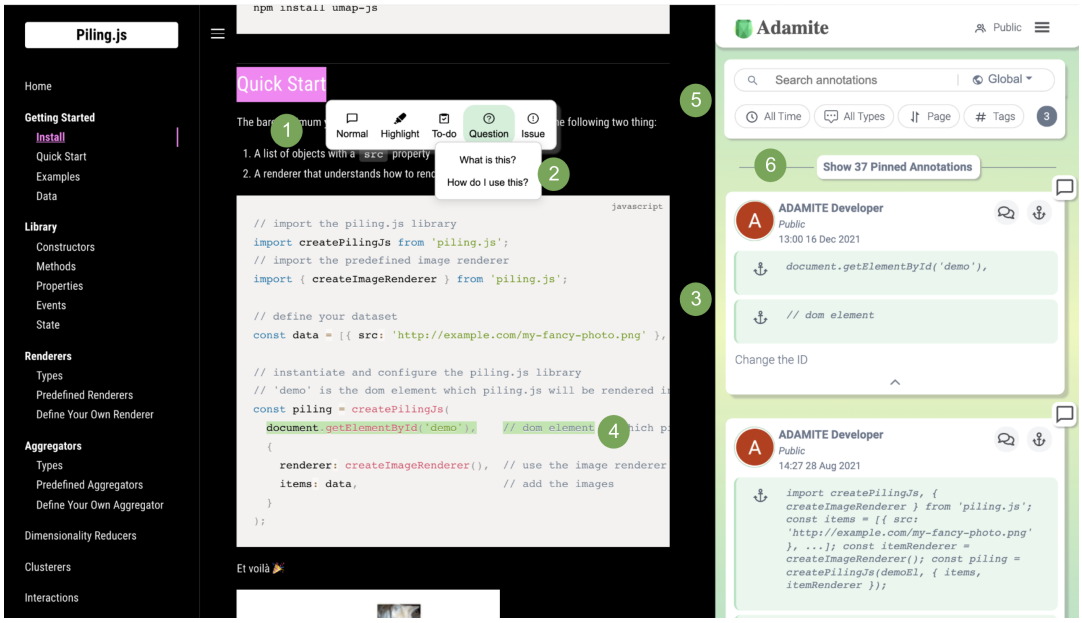

FIGURE 3.1: Adamite's sidebar (on the right) open on an already-annotated web page in the browser. (1) shows the pop-up for when a user selects some text – at this point they can begin creating a new annotation by selecting an annotation type. (2) shows the menu of question annotation prompts users can choose from. (3) shows a published normal annotation with two anchors. (4) shows how the annotated text appears on the web page. (5) shows Adamite's search and filter pane. (6) shows the pinned annotation list button.

- **Support developer task-tracking.** Developers take notes on open tasks that they must work on, especially when interrupted [187], and occasionally take notes on tasks they must complete that are related to parts of the documentation they are reading (Section 3.2.2).
- **Support connecting related parts of the documentation.** Developers need to build up a mental representation of an API [105, 121, 148] and connect related resources [4, 52, 246].
- **Support better discovery of important parts of the documentation.** Developers, especially selective [30] and opportunistic [29] learners, want to quickly find information that is relevant to them ([160], Section 3.2.1).

3.3 Overview of Adamite

We designed Adamite, a Google Chrome browser extension, specifically to help developers keep track of important information, organize their learning, and share their insights with one another. To create an annotation, a developer, who we call the annotation "author" simply needs to open the Adamite sidebar, highlight some text on the web page (called the "anchor"), select the type of annotation, optionally add text to a rich text editor that appears in the sidebar, and click on the publish button. Once published, the text that the user annotated will be highlighted on the web page and the annotation will appear in the Adamite sidebar – see Figure 3.1.

Users may also add tags and additional anchors to the annotation. Annotations can be published publicly, privately, or to a group of Adamite users. Once an annotation has been published, it may be replied to by others, and edited or deleted by the original author. Clicking on the anchor icon on the annotation will scroll to the part of the web page the annotation is anchored to (or will open a new tab if the anchor is on a different page) – conversely, clicking on the highlighted text on the web page will scroll to the corresponding annotation in the sidebar.

One goal of Adamite is helping developers structure and share what they learn in the documentation in a way that is useful both to themselves and for later developers. To achieve this, we developed annotation types. In addition to the typical "normal" (with a user-written comment) and "highlight" (just the anchor and no comment) annotations, Adamite supports question, issue, and to-do annotations. We chose these three annotation types to assist developers in keeping track of their questions, to point out and possibly attempt to rectify issues found in the documentation, and to help them keep track of their tasks. Issue annotations have a button intended to alert key stakeholders, such as the documentation writers, of the described problem with the documentation. Question annotations are stateful, meaning unanswered annotations will stay available until the developer either marks the question as "answered" (at which point the answer will be appended to the original question), or marks the question as "no longer relevant". To-do annotations are also always available until they are marked as complete.

Question and to-do annotations are always available using Adamite's "pinning" mechanism. Web annotation systems typically only show annotations that are on the user's current web page. However, considering that documentation may be spread across many pages and developers may visit many web pages when attempting to complete a programming task, we added in the ability to *pin* an annotation, such that it is always available in a list at the top of the sidebar. To-do and question annotations are pinned by default, since the developer is unlikely to find their answer or finish their task while they are on the same web page.

Developers can further structure and connect their information using *multiple anchors*. A common documentation problem is fragmented information, so developers can compose together related pieces of information using the annotation as a joining factor. This feature can be used to connect parts of the documentation that the user feels should be presented together, or to better contextualize their annotation. Developers may also use anchors as a way of collecting multiple parts of the documentation that they feel are related to one another given the developer's task and their evolving understanding of the API.

For later users of the annotated documentation (who we call annotation "readers," but can be the same person as the annotation authors), it is likely that not all of the annotations are relevant to what the developer is trying to do. To help readers find the most relevant annotations, we support search (using Elasticsearch [64]) and filters (see Figure 3.1-5). Readers can search across a web page, website, or across all

of Adamite's annotations and filter on the annotation type, when the annotation was created, and what tags the author has tagged the annotation with. Readers may also sort the annotations by their location on the page or by the time at which the annotation was authored. Adamite also follows the design of other web-based annotation systems by only showing annotations on the current web page, by default.

3.4 Lab Study

In order to understand the role that annotations play in developers' documentation usage while learning a new API and using Adamite, we ran a lab study with three conditions to understand how developers create and use annotations. Participants in one condition *authored* annotations while completing an API learning task, and participants in the second condition *read* these participant-authored annotations. The third condition was a *control* condition where participants completed the same API learning task using just the documentation. The lab study consisted of a training task, a programming task, and a survey to assess the participant's background.

3.4.1 Method

Training

Each condition included a training exercise using Tippy, a React library for making tooltips, and its documentation to either familiarize the participants with Adamite and its functionality (Adamite conditions) or to familiarize them with thinking aloud while reading through documentation (control). Participants in the Adamite conditions learned how to create an annotation of each type, reply to an annotation, add an additional anchor to an existing annotation, search, filter, edit and delete an annotation and practiced thinking aloud while performing these tasks. The control condition practiced thinking aloud when they had a question, found an answer to their question, and identified an issue in the documentation.

Task

For the task, participants were asked to complete an image aggregation and organization task using Piling.js (hereafter referred to as "Piling"), a JavaScript library for handling visualizations [193]. Piling was chosen as it is a relatively small library, meaning the participants would have adequate time to gain a high-level understanding of the library during a lab study.

The task was to use Piling to take a set of four provided images and render and sort the images (see Figure 3.2 for the output and detailed steps). The task was chosen as, despite its apparent simplicity, it requires the participant to learn some of Piling's core concepts. Participants were objectively graded upon how many of the 4 steps they were able to complete correctly. To start, participants were given a JavaScript file containing comments stating the goal of each step.

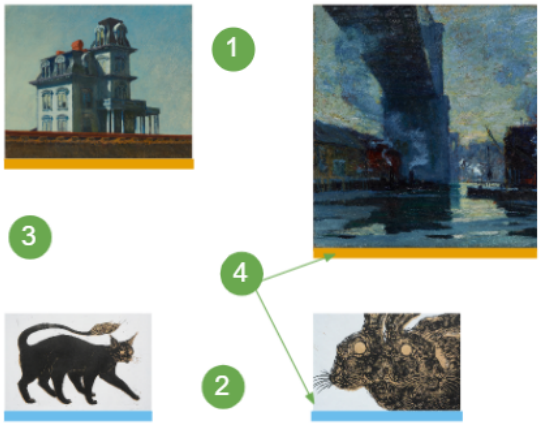

FIGURE 3.2: The correct output for the task. Each number refers to the step number. (1) creates and renders the 4 images. (2) puts the images in 2 rows. (3) arranges images by a user-defined property using the `arrangeBy` method. (4) requires the user to set a label on their data, such that elements with the same label will have a matching stripe along the bottom of the picture.

In addition to completing the programming task, participants were asked to think aloud and to pretend as though they were in a small team learning Piling. Dependent upon the condition, further instructions differed slightly. *Control* condition participants were told that they needed to relay what they had learned to their teammates in whatever way they would normally do so, such as note taking. Adamite *authoring* participants were instructed to create annotations with any questions or thoughts they had about the documentation, issues they found in the documentation, and thoughts they wanted to follow up on and that these annotations would be shared with their future teammates. Each authoring participant started with an un-annotated version of the documentation. In the Adamite *reading* condition, participants were given annotated documentation and were told to pretend that the annotations were created by a teammate who had already learned Piling and were instructed to speak aloud when an annotation was helpful or unhelpful. Participants in the reading condition were not required (but were allowed) to create annotations or interact with the annotations present in the documentation.

Annotation Selection

In choosing annotations to include in the reading condition, two researchers separately coded each annotation created during the authoring condition for whether or not to include it. Inclusion criteria included identifying matching annotations across participants to select which one was the clearest, most appropriately anchored, and concise – qualities informed by our preliminary study and others [5]. The predominant cause for the majority of annotations to be removed was redundancy – participants commonly annotated information related to the first two steps of the task (see Figure 3.3)[3]. We also excluded annotations that the participant later stated were

[3]Note that this is due to being a lab study – in a realistic situation, people would likely read an existing annotation and not create a redundant one.

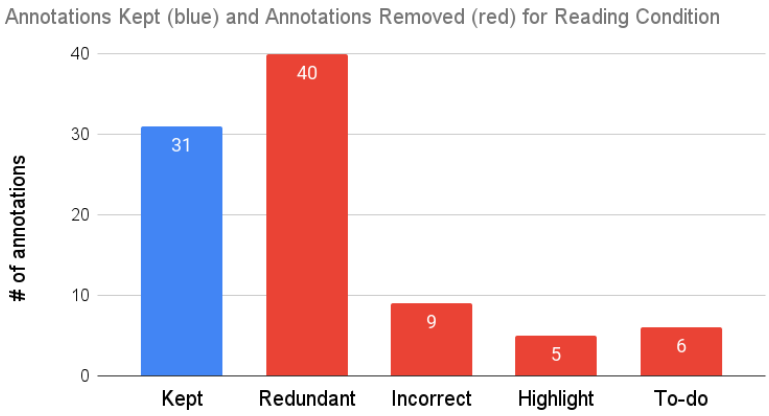

FIGURE 3.3: The number of annotations removed for each reason, along with the annotations kept, out of the 91 total annotations. 2 highlight annotations were retained as the users edited them to add text, making them semantically identical to normal annotations.

incorrect or that the participant deleted, and annotations that lacked sufficient context (including all highlight annotations). Finally, we omitted to-do annotations, as they are designed only for the original author's usage. The researchers had a 71% agreement – in the cases where the researchers disagreed, they had a discussion until agreement was reached. Through this process, we were left with 31 annotations. One additional annotation was added by the first author to assist with the final step of the task, as no participant in either the authoring condition or control condition got to that point of the task. After this process, we had 32 annotations, with 18 normal type annotations, 10 issue type annotations, and 4 question annotations, 3 of which were answered[4]. We did *not* omit any annotations due to relevance or correctness, since we wanted to leave in anything that at least one participant wanted to comment on to be more realistic.

Participants

We recruited 31 participants using departmental mailing lists at our university and social media. One participant could not finish the study due to technical difficulties, so we only report on the 30 who completed the whole study. Each condition included 10 participants and were randomly assigned between the authoring and control condition – the reading condition occurred after the other two conditions so all remaining participants who signed up were assigned to that condition.

All of the participants were required to have some amount of experience using JavaScript, not to have used Piling before, and to have been programming for at least 1 year (actual minimum: 1 year, maximum: 20 years, average: 7.98 years). The participants' professions included graduate students in computer science-related fields, user experience researchers with a computer science background, and professional

[4]We included one unanswered question to account for the realistic situation that not all questions would be answered and because the question asked was a common question among participants – notably, answering this question was not necessary for succeeding in the task.

programmers. The gender composition of our study was 19 men, 9 women, and 1 non-binary person. Participants across each condition had a similar amount of JavaScript experience and years of programming experience.

All study sessions were completed remotely using video conferencing software. Participants were audio and video recorded, and each participant's session took approximately 90 minutes, with 45 minutes of that time allotted for the programming task. Each participant was given access to the Piling documentation and a CodeSandbox.io [31] project which had JavaScipt, HTML, and CSS files with Piling installed and a photo of the output, along with written-out steps for the task. Participants were compensated $25 for their time, save for 2 participants who elected not to be compensated.

Analysis Methods

Across all of the conditions, we objectively graded participants on whether or not they succeeded in completing each of the 4 steps outlined in the task instructions. In the Adamite conditions, we analyzed the video recordings and log data to count how many annotations participants authored, and how often they interacted with Adamite and its annotations.

We qualitatively coded the annotations developers made in order to characterize developers' annotating strategies. Using an open coding method, two authors coded the normal type annotations by independently coding each annotation and refining categories based upon their individual codes. For issue and question type annotations, we coded the annotations dependent upon what issue in a list of commonly defined documentation issues [4] was identified in the annotation (issue type) or what issue caused the participant's confusion with issue types including: incompleteness, fragmentation, incorrectness, poor code example, and ambiguity. Two of the authors independently coded the annotations and reached 75% agreement when coding the issue annotations and 73% for the question annotations – remaining annotations were discussed until agreement was achieved.

In the annotation reading condition, we analyzed how often participants said that an annotation was helpful or unhelpful in order to better understand what annotations succeeded in helping participants. We calculated average helpfulness by how many participants said an annotation was helpful and dividing by how many participants encountered the annotation. We ranked annotations from most helpful to least helpful by how many participants said the annotation was helpful subtracted by how many said it was unhelpful[5]

[5]We chose this method of ranking in order to account for the "impact" of an annotation – for example, if only one participant encountered a specific annotation and found it helpful (100% helpful), we did not want that to be seen as "more helpful" than another annotation that helped 5 out of the 6 people that encountered it (83.3% helpful).

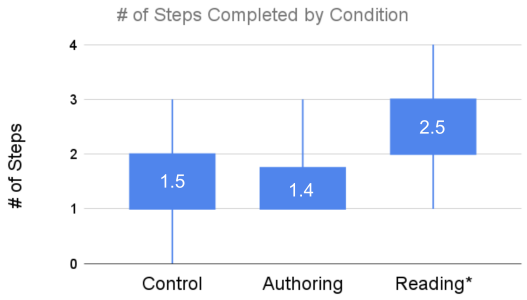

FIGURE 3.4: The difference between reading and each of the other two conditions is statistically significant, but the difference between control and authoring is not. The average number of steps completed is in the center of each box.

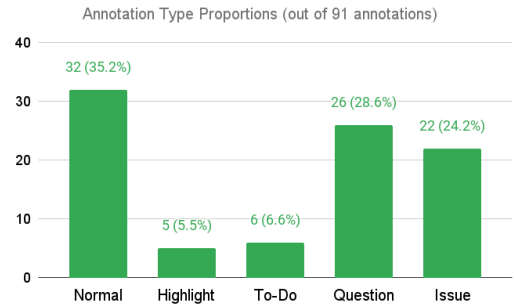

FIGURE 3.5: The proportions for each type of the annotations made in the authoring condition (out of 91), with the exact count for each type above the bar and the proportion in parentheses.

In the control condition, we kept track of whether and how the participant chose to relay their information to their teammates. We also referenced the auto-generated transcripts to find and count whenever a participant stated a question.

3.4.2 Results

On average, participants in the control completed 1.5 of the 4 steps, authoring participants completed 1.4 steps, and the reading condition completed 2.5 steps (see Figure 3.4). Participants in the reading condition performed significantly better than participants in the control and authoring conditions (paired T-test versus control, $p < .01$, paired T-test versus authoring, $p < .01$). This provides evidence annotated documentation helps developers in using API documentation.

In the control condition, 1 participant chose to take notes in a Google Doc and 1 participant made comments in their code as notes for their future teammates. 2 participants spoke aloud when they had a thought that they would want to share as a note to their teammates, but did not actually write any notes down. The remaining 6 control participants did not take notes or verbally indicate the intent to take notes at any point during the study. This suggests that without a mechanism for externalizing their thoughts, these 6 participants may not have been able to actually share what they learned and only 2 participants had any artifact to share with their future teammates.

How Developers Annotate Documentation

The 10 participants in the annotation authoring condition created 91 annotations across all five of the annotation types (see Figures 3.5 and 3.6). The annotations that participants authored were, on average, relatively short in length at 9.31 words (minimum = 0, maximum = 34, median = 8). On average, each participant authored 9.1 annotations (median = 8, standard deviation = 4.094, minimum = 5, maximum = 18), with the most used annotation type being the normal-type at 32 authored annotations (35.1%). 2 participants in the annotation *reading* condition created 6 annotations (3 normal, 2 highlights, and 1 question), resulting in 97 annotations across all conditions. The rest of the analyses just look at the 91 annotations from the authoring condition.

Considering the large amount of normal type annotations and how normal annotations can contain nearly any type of information, we sought to characterize the content of these annotations. Through open coding, two coders defined 5 categories – "note to self" in which the participant made a note about the documentation's content that was primarily for themselves, "explanation of code" in which the participant tried to better explain what a particular code example was doing, "hypothesis" in which the participant guessed about how some part of Piling works, "important to task" in which the participant highlighted a particular part of the documentation as critical for one of the steps of the task, and "other" for any annotations that did not fit into the previous categories. With this categorization, we had 12 "note to self" annotations, 10 "explanations of code", 7 "hypotheses", 2 "important to task" annotations, and 1 "other" annotation. The 1 "other" annotation was an annotation with no content that was created purely as a navigational aid. Participants commonly used "note to self" to keep track of their information and contextualize it to their open task, and used "hypotheses" as ephemeral thoughts about the documentation – these activities are more directly supported in our tool, Catseye, within the context of the code.

The "note to self" annotations typically served as reminders to the author to externalize an important detail about the API or a code example. For example, one participant annotated a call to `document.getElementById('demo')` with "remember to change the ID" as a reminder to themselves, as they were in the process of adapting the example. This note could also benefit future users of the documentation as a note that the code example will not work without some modifications.

Unexplained or poorly explained code examples are a frequent problem in documentation [4, 246] and Piling was no different, so our participants attempted to explain some of the code examples and, sometimes, contextualize them to the goals of the task. The most helpful and second most helpful annotations are both explanations of code with the most helpful explaining how to use Piling's row property to create columns, and the second most helpful annotation explaining how the code example for `piling.arrangeBy` works and how to adapt the code example to work using a callback function – both necessary steps for completing the task. Further, in

Documentation Issue	Issue Annotations	Question Annotations About Issue	Percent Questions Answered
Incompleteness	3	6	50%
Fragmentation	4	3	33%
Incorrectness	4	2	0%
Poor Code Example	10	7	43%
Ambiguity	2	12	33%

TABLE 3.1: Counts of each issue and question annotation that identified or was caused by an issue discussed in [246]. Note that two code examples did not work because they suffered from a fragmentation issue, so they are coded both as a poor code example and a fragmentation issue. Similarly, all fragmentation questions were caused by fragmented code examples, so they are also coded as both a fragmentation question and code example question.

our qualitative coding of issue and question annotations, poorly explained code examples were the most common type of identified issue, suggesting that participants valued the ability to express lightweight thoughts about code in context.

Participants also often annotated code examples which did not work as an issue type annotation, with 10 issue annotations being labeled in the closed-coding as a "poor code example". Poor code examples were the most frequently identified issues (Table 3.1). These code examples typically did not work either because a variable in the example was undefined and thus the code could not be just copy-pasted (7/10) or because the documentation did not show an output of what the code example actually did (3/10). Some of the undefined variable issues occurred because the variable was defined in a different part of the documentation (2/7) – a documentation fragmentation issue as well as a code example issue. One participant was able to use multiple anchors to suggest where the definition for the variable should be moved to in order to make the code example work.

Developers also had many questions that relate to documentation issues reported by prior studies [4, 246]. Ambiguity and poor code examples were the source of the majority of developers' questions, which matches the findings reported in [246], with ambiguity, in particular, standing out as a common and severe blocker for developers. Ambiguity and fragmentation issues also resulted in questions that were difficult for annotation authors to answer, with only 33% of questions caused by ambiguity and 33% of questions caused by fragmentation being answered. Considering some participants were able to solve fragmentation issues using multiple anchors with Adamite, this suggests these developers' questions may have been answered if they had been presented with these annotations. In fact, in the Adamite reading condition, 2 participants had their issue of `aggregateColorMap` not compiling solved by an annotation that used multiple anchors to link to the part of the documentation that defines `aggregateColorMap`.

Issue-type and question-type annotations accounted for roughly half of all the authored annotations. Nearly all issue annotations succeeded in identifying at least one of the issues identified in [246] (see Table 3.1), save for one issue annotation that stated that a particular part of Piling is "super high maintenance for a simple use case", which is not an issue with the documentation, but with the library itself.

Notably, this issue annotation was the third most helpful annotation in the reading condition with participants appreciating that it warned them about a part of the library they were thinking of using, suggesting that issue annotations are useful beyond identifying documentation problems.

Participants also hypothesized about parts of Piling, including how the library worked and what various constructs were relevant to the task. One participant annotated a code example that used an undefined parameter and said "I think that k is equal to the number of photos in the data set. I could be wrong though - TBD". Another participant was trying to determine what function to use to sort their images and annotated `piling.groupBy` with the text "This might be helpful", but, upon finding `piling.arrangeBy`, annotated that method with "Actually, maybe this". These hypotheses along with "note to self" annotations demonstrate how annotating can be a lightweight technique for jotting down thoughts as a developer is gaining familiarity with an unfamiliar library.

Considering roughly half of the authored normal annotations are primarily beneficial to the original author (i.e., notes to self and hypotheses) and every participant made a personal annotation (i.e., notes to self, hypotheses, to-do's, and highlights), we find evidence that annotating is an effective mechanism for externalizing information and helpful for the author. Further, all participants revisited at least one of their annotations at least once (min = 1, max = 36, average = 10.225 revisits per participant), suggesting participants were able to get some utility out of their annotations. We also included 6 notes to self and 3 hypotheses in the reading condition to see whether these thoughts could benefit other developers. Notes to self, in particular, were the most revisited type of annotation by their authors, and participants, on average, revisited these notes 1.8 more times – more than any other annotation type or coded normal annotation types.

Some of the annotations were used in conjunction with Adamite's other novel features, resulting in the annotations being more useful. The most commonly revisited annotation, a note to self, had 5 anchors with each anchor describing a necessary step in order to properly instantiate the `piling` object. The participant pinned this annotation such that they could reference the anchor steps in their CodeSandbox project (which was open in a separate Chrome tab) – this annotation was also useful in the annotation reading condition with one participant replying to thank the author. This shows that Adamite's features not only support the creation of lightweight notes but also allow developers to utilize their and other people's notes in context.

The annotations that participants authored were, on average, relatively short in length at 9.31 words (minimum = 0, maximum = 34, median = 8). The short length of annotations makes them relatively easy to author. The annotations included in the reading study averaged 11.94 words and the 10 most helpful annotations were, on average, 13 words long. These results suggest that short notes are able to help future users of documentation, while not requiring a large amount of effort on the author's part to create. These annotation lengths are also consistent with the annotations

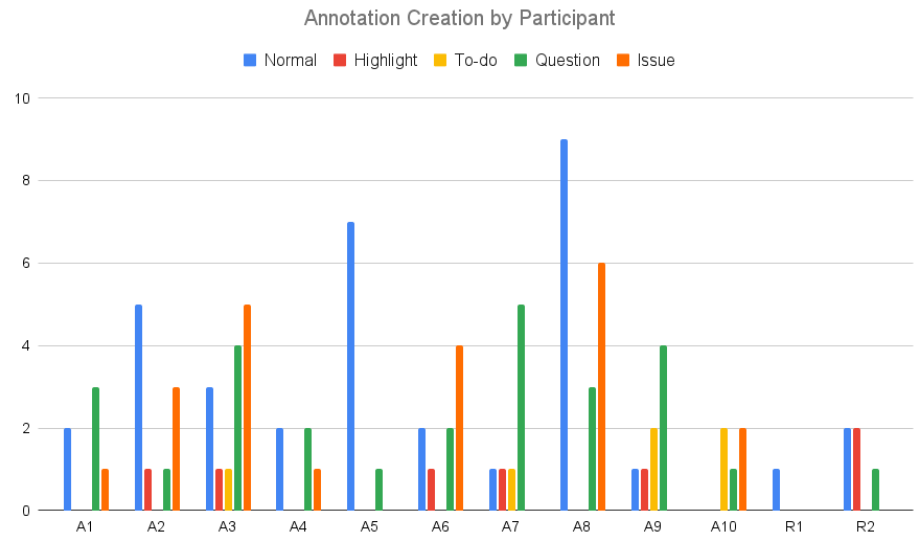

FIGURE 3.6: Which participants made what type of annotation. A1 through A10 refer to the 10 authoring condition participants. R1 and R2 are the two reading condition participants who created annotations.

authored using Hypothesis in our corpus analysis, suggesting these annotations are similar to the types of annotations authored in the wild.

In terms of task completion, participants in the annotation authoring condition on average completed 1.4 steps out of the 4 steps required to complete the task (standard deviation = 0.69, minimum = 1, maximum = 3, median = 1) (see Figure 3.4). There is no statistically significant difference between the control condition completion rate of 1.5 and the 1.4 in the authoring condition (two-tailed T-test, $p = 0.78$), suggesting that annotating the documentation, while not increasing their performance, also did not require so much overhead that participants were unable to complete the task in the same amount of time as if they had not been annotating. Moreover, considering how often developers' revisited their notes, specifically their "notes to self", this suggests authors were able to successfully use annotations as an externalization of their thoughts. Chapter 4 further explores how to make annotations more beneficial for the initial author.

How Developers Use Annotated Documentation

Participants in the reading condition read, on average, 23.7 annotations (including revisiting annotations they had already read, with 72% of annotations read more than once), and read, on average, 15.6 unique annotations. Participants found 45% of the annotations that they encountered helpful, and only 8% not helpful. The top-performing 6 participants in the reading condition also reported the highest proportion of helpful annotations, suggesting that their success may be attributed to their use of the annotations.

Of the 32 annotations included in the reading condition, the most helpful type of annotation for readers of the documentation was answered question annotations, with, on average, 54% of participants who encountered them stating they were helpful. Normal-type annotations were the second-most helpful type of annotation (avg. 47% helpful) and issue type annotations helped on average 35% of the time. Some issue type annotations were more helpful than other issue annotations including annotations that identified poor code examples, which were helpful, on average, 45% of the time. Even though these issue annotations did not necessarily suggest a solution, they did work in confirming the participant's suspicions that the documentation itself was incorrect and not the participant's implementation. Sometimes, participants found useful annotations through search – participants searched a total of 78 times and 11 of these searches returned an annotation that the user immediately found useful.

The most helpful annotation was anchored to the text "columns" and simply states "Use this to create rows" — a short, 5 word annotation that explains how this property can achieve an effect required by the second step of the task that is not immediately clear when reading the documentation. The second most helpful annotation also succeeded in elucidating how to use part of the API that is relevant to the task through using multiple anchors and clarifying a code example for `arrangeBy` – a method necessary for the third step. In the reading condition, participants were more successful in completing these two challenging steps, with 9 participants able to complete step 2 and 4 participants able to complete step 3. This increase in performance suggests that participants were able to utilize what the annotation authoring condition learned in order to more effectively complete their task.

Annotations that were not as immediately relevant to the task could also be helpful. Two issue type annotations were the third and fourth most helpful annotations, each warning participants about unhelpful and incorrect parts of the documentation. For example, the fourth most helpful annotation, which helped 4 participants, stated that a code example in the documentation throws an error that a variable is not declared — participants found this annotation useful as it deterred them from using that code example or, if they did use it, reassured them that they were not doing something wrong, since another user had the same problem.

Conversely, the *least* useful annotations were the ones that lacked enough context to be reusable. For example, an unhelpful annotation was an annotation anchored to the text "columns 10", stating that the default value of `column` is 10, which is redundant with the text of the anchor. The original author annotated this as the reason their 4 images showed up in a single row since the column parameter needs to change to make 2 rows, however, the annotation is missing this full context. Future annotation systems designed to help programmers should explore automatically inferring additional context to make the annotation more comprehensible to later users — if Adamite were to be integrated with the developer's integrated development environment (IDE), we may be able to capture the code and its output before and

after the user created the annotation to better explain why they made the annotation and what they were trying to achieve. Adamite and Catseye (Chapter 4) share the same database, so supporting synchronicity between the two applications is technically possible – ultimately, this path was not fully explored due to a lack of clear design vision in terms of how a between-application, contextually-aware annotation platform could support developer sensemaking. Meta-Manager (Chapter 7) also explores bridging together browser and IDE activities in order to support developer question-asking and answering.

Participants especially appreciated the normal type annotations that were explanations of code, with participants finding them helpful 63% of the time. Code explanations typically elucidated what a code example was illustrating or explained how to adapt a particular code example for the purposes of the task. "Notes to self" were also surprisingly useful, with participants finding them helpful 53% of the time – given that the notes to self typically represented a thought or reminder the developer had about the documentation while completing the task, these results suggest that the participants in the reading study had similar thoughts about the documentation. Conversely, hypotheses were not very helpful, with only 16% of participants finding them helpful – given the uncertainty of these annotations, participants may have found them less trustworthy. These results suggest that explanations of code and developers' personal notes can be useful if they are framed in a knowledgeable fashion, while hypotheses are more useful for the original author.

3.5 Limitations

Given Piling's complex documentation, Adamite may not be as helpful when the documentation is simpler or clearer, so the study cannot necessarily be said to apply to those situations. Our lab study was also constrained to a single forty-five minute session, so it is unclear how developers' API learning and annotation authoring and reading behavior would change over a longer period of time. Piling also has a small user-base, so we have less evidence that Adamite would be useful for APIs with better documentation or APIs with a large user-base that can provide useful crowdsourced information on Stack Overflow or mailing lists. Future work should see how developers use Adamite in the wild with more popular APIs over a longer period of time.

Considering we selected the annotations to be included in the reading condition, there is an additional limitation that this curating process would not happen in the wild, and, since the last annotation added was created by a researcher, we cannot say that every annotation was participant-authored. The most common reason for removing an annotation was due to the annotation's content being redundant with another annotation which would be less likely to occur in the real world where users can see other users annotations and will most likely be performing different tasks. While annotations unrelated to the user's task may be distracting, some annotations,

such as issue annotations, may be useful to any developer using the construct(s) the annotation references. Further, Adamite supports tagging and filtering which could be used to filter out annotations that are unrelated to what the user is working on. Annotations could also be curated in order to ensure higher quality annotations are more commonly seen using crowd-sourcing methods (e.g., supporting voting and editing other users' contents [6]) which could be added to Adamite.

Adamite as a tool is also limited by its inability to work on dynamic web pages such as Google Docs since dynamic web pages do not have stable anchor points for our highlighting algorithm and the content on these web pages often changes, causing the annotation to lose its original context. Considering developer documentation is relatively static, Adamite works well in this situation, but we do not claim that Adamite will work on more volatile pages. Adamite also does not work on PDFs, despite some documentation existing in the form of a PDF. API documentation is a good use case for Adamite, though, since there are many well-known issues with documentation that annotations can address and API documentation is commonly presented on a website.

3.6 Discussion

Our results suggest that annotated documentation is useful for documentation readers in overcoming some of the known barriers of documentation and that the act of annotating when learning a new API can help developers keep track of their thoughts and open questions. Creating annotations was also useful to the author as a form of self-explanation, which has been shown to be useful for learning in prior studies [32, 40], and these self-explanations, or "notes to self", were useful to others. The novel features of Adamite, especially types, multiple anchors, and pinning, helped annotation authors better structure and contextualize their information and helped annotation readers find relevant information.

Participants particularly enjoyed that the annotations had types, and also envisioned future enhancements. 7 participants in the authoring condition noted that they enjoyed the question-type annotations, with 2 specifically mentioning the two built-in question menu items, suggesting that assisting in annotation authoring may be a fruitful avenue for future annotation systems. One participant made an issue type annotation, but wanted the issue to only be shared with documentation writers, while 2 other annotators wanted the "issue" type annotation to be less "confrontational" and instead frame the annotation as a "suggestion" to the documentation maintainers.

Having types for annotations also resulted in two completely separate classes of annotations users made. As demonstrated in our qualitative coding, the kinds of information developers noted in their normal annotations (i.e., notes to self, important to task, hypotheses, and explanations of code) is very different from the information that developers noted in their issue and question annotations, which were primarily

documentation-focused. This suggests that annotation typing is an effective way to elicit information through annotations that may not otherwise be noted.

Reflecting upon Adamite, some of the most significant lessons-learned and open challenges from this work that influenced future work were as follows:

- **Context is king**. One may ask "why annotations and why developer documentation?" – the two research areas feel disparate, yet annotations provide a key function that makes them particularly well-suited to addressing documentation barriers, and that is the context that is inherent to the annotation anchoring point. Consider some of the previously-discussed helpful annotations – they were only helpful because their terse message was bound to a rich information context, that being the documentation's text. Documentation issues often stem from a lack of context (e.g., "poor code examples" are often missing context about when to use the code, what the code means, etc.) so providing a vehicle by which additional context can be built upon by users of the documentation is a successful approach. Further, annotations often lost their utility when context was not conveyed well, usually due to poor anchor location (e.g., earlier example about the annotation's text being redundant with the annotated text). In the later works discussed in this dissertation, we further explore the power of providing tight couplings between user-generated or user-derived data and challenging source materials such as code and documentation.

- **What is the incentive to annotate?** When beginning this work, we were motivated by the knowledge that developers sometimes take notes when working on code [154, 156], including when understanding documentation [148]. However, the behavior of the control participants, I believe, presents perhaps a more realistic look at how this behavior manifests in the "real world", with the majority of participants not taking notes or even verbally indicating at what point they would want to keep track of some information. Developing good information tracking practices is difficult for even the most organized and disciplined engineer, so the benefits of using a tool such as Adamite must be high enough that it is worth pausing ones work to annotate. With a system like Adamite that has an implicit crowd-sourcing model, getting enough upfront users to develop a community that can be self-sustaining is a known challenge [6]. Our later work explored this question from multiple angles, including moving working contexts and designing additional tooling to more directly benefit the initial annotation author (Chapter 4) and eschewing direct developer note authoring all-together in favor of capturing already-authored information that may be useful, including web browser and code editing events (Chapter 7) and, later, log statements and values (Chapter 9).

In the creation and evaluation of Adamite, we sought to explore to what extent annotations may help annotation authors and readers in overcoming previously-reported shortcomings of documentation. Through this exploration, we have evidence developers are able to identify documentation issues using annotations and are able to answer some of their documentation questions. Specifically, our participants were able to identify "poor code examples", "incompleteness" and "ambiguity" issues, some of the largest blockers when using documentation [246]. Other developers can make use of these answered questions and issue annotations, with answered questions as the most helpful annotation type and explained code examples also helping annotation readers. However, annotations cannot solve every documentation issue. If the API and its documentation are updated, the annotations may go out of date, at which point they may be more harmful than helpful. While our algorithm attempts to reattach the annotation to its anchor point, the annotation content will not change to reflect that reattachment, at which point the content may be incorrect. This tension when updating content between developer-authored meta-information and the original source it is connected to is further explored with our tools Catseye and Sodalite, along with our design exploration of curating annotations through re-anchoring in Chapter 5.

Chapter 4

Catseye: Meta-Information for Sensemaking About Code

This chapter is adapted from my paper:

[104] Amber Horvath, Brad A. Myers, Andrew Macvean, and Imtiaz Rahman. 2022. "Using Annotations for Sensemaking About Code". In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA*. ACM, New York, NY, USA, 16 pages.

4.1 Overview

With our previous tool, Adamite, we found evidence that developer meta-information, in the form of annotations, was useful in overcoming known barriers in documentation usage. In particular, *later* users of the documentation experienced a significant increase in task completion. However, the *initial* authors did not experience a significant effect. Despite this, we had evidence that developers did appreciate the ability to annotate programming-related documents with their thoughts, questions, and hypotheses that they wanted to keep track of. With this in mind, we wanted to explore how to make annotating more beneficial for the initial user.

One context in which keeping track of information is possibly even more important than in the context of using developer documentation is when actively programming. Prior work has discussed some of the challenges developers must face when making sense of code. A developer must maintain their own task context [168], while also keeping track of the various questions [223] and hypotheses [158] they have about the code, the answers they find to these questions [135, 223], their code locations of interest (commonly referred to as the "working set" [27, 45, 123, 270]), the different versions of the code they try [116, 265], and how those various versions produce differing outputs [116]. All of these information tasks can become even more difficult to manage when a developer is interrupted [153, 187, 189]. Strategies for keeping track of these various forms of information are rarely successful and are

often siloed into different tools or approaches, given the different types of information to track.

In this chapter, we explore the concept of generalizing annotating for capturing different forms of meta-information using a singular, unified mechanism in the IDE with our tool Catseye. Catseye adopts and extends many of the features introduced in Adamite, such as multiple anchors and pinning, while adding in support for programming-specific information tracking, including lightweight versioning and output capturing. Catseye has advantages over other forms of information tracking commonly employed by developers, such as code comments for tracking open tasks [234], by both allowing for a higher level of specificity through anchoring the meta-information to the relevant code yet abstracting that information away from the source code, such that it does not clutter the source with potentially erroneous information. Our lab study suggests that Catseye did assist in information tracking, with participants using Catseye performing better on a debugging task when compared to participants using their own information tracking strategies.

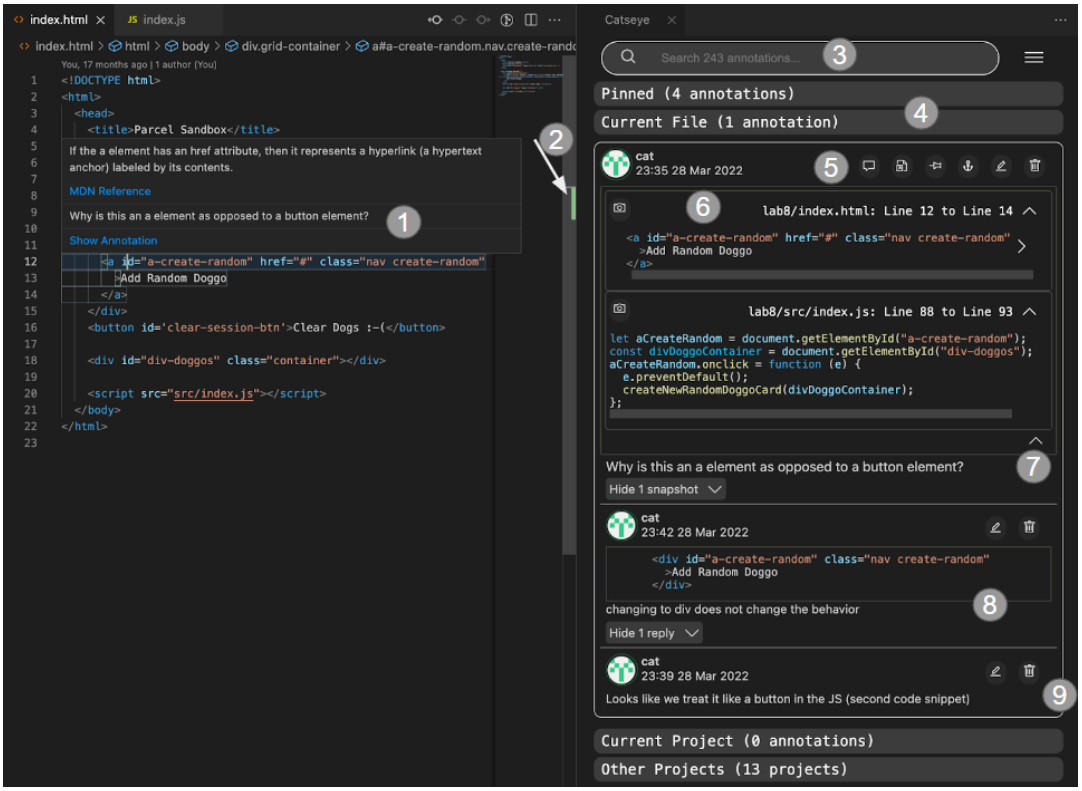

FIGURE 4.1: Catseye as it appears in Visual Studio Code. (1) shows how the annotation appears in the editor – the code is highlighted with a light gray box and, when hovered over, the annotation content appears in the pop-up with any other documentation. Clicking on the "Show Annotation" button opens and brings into focus the Catseye pane if it is not already visible, then scrolls to the annotation. (2) shows the annotation location(s) in the scroll bar gutter in a light green. (3) is a search bar for searching across the user's annotations. (4) is the Catseye pane – the pane is segmented into sections corresponding to the annotations' locations in the file system, with the "Current File" section currently open. (5) is an annotation – the top of the annotation shows the author and creation time information on the left, and buttons for various actions. (6) shows the two code anchors for the annotation. (7) is the content the user added as an annotation to the code snippets. (8) is a snapshot of the code at a previous version with a comment added by the author about this version of the code. (9) is a reply to the original annotation.

4.2 Catseye

4.2.1 Overview of Catseye

We developed Catseye– an extension for the Visual Studio Code editor [164] – that allows developers to keep track of their tasks, open questions and hypotheses, answers to these questions, and more in the form of annotations attached to one or more snippets of code (see Figure 4.1)[1]. We chose to adopt some of the features of the Adamite system [101] for Catseye, given the similar goals of the systems with Adamite based around supporting developers' information tracking on the web. Adamite showed the benefits of multiple anchors and pinning, and we expected that these features would help with code comprehension issues, such as managing a working set. We also introduced code-specific information tracking capabilities, including lightweight versioning and output capture.

To create an annotation, a developer selects a snippet of code in the editor and, using a keyboard shortcut, the context menu, or Visual Studio Code's Command Palette, indicates that they want to create an annotation. The Catseye pane will update with a preview of the annotation, where the developer can add text and choose whether or not to pin the annotation. Once the annotation has been created, it will appear in the Catseye pane and the editor will update with a light gray box around the annotated code at the anchor point (see Figure 4.1-1). With an annotation, a developer can click on it to jump to the anchor point in the code (and vice versa), build upon it through adding additional code snippets as "anchors", capture versions of the code and the code output, "pin" the annotation for easier navigation, "reply" to the annotation with more information, search for the annotation, export the annotation as a code comment, and edit and/or delete the annotation.

Given our high-level goal of creating an annotation system where annotations serve as ephemeral notes when making sense of code, we explicitly designed annotations to function similarly to Google Doc comments or Microsoft Word comments. Annotation anchors in Catseye move around with the code as the code and its location in the editor change over time, and the annotations appear in their own designated area that is detached from the developer's editor. We chose this design metaphor to emphasize the point that annotations are not code comments – they are separate, meta-level notes that are attached to but abstracted from the developer's working context. Annotation anchors update whenever the developer edits their code, and the copy of the code at the anchor that is shown in the annotation (Figure 4.1-6) is updated whenever the developer saves their code.

Annotation code anchors can also be used as navigational aids. The developer can click on the code or the file path in the annotation (Figure 4.1-6 and Figure 4.2-1) which will open that file in a new tab if it is not already open, bring that file's tab to

[1]Note that this figure and remaining figures in this chapter show the version of Catseye that existed at the time of publication and can be considered "version 1" – "version 2", which is the current version, differs slightly in terms of its user interface and can be seen in Chapter 5.

FIGURE 4.2: An annotation made by a participant in our study. (1), (2), and (3) show the 3 different code anchors the participant created across multiple files, with the first anchor ("gameloader.js") as the site of their question, and the remaining two anchors and reply (4) answering their question. The annotation was pinned.

the front if it was not already, and scroll to the code's location in the file. Additional navigational affordances are provided for pinned annotations – a developer can use a keyboard shortcut to cycle through each pinned annotation's location to help with navigating through important code patches. Figure 4.2 shows a pinned annotation one participant in the user study created to help with managing their working set.

Given the mutable nature of code, keeping annotations attached introduced some design challenges unique to Catseye in comparison to other annotation systems designed for more static information. Since code is expected to change, retaining the original anchor point becomes more important as the annotation's content is more likely to become out-of-date – we choose to store a copy of the user's original anchor point for reference as the code changes. The developer can also deliberately save a version of the code by clicking the snapshot button on the code anchor box (see Figure 4.1-6). Once the snapshot has been created, the developer can edit the snapshot to add their own meta-information, such as what output that version of the code produced (Figure 4.1-8). Notably, these design choices were reasonable for a first-pass prototype system, but the tension between annotation and anchor contents is

focused on more in Chapter 5.

Another design challenge is how to handle the case where the user deletes the code that an annotation is attached to – should the annotation be removed as well or should it persist? Notably, this was not a challenge we had to consider with Adamite, given that an annotation author in most cases cannot directly edit the web page and, thereby, delete their annotation's anchor. Different annotation systems do different things in the case of anchor deletion – Overleaf comments will persist when the anchor is deleted, with a line showing where the text used to be, while Google Doc comments will be removed if their anchor is removed. We chose to follow Google Doc's design choice, with the rationale that, if the user wants the annotation content to persist, they can attach an additional anchor to the annotation or export the annotation as a code comment. Again, this design challenge is more directly discussed and approached in Chapter 5.

A related issue is what to do with annotations on code which is copy-and-pasted. Like with deleting, different annotation systems do different things. We decided to *not* copy the annotation with the code with the justification that we want to reduce as much potential annotation clutter as possible (thus we choose to never create an annotation without the user's explicit request). These design choices also turned out to be the most useful way for these features to work in my personal usage of the tool (see Section 4.6). Notably, handling copy-pasted code is addressed more directly with our later tool, Meta-Manager, where the relationship between the copied code and pasted code is retained and tracked but in service of supporting slightly different design goals.

Related to questions of what to do with copy-paste, we also had to consider what to do when the developer uses Visual Studio Code's undo or redo functions. Currently, annotation creation is not put on the undo stack, thus a user cannot undo creating an annotation. Annotations may also be anchored to code that is subsequently undone, thus removed from the code base – this works the same as deleting the code, in that the annotation is removed. However, Catseye currently does not handle redo, thus it will *not* detect if some removed code has returned due to a redo and re-attach the annotation.

Oftentimes, when a developer wants to keep track of some information, they will want to resolve or build upon the information they initially felt was worth jotting down. Catseye follows a similar model to other annotation systems where "following up" on the content of a note is kept very general, so the developer can either edit their original note, or reply to it with their additional thoughts.

4.2.2 Background and Design Goals

In creating Catseye for Visual Studio Code, we were particularly interested in helping developers capture and keep-track of their ephemeral thoughts, questions, concerns, and open action items related to their code, since this is the least well addressed aspect of previous tools. We envision that annotating will help with the

following use cases:

- **Keeping track of developers' questions and hypotheses about code.** Sillito et al. [223] found that, during software maintenance tasks, developers reported over 40 different kinds of questions they had about the code and that there is little tooling support for finding answers to those questions. They also found that there is limited tooling support for helping developers keep track of these questions as they come to an answer. Given that Catseye 's annotations retain the original context of the code, and allow for composition of multiple code snippets through multiple-anchoring, we hypothesize that annotations may help with keeping track of these complex questions and the eventual answer a developer finds.

- **Keeping track of facts developers learn about code.** During any task that requires comprehending code, developers will naturally collect a body of knowledge about the code base [135]. Only some of these facts are appropriate as documentation, either because the behavior of the code is expected to change (such as during debugging) or because there is uncertainty about the veracity of the fact. We hypothesize that Catseye and its annotations will help with externalizing these thoughts while not requiring laborious clean up, since the source code is unaffected.

- **Keeping track of developers' open to-do items.** In trying to complete a complex programming task, a developer needs to keep track of a multitude of both high level goals and lower-level implementation steps in order to achieve that goal which the developer may forget, especially when interrupted [187]. Developers can use annotations to mark the code to change with the details of their "todo" item, compose snippets that are related to the change using multiple anchors, and can pin and un-pin the annotation as a way of marking whether or not the task still needs to be addressed.

- **Helping developers navigate their code.** An oft reported difficulty in programming is navigating the code base, especially when it is large. Developers typically discover a "working set" of task-relevant code fragments [27, 45, 123], then spend time navigating among these fragments as they implement their change. This navigation takes up a large amount of time, especially since these fragments can be difficult to return to [123]. We expect that clustering annotation anchors using multiple anchors, pinning these annotations for easier tracking, and using the code anchors as quick links will make this navigation easier.

- **Keeping track of localized changes.** When a developer is implementing a change, they often try multiple versions of the code in order to investigate the differences in output and ensure that the change works. These changes can be relatively small (i.e., less than 5 lines of code), may not be tracked in version

control [116], and switching between these versions can be difficult if the prior versions are not retained, especially since they may be inaccessible through undo commands [265]. Catseye allows developers to snapshot their code for versioning, such that developers can keep track of the different changes they try and can optionally associate these versions with the output they produced.

- **Keeping track of changing system output.** While testing changes, developers have reported a need for keeping track of what version of their code produced what output [116] and have used strategies such as copy-pasting the output into text files. We provide annotations as a place to store these outputs – developers can either reply to their annotation with the output values or edit their snapshots with the output which comes from that version of the code, thereby leveraging the context of the code.

Notably, the use cases described above are designed for the benefit of the *original author of the annotation*. We chose to focus on supporting the initial annotation author given the goal of supporting a developer's tracking of information, which is largely localized to a single author and their implementation session(s). Whether or not code annotations can be helpful across time or between developers is discussed and explored more in Chapter 5.

4.2.3 Implementation Notes

Catseye is implemented as a Visual Studio Code extension and is written in TypeScript [163] with the implementation using React for the user interface [66] and Google Firestore [58] for storing annotations on the database, along with authenticating the user.

One particularly important aspect of Catseye is managing the code anchors as the user actively edits the code. Annotation anchors are kept up-to-date using Visual Studio Code's document change event handler. Whenever the user modifies a file that contains an annotation, the Visual Studio Code API generates a change event object that we interpret. For simple cases, such as adding a new line at the top of the file, updating the anchors is trivial, but, in the case of more complex changes, such as pulling in a new version of code from GitHub or formatting a code file using a package such as Prettier.js [201], updating the annotation anchor becomes complex. The Visual Studio Code API treats these batch changes as many small edits applied in rapid succession. These rapid, successive edits leads to compounding and cascading errors in terms of the computations used to evaluate an annotation's anchor point. If this occurs, we detect the anchor as invalid and delete the annotation. Given that tracking source locations has been a long-standing challenge in software engineering [206], we explore mechanisms for better handling re-attaching annotations in Chapter 5. Additionally, Sodalite expands upon the code anchoring functionality thru introducing new types of code anchors that are designed to be more flexible, such that un-anchoring is less common (see Chapter 6).

4.3 Lab Study

In order to understand how developers keep track of information while making sense of code when using their own strategies and when using annotations, we ran a small lab study. Participants in the *experimental* condition authored annotations while using Catseye to help them keep track of information, while participants in the *control* condition used whatever strategies they normally would employ. The lab study consisted of a training task, then a debugging task, and ended with a survey to assess the participants backgrounds, their experience with Catseye if in the experimental condition, and their experience completing the task. We chose to use a between-subjects design as opposed to a within-subjects design due to the nature of the task. The study took around 90 minutes, so adding another training session and 45 minute task would make the study too long. Further, as discussed in [125], since these are problem-solving tasks that you can only do once, creating 2 tasks which are independent but of equal difficulty is challenging.

4.3.1 Method

Training Task

Both conditions included a training task using a repository of website templates[2] to either familiarize the participants with Catseye (experimental condition) or to showcase how the participants currently keep track of information when programming (control condition). Participants in the Catseye condition learned how to create and edit an annotation, pin and reply to an annotation, navigate and version their code using annotations, and collect system output, with all functionalities contextualized to how they may be useful for keeping track of different kinds of information. Participants in the control condition were asked to describe how they currently keep track of the different types of information we expect Catseye to support. In this way, we tried to make sure that both groups were primed about the kinds of activities that Catseye is designed to support.

Main Task

For the main task, participants were instructed to understand and attempt to debug a website. Participants were told to imagine that they were a new developer on a team and that they were tasked with understanding and debugging some code. For the first 15 minutes, the participants were not allowed to edit the pre-existing code (but they could add comments and print statements) as part of the scenario in which they are new to a team and should spend time familiarizing themselves with the code prior to contributing changes. This also allowed us to investigate differences in the kinds of annotations made while understanding versus debugging and editing.

[2] https://github.com/ShauryaBhandari/Website-Templates

Game	Bug	Minimal Solution
Both	Unable to Play Games Independently	Change one of the event listeners to a different key (1 value change)
Snake	Screen Does Not Refresh	Adapt Tetris's screen clearing function to Snake (10 line change)
Snake	Snake is Too Fast	Adapt Tetris's timing function to Snake (10 line change)
Snake	Snake is Drawn Incorrectly	Change the constant value for the snake segment length (1 value change)
Snake	Food Collision Check is Incorrect	Change the ORs in the boolean to ANDs (2 value change)
Tetris	Blocks Falls in Last Key Press Direction	Set current direction of block fall to "down" on each game loop (3 value change)
Tetris	Rotating Square Causes Square to Move Upwards	Add conditional to prevent square from being rotated (3 line change)
Tetris	Game Does Not End	Change conditional to whether the stack of blocks is at the top of the screen (1 value change)
Tetris	Game Calculates Score Incorrectly	Increment user's number of cleared rows instead of setting to last clear row value (1 value change)

TABLE 4.1: The bugs present in the two games. "Value" refers to a construct in the program, such as an operator, boolean, or variable.

After the 15 minutes of understanding and testing the code, participants had 30 minutes to attempt to use what they learned to resolve issues they had discovered.

The website included buggy implementations of Snake and Tetris. Each game had 4 bugs, with an additional bug that affected both games, totalling 9 bugs (see Table 4.1). We chose these two games since they are both relatively well-known, are event-based which makes understanding their structure less straightforward, and have clear requirements such that testing the games takes less time in comparison to actually debugging their logic. Similar tasks have been used in related studies [187, 189].

The code was specifically designed to be confusing in order to make keeping track of information particularly important (see Table 4.2). Given prior literature around what makes code confusing [223], we purposefully included bad code smells such as poorly-named variables, global variables, lack of organization amongst methods, and no documentation. Since participants only had 45 minutes for the task, we wanted to necessitate keeping track of information while also keeping the task semi-realistic through using known issues when comprehending unfamiliar code. To further validate the realism of the code, we included two questions in our post-task survey that asked participants how similar the code they saw in the study is to code they have encountered during their time as developers and how frequently they have encountered such code. Participants reported the code is similar to code they have encountered before[3] but that they (fortunately) do not encounter code like this very frequently[4].

4.3.2 Participants

We recruited 13 participants (5 women and 8 men) using study recruitment channels at our institution, and advertisements on social media. Participants were randomly assigned between the control and experimental conditions, with 7 participants in

[3]average = 3.4 out of 5, using a 1-to-5-point Likert scale from very dissimilar to very similar
[4]average = 2.6 out of 5, using a 1-to-5-point Likert scale from never to very frequently

What Information to Track	What Aspect of the Task	Explanation
Questions, Hypotheses, and Answers	Debugging task	Debugging naturally leads to many questions and hypotheses about the program behavior but subsequent answers may be lost or forgotten [223]
Facts	Poorly-written and documented code	Developers are tasked with learning what the code constructs are and how they are used
Open Tasks	15-30 time split	Allow developers to discover many bugs, then have them decide which ones to focus on and how to fix them
Navigation	Poorly-organized code	Constructs, including methods and classes, are spread across multiple files including some files the participant cannot edit
Localized Changes and Output	Arcade Games	By having two arcade games, participants are tasked with making changes and seeing how that affects each game and how that affects each game's output

TABLE 4.2: How the study task encapsulates the types of information Catseye supports.

the experimental condition and 6 in the control condition – participants in the experimental condition are referred to as "P1" through "P7" and control participants "C1" through "C6".

All of the participants were required to have some amount of experience using JavaScript, to have a GitHub account, and to regularly use Visual Studio Code. The participants' professions included graduate students in computer science-related fields, undergraduate students in computer science, and professional programmers. On average, participants had 10.2 years of programming experience, 5.2 years of professional programming experience, and rated their familiarity with JavaScript at 4.5 out of 7. Participants in the control condition had more experience and more professional experience, on average, than experimental participants, but not significantly more.

4.3.3 Analysis

Across both conditions, we objectively coded what bugs the participant succeeded in fixing (see Table 4.1). In the experimental condition, we analyzed the video recordings and log data to count how many annotations each participant authored and how often they interacted with their annotations to assess the utility of the annotations for keeping track of information. We additionally logged whether or not any annotations were made in the first 15 minutes and, for annotations created during the debugging part of the task, what bug the participant was attempting to solve at the time of creation. We analyzed the videos in the control condition to log the same types of interactions including the artifacts developers made in that condition, such as code comments and external notes.

We additionally labeled the annotations and control condition notes with the type of information it was being used to help keep track of. We objectively coded this conservatively based off the content. If an annotation's or artifact's content was phrased as a question or had a question mark, it was coded as a question; if the content had words such as "might" or "seems like", it was coded as a hypothesis; if the content was phrased as an objective such as "change this", it was coded as a task; and if the content was stated as a fact (e.g., "game-1.js is snake") it was coded

as a fact (even if the fact was incorrect). The same process was used for annotation replies.

We counted items as used for "versioning" when they either contained a snapshot (annotation) or were used to mark a change they made to the code base (annotation or control artifact). For navigation, we counted an annotation that is pinned and/or had multiple anchors as used for navigation.[5] We counted an annotation or control artifact as being used for output if the participant used it to store or comment upon the game output. If the content of an annotation or artifact did not fit into any of these categories, it was marked as "Other". For replies, we also labeled whether or not a reply served as an answer to their question annotation – a reply was considered an "answer" if its content was a direct response to the question's content that supported or refuted it.

4.4 Results

Participants in the experimental condition fixed, on average, 1.85 bugs (min = 0, max = 4), while participants in the control condition fixed 0.67 bugs (min = 0, max = 1), a significant difference (*two-tailed T-test, p = .04*). To further explore these results, we investigate what types of information participants chose to keep track of through annotations versus what information control participants used their artifacts to keep track of, how participants used their information when completing the debugging tasks, and how participants performed on the debugging task.

4.4.1 What Information Do Developers Keep Track of with Annotations and Artifacts?

Experimental condition participants created 84 annotations, with each of these participants creating, on average, 12 annotations (min = 6, max = 21, median = 10, std. dev. = 5.446). 44 of the annotations were made in the first 15 minutes (a rate of $\tilde{2}.93$ annotations per minute) and 40 were made in the last 30 minutes (a decreased rate, at about 1.33 annotations per minute). This slight drop-off in annotating is not particularly surprising, given that the types of activities developers were performing changed as they moved from understanding code to more actively writing and fixing code. The size of the annotations averaged 12.2 words (min = 1, max = 45, median = 12.5) and they were attached to code averaging 29.3 characters. Each anchor was, on average, 1.59 lines long, with the majority of annotations attached to one line or less of code (71/84).

Across those 84 annotations, developers had a variety of types of information they chose to keep track of through annotations (see Figure 4.3). The most common usage for an annotation was to keep track of open questions developers had with 28 out of the 84 annotations being questions (33.3%). 16 of these questions

[5]Since multiple anchors and pinning are unrelated to the text content of an annotation, this means an annotation could be marked as both "navigation" and, for example, "fact".

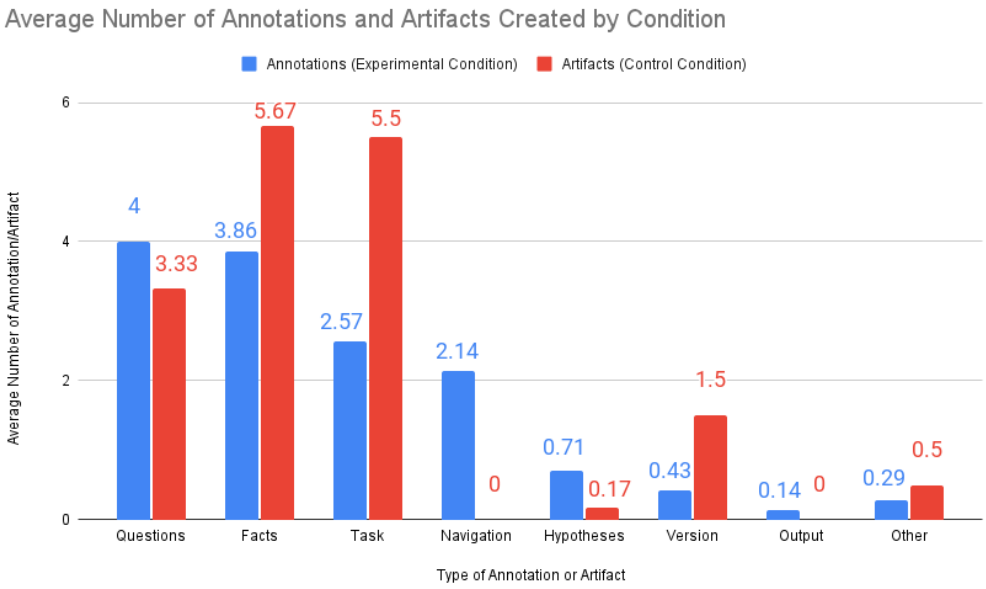

FIGURE 4.3: The average number of annotations and artifacts participants created during the study.

were made during the first 15 minutes, while the remaining 12 were created in the last 30 minutes. 6 out of those 28 questions were definitively answered, while 2 had follow-up hypotheses given program output behavior, and 1 had a follow-up question associated with the original question, resulting in 9 out of the 28 questions being followed-up on in some way. Given the complexity of the task, the amount of answered questions is not particularly surprising, but the fact that participants followed-up on their questions at all provides support for the argument that annotations can serve as dedicated spaces for these questions. In contrast, as discussed below, only 1 of the control condition's 17 questions were followed-up on or answered.

The second most common type of information experimental condition participants kept track of in their annotations were facts they discovered about the code, with facts comprising 27 out of the 84 annotations (32.3%). 23 of these 27 facts clarified information that was explicitly designed to be confusing. For example, P1 annotated `const c = document.getElementById('t')` with "this is the canvas of the tetris game". 18 of these 27 annotations were made in the first 15 minutes, when participants were reading through and understanding the code.

Experimental condition participants also utilized annotations to keep track of their open tasks (21.4% of their annotations) and to navigate the code (17.9% of their annotations). These annotations typically served as reminders to the participant about places in the code base they suspected were related to the bugs they identified in the code. The majority of task annotations were made during the 30 minute debugging phase (12/18) suggesting that there was more of a need for keeping track of their areas of interest in the code once they were developing, as opposed to when they were trying to understand the logic.

Experimental condition participants did not use their annotations for keeping track of their code versions, with no participants using the snapshot feature. Participants did create annotations to comment on parts of the code they added or modified, with 6 annotations made on the participant's own code that they added. Considering that participants, in general, did not edit the code very much, since the bugs did not require large modifications to fix, there may have been less of a need to keep track of small localized changes. Further, the code that participants chose to annotate was usually code that the participants did not edit, with only 12 of the annotations' corresponding code being edited. 3 annotations were used to keep track of output – the small number of output annotations may also be due to the minimal amounts of changes participants made to the code. Further, since the program was a computer game, much of the output changes were graphical which is not output that Catseye can capture currently.

In the control condition, the participants created a total of 100 different artifacts, averaging 16.67 artifacts per participant (min = 2, max = 27, median = 15, std. dev. = 9.771) – which is slightly more artifacts per participant than in the experimental condition, but the difference is not statistically significant ($p = .76$, T-test). This may partially be due to the fact that control participants were primed to think about and show how they keep track of information. Further, all of the control participants had some note taking strategy that they described using in their daily work.

Their artifacts included 78 code comments, 14 external notes (with 5 of the notes being created on a tablet computer, 1 being created in a Notepad document, and the remaining 8 created using pen and paper), and 8 Git commit message[6]. The artifacts averaged 5.95 words (min = 1, max = 22) – notably shorter than the annotations, which averaged 12.2 words per annotation.

The information that control participants chose to keep track of through artifacts differs from the information that was annotated (see Figure 4.3). While, in both conditions, facts, questions, and open tasks were the three most commonly kept track of information types, participants in the control condition kept track of facts the most, while participants in the experimental condition kept track of questions most often. Participants in the control condition also favored "task" artifacts more so than experimental condition participants, with most in the form of "TODO" code comments (66.6%) consistent with prior work [234]. Only one control participant actively attempted to keep track of different versions of their code, and none of the control participants kept track of the output of their code or used any specific mechanisms to help navigate their code, aside from traditional code search (which participants in the experimental condition also used). These results suggest that annotations can promote more entering of questions and answers in comparison to traditional notes and annotations can keep track of other types of information that may otherwise not be captured.

[6] All of the Git commit messages were created by one participant.

4.4.2 How Do Developers Use Their Annotations and Artifacts?

We quantify usage of annotations or artifacts by counting whenever a user interacted with their annotation or artifact in some way. On average, experimental participants revisited 5.71 unique annotations, and revisited their annotations 11.14 times over the course of the study. In contrast, the control condition, on average, revisited 3.5 artifacts a total of 4.67 times suggesting that the annotations were more successful in encouraging participants to follow-up on their information. One of the two top participants, both of whom fixed 4 bugs, also created the most annotations (21) and revisited them the most, revisiting 14 annotations 32 times. This suggests that annotation usage may have contributed to his success.

Experimental participants often revisited their annotations to add replies to their annotations, with participants creating 16 replies and 5 out of 7 participants creating at least 1 reply. Replies typically served as an extension of the annotation's original content, with some annotations serving as answers to the original question (6), hypotheses about the behavior of the code (3), and follow-up tasks that they wanted to complete related to the original annotation (3). For example, P5 made an annotation about Snake where the initial annotation just said "Game 1: Snake" and created two replies, with the first reply explaining what the two Snake-related files did, and the second reply listing all of the bugs she had encountered with Snake – she then revisited this annotation 3 times over the course of the study to keep track of her bugs. Replies also sometimes functioned as places to discuss the behavior of the code *after* the participant attempted to fix a bug associated with the annotated code, with 2 replies commenting on whether their implementation worked or not.

Participants also used pinning, multiple anchors, and anchor clicking as a way of supporting their navigation while working on the task. 5 annotations had multiple anchors, 3 annotations were pinned, and the participants used the anchors to navigate the code base 19 times. In contrast, the control condition did not use any of their artifacts to help them navigate the code base.

Three experimental participants also chose to delete their annotations once they were "done" with them, with these participants deleting a total of 14 annotations. The most successful participant deleted 12 of his 21 annotations over the course of the study – whenever he fixed a bug he would find each annotation that related to that bug and delete it, while keeping open the annotations that were still unresolved. His usage of Catseye suggests that annotations can function similarly to comments in systems like Google Docs where, even if the content of the comment is not necessarily a "to-do item", the comments can still be resolved in a similar manner. Control participants only deleted their artifacts, on average, 0.8 times while experimental participants averaged 2.16 deletions, further suggesting that a Google Docs-style design encourages more clean-up than regular code comments or external notes.

Bug	# of Experimental Participants Who Fixed This Bug	# of Control Participants Who Fixed This Bug	% of Debugging Annotations Made About Bug	% of Debugging Control Artifacts Made About Bug
Unable to Play Games Independently	2	2	12.5%	0%
Snake Screen Does Not Refresh	3	0	12.5%	0%
Snake is Too Fast	2	2	15%	88.6%
Snake is Drawn Incorrectly	1	0	2.5%	0%
Snake Food Collision Check is Incorrect	1	0	7.5%	5.7%
Tetris Blocks Falls in Last Key Press Direction	1	1	35%	5.7%
Tetris Rotating Square Causes Square to Move Upwards	0	0	0%	0%
Tetris Game Does Not End	1	0	12.5%	0%
Tetris Game Calculates Score Incorrectly	0	0	2.5%	0 %

TABLE 4.3: The annotations and artifacts participants created during the study while working on each bug. The experimental condition made 40 annotations while working on bugs, while the control condition created 35 artifacts. The last 2 columns refer to the proportion of annotations made about that bug out of the 40 annotations made while debugging, and the proportion of control condition artifacts made about that bug out of the 35 artifacts made while debugging, respectively.

4.4.3 How Did Participants Identify and Fix Their Bugs?

All bugs were identified by at least 1 experimental participant and had at least 1 annotation created about it, save for the Tetris square rotation bug. No participants in the control condition identified the Tetris rotating square bug or the Tetris score calculation bug, so no control participants made artifacts about those bugs.

When struggling with difficult bugs, participants seemed to create more annotations and artifacts. For example, the "Snake is Too Fast" bug, which required 10 lines of code to change, was only successfully completed by 3 out of the 7 participants who attempted it, and resulted in the majority of control condition artifacts to be about this bug, along with some annotations (see Table 4.3). Conversely, some of the simpler bugs to fix, such as "Snake is Drawn Incorrectly" had fewer annotations made about them as there was less need for participants to externalize their thought processes.

Some particularly complex bugs led participants in the control condition to utilize their notes in different ways than their experimental counterparts did. 3 control participants made a total of 3 external notes that were either visual diagrams of how they thought the games should function or were algorithmic step-by-step instructions for how to design their bug fix. Since the experimental condition created no similar notes, this suggests that future versions of Catseye may better support users by including a way to attach and create visual diagrams, screenshots, or drawings to annotations and support richer interactions for checking off completed steps in an algorithm.

Participants in the experimental condition occasionally made annotations that documented how they fixed a bug, with 4 annotations created for this purpose. For example, P6 wrote some code to try and fix the Snake Screen Does Not Refresh bug and annotated their code with the text "Attempt at clearing the score" and edited their implementation 3 times to try and achieve the correct behavior. P6 then made

2 more annotations on other code snippets that they were referencing when trying to fix their implementation, hypothesizing about how they could adapt the functionality of that code to solve their bug. Their usage suggests annotations can help with marking and documenting code while debugging, including code the user has added that attempts to fix the bug.

4.5 Discussion

Our experiments lend support to the concept that annotations may be used as a lightweight way of capturing and following-up on information that may not otherwise be kept track of when programming. Participants succeeded in creating questions, following up on those insights, and revisiting these notes in order to fix their bugs and had more success, on average, than the control condition.

Four participants asked to continue using Catseye after the study, with 2 participants creating more annotations in their own code after their session. One of these 2 participants reported back on her usage of the tool in her daily work. She found the tool useful for externalizing her "design-oriented notes-to-self" such as "maybe I should do X instead, if I do decide to do that, this is the code that needs to be edited to make it happen". Notably, this is the type of information she says she would normally write down on a piece of paper and *not* use code comments for since she does not want to create clutter that will only confuse her or her collaborators later. She found Catseye valuable for acting as a space for capturing this "thought history" that leverages the context of the code. Her experience lends further support to our claim that supporting thinking through and keeping track of developers' thoughts in a dedicated space when programming is useful.

Participants found the annotating metaphor familiar and understandable, despite the amount of complex activities participants could use the annotations to support, with participants in the post-task survey saying the system was very easy to learn how to use.[7] Prior work has noted that annotations' flexible nature and structure allows them to be used in a variety of ways [5, 21] – we build upon this by showing that annotations can be used in new ways, including to store output, to store and capture versions of code, and as navigational aids. Typically, attempts to support these different activities are siloed into different research tools; annotation systems show a promising alternative where, by utilizing annotations' flexible nature, they can act as a more general "workspace" for storing and thinking about contextualized information a developer cares about.

Another way that participants used annotations as a general "workspace" for thinking was through using their annotation as a "placeholder" when navigating their code. For example, P3 created a question annotation asking why a certain method gets called twice. They followed-up with a hypothesis stating "seems like

[7] "I consider it easy for me to learn how to use Catseye", average score 6.83 out of 7, with 7 being "Strongly Agree"

it's because it calls the draw function, which has some special logic that only occurs if play is true", but, while writing this reply, they paused to continue exploring the file while reflecting on their hypothesis, before returning to the annotation and finishing their initial thought with a guess, "but you still want to call window.requestAnimationFrame i guess?", given what they had learned while exploring. Three other participants paused while creating their annotations to explore the files and think critically about what they were choosing to annotate, which suggests that the choice to have the annotations in their own dedicated pane separate from the context of the code may better support this kind of self-reflection. Notably, these self-reflections have been shown to improve learning outcomes [41].

Some participants in the Catseye condition thought that the output and version capturing features would be particularly useful, despite not using them in the study. Two experimental condition participants reported in the post-task survey that they would use the snapshot feature in their own programming, since they found it difficult to go back to GitHub to see other versions of their code and they sometimes created many small changes that were not tracked in version control. A third experimental participant noted that they wanted to use this feature to capture output when they are performing maintenance tasks like refactoring and need to keep track of many "moving parts" and how their changes affect the behavior of their code. Since running the study, we enhanced Catseye to automatically capture intermittent output through connecting into the Visual Studio Code debugging API to capture and store run time data as replies when an annotated line of code is run.

Two participants in the control condition and 2 participants in the experimental condition created artifacts that, while phrased as a fact, were incorrect. The control participants added comments above functions incorrectly stating what the functions' purposes were. The 2 experimental participants incorrectly assumed what a function and variable were used for, respectively. These annotations, while incorrect, are only visible to the original annotator, while the code comments could, in theory, be viewed by any collaborator, which could potentially misinform them. Even if the annotations were viewable to collaborators, they would not be in the code acting as documentation, lessening their potential to be harmful. An incorrect annotation could be a learning opportunity with the reply feature, where a collaborator could clarify or correct a misinterpretation of the code.

Two control participants had problems managing their code comments. C1 created a comment noting that a particular part of the Snake code looked like it was used for initialization, then discarded the commit that contained that comment. 10 minutes later, they searched for that comment, forgetting that they had discarded the comment. C2 marked a part of the code with the comment "REVISIT" but then undid a series of changes in order to revert to an older version of the code, removing that comment in the process, and then never revisited that part of the code. Participants in the Catseye condition did not create any code comments (2 participants started to make code comments before removing them and manually converting

them into annotations) and did not lose any of their annotations during the study – annotations' meta-nature may serve as a safeguard from erroneously removing them. However, Catseye users could, in theory, lose their annotations by erroneously deleting the code with which the annotation is associated. Currently, Catseye does *not* put annotation creation, editing, or deletion into the Visual Studio Code undo stack, another area where other annotation systems differ – Overleaf similarly does not, whereas Google Doc puts comment creation in the undo stack (but *not* comment editing or resolving!). In my own usage of Catseye, putting annotation creation on the stack would be beneficial, given that I normally annotated code I had just worked on – this means that performing an undo operation is likely to modify the code anchor the annotation corresponds to, which may change the annotation's meaning or cause it to become detached if the code did not exist in the prior edit. Annotation deletion would also make sense to go on the undo stack, especially in the case that the annotation is deleted because its code anchor was deleted, given that an undo operation would bring back that code anchor, thus should bring back the annotation. This is how similar systems, such as Google Docs, works.

4.6 First Author Usage of Catseye

I used Catseye while developing Catseye as a form of "dogfooding." Anecdotally, we report on the annotations created by myself using the same methodology of labelling each annotation by the primary type of information it was meant to keep track of. We omit annotations made without any text content, as they do not have enough context for labeling, and annotations made purely for testing the application, considering they do not represent "real" usage of the tool. All of these annotations were created by myself to help myself, either in the short term (so they were ephemeral) or for when I returned to the code later.

Over 12 months, I created 182 "real" annotations in the Catseye repository across 25 source files, with each annotation averaging 28.95 words (min. = 2, max. = 143, std. dev. = 27.99). 42 annotations had a total of 58 replies, 8 annotations had 18 snapshots, 3 annotations had multiple anchors,[8] and 9 annotations were pinned.[9] 110 of these 182 annotations were deleted as I finished open tasks and iterated over the code, including removing the code the annotation is anchored to which deletes the annotation.

The content in my annotations differs slightly from the annotations created in the lab study, perhaps due to the different nature of development. The most common annotation type was "task" type annotations, with 48 of the 182 annotations

[8]Multiple anchors were added late in development, so the lack of multiple anchor usage is primarily due to the short amount of time to use the feature.

[9]This is a conservative count, considering we used our log data and the log does not count whenever an annotation is pinned or un-pinned, just whether the annotation was pinned the last time it was updated in the database.

reporting some open action item I needed to act on – in contrast to the task annotations made in the study, these task annotations served as reminders for places to change when performing maintenance tasks like refactoring, as opposed to parts of the code that may have a bug. I also created many question annotations (47 out of 182) – 12 were replied to, with 4 of these replies answering the original question, and 17 questions were deleted. These questions typically pondered the system behavior, previous design choices, or details of the Visual Studio Code API. The third most common code was "Other", with 16 annotations. All of these "other" annotations served as a reaction to the code, with these reactions sometimes pondering the former design rationale and sometimes expressing frustration with implementation challenges. For example, one annotation, anchored to a particularly confusing function said "This is a pain". No annotations were made like this in the lab study, suggesting that actively implementing and writing code may elicit different types of information than code understanding and debugging tasks.

Actively using Catseye also led to changes in the tool's design. Initially, an annotation would be copied if the developer copied the annotation's code anchor point. However, given that code was often copied to be used as a template, in active development, copy-pasting the annotations resulted in duplicated annotations with irrelevant content as the code changed, which I found to be more distracting than helpful. This experience informed our design decision to not copy the annotation content when the user copy-pastes the anchor code. However, there may be situations in which it would be better to copy the annotation with the code, such as when the annotation serves to document behavior about the code. Future versions of Catseye might benefit from allowing the user to choose whether or not to have the annotation copied with the code, or alternatively, to add a new anchor to the original annotation.

I found the tool useful for externalizing and thinking through problems and stopped using code comments in favor of annotations. The ephemeral nature of the annotations was particularly useful in the development of Catseye since the code was nearly constantly changing which lead to a lot of uncertainty about the design and implementation details – information that I did not want to commit to the code base as comments in case the information ended up becoming obsolete (a common problem with code comments [71, 199, 237, 238]).

4.7 Conclusion and Future Work

In our development of Catseye, we found evidence that note taking and collecting of code-related meta-information was helpful for developers when tracking information. Further, we found evidence that these activities could be supported using a singular, lightweight interaction design – annotation. These findings extended our previous findings from Adamite by generalizing our approach to a different developer sensemaking domain while tracking other types of information. Below, I briefly

reflect on some take-away lessons and challenges from this work that moved the research forward:

- **Working in a Dynamic Environment with Static Data Introduces Complexity.** Before, in Chapter 3, I claimed "context was king" and, as we showed in this work, it still is. Annotations and their anchor points were commonly revisited and helped both myself and our study participants think through and reason about problems that were subsequently solved. However, with a mutable source document, "context" is now multi-dimensional – the source location and the surrounding information is constantly changing across time while the annotation, if a developer chooses not to edit it, is stuck back in time. We largely, albeit unintentionally, avoided this challenge in this study, given that participants did not edit the source code very much and the study was a short, finite window of time in which the annotation could not "age" out of its context. However, in my own usage of the tool, I experienced firsthand how annotations, while useful in the moment, would eventually lose their context and become clutter, either due to my memory slowly eroding over time or the source code changing so much such that the annotation content itself no longer made sense. Small-scale design decisions, such as choosing to delete an annotation when its anchor is removed and not copy-pasting an annotation if its anchor is copied, helped a bit but are not enough to fully address the problem. How do we build systems that are designed with information mutability in mind? Chapters 5 and 6 probe on this challenge.

- **Lightweight unification of tooling that operates at a meta-level is a rich and relatively untapped market.** Annotating, as a practice, has largely been localized to and studied in the context of reading materials. However, choosing to apply annotating to a dynamic space opens up a myriad of opportunities for more domain-specific tooling. While this chapter only explores source code and some of the types of meta-information you may want when working with source code, such as code versions or outputs, one can imagine generalizing the design approach put forward by this work to other domains. For example, Figma has a commenting system, but it is relatively limited – what if we "supercharged" these comments to additionally version themselves and the designs they are attached to to help with reasoning about design rationale over time and across collaborators? What if these comments could be used to extract and aggregate different parts of a design together to create a new prototype? I believe part of the reason commenting and annotating are so ubiquitous across information authoring and management platforms is because there is a lot of power in being able to directly point at something and say "this is what I am talking about". Information closeness is a strong undercurrent of this work and Catseye and Adamite show the power of this in both communicating across time (Adamite) and in binding otherwise-disparate pieces of

information together (Catseye). For those interested in building their own annotation systems, I strongly suggest understanding the types of information management tasks your clientele are actually trying to complete when annotating and more directly supporting those tasks through increasing both the types of information annotations can hold and giving more functionality in terms of what an annotation can do.

Much of the success of the annotation approach can be attributed to the localized nature of the annotations and their connection to the source material through their anchor point. This connection allows for the note to serve as a navigational way point, allows for commenting on different types of code with the same interaction method (e.g., commenting on a variable within a line of code versus commenting on a whole conditional that spans multiple lines), and supports code-specific activities including versioning and output capture. Despite these powerful attributes, much of that power stems from the tight coupling between the annotation and its anchor point – thus, that power is lost when the annotation becomes unanchored. We further explore questions of anchoring, along with longer-term benefits and management of annotations in the next chapter on annotation curation. We then extend our code anchoring support and the types of information that may be attached to the code anchor points through extending Catseye into a long-form documentation tool, Sodalite.

Chapter 5

Curating Ephemeral Meta-Information: An Exploration of Catseye Annotation Management

5.1 Overview

In our creation and development of Catseye and Adamite, a constant design question was how to manage the *scale* of annotations, how to deal with *changing source content*, and *what to do* with annotations across time to increase their utility, given their ephemeral nature. These challenges are not dissimilar to challenges encountered with other "information scraps" (a class annotations fall into) [21, 22] – namely, the ad hoc nature of creating and using information scraps is often at odds with the design of traditional personal information management (PIM) tools, which typically prescribe an organizational and rigid structure for managing these scraps [21, 248]. "Information scraps" are characterized both by their form (e.g., lightweight, often captured on various platforms across different modalities) and by their intent (e.g., sticky note to *remind* you of an upcoming appointment, Google Doc *reference list* of URLs linking to recipes[1], etc.). These scraps are most often used in the moment as a form of cognitive offloading and externalization, but occasionally are meant to persist indefinitely (such as in the case of the reference list – a living document) or until a set time (such as with a reminder of an appointment).

Annotations and our annotation platforms, Catseye and Adamite, both require and allow for some amount of structure (i.e., requiring an anchor point for contextualization and, in Adamite, supporting organization through types and optional tags). However, the content of these annotations and what to do with the annotations falls into the information scrap challenge in that their utility, especially over the long term, is often not obvious (e.g., is a developer using Adamite to annotate

[1]This example is taken from my lived experience – my fiancé and I have one of these documents and it is chaotic, to say the least!

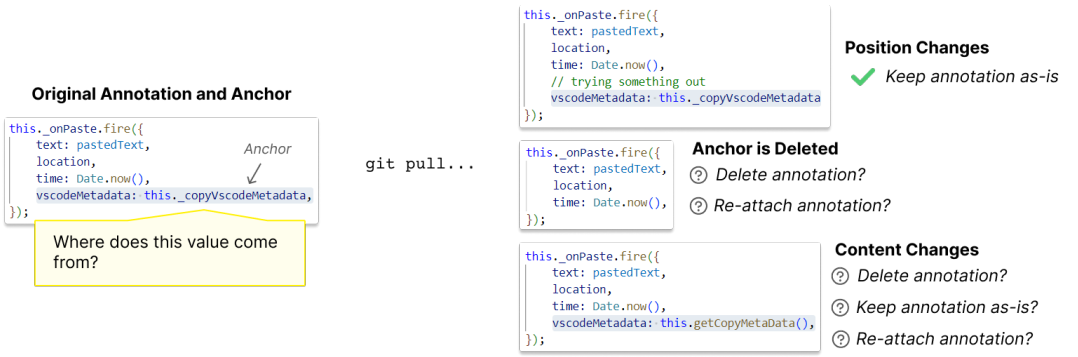

FIGURE 5.1: An example scenario in which an annotation's anchor is updated in multiple ways after a `git pull`. In the case only the position changes, the annotation content should stay as-is and the anchor will update to represent the new start and end positions. If the anchor is deleted, it is unclear whether to delete the annotation or to re-attach the annotation at a different, semantically similar point. Likewise, if the content within the anchor's bound changes (`this._copyVscodeMetadata` to `this.getCopyMetaData()`, in the figure), under different circumstances it may make sense to delete the annotation, keep the annotation on the new anchor content, or re-attach the annotation elsewhere.

this API method as a *reminder* to use it later? Or to mark it as *obsolete*?). Annotations have the additional challenge of being dependent upon a stable anchoring point, which must remain constant or, at least, semantically equivalent to the original anchor point's content, which introduces additional design challenges in terms of determining what to do with these information scraps after some time has passed and the associated document has changed. Lastly, another design challenge that differentiates our annotation systems from prior work around information scraps and PIM tools is that our tools are meant to be used *collaboratively*. This leads to the challenge of scaling these systems to a point where potentially hundreds of annotations are shared to accelerate other users sensemaking of documentation and code. Given enterprise level systems, with potentially hundreds of engineers using these systems on a shared code repository with Catseye, how does one ensure all annotations are versioned properly, are useful, and have consistent and stable anchoring points?

Given this tension between retaining the pros of annotating as a lightweight way of expressing an idea when sensemaking while ensuring that our design is scalable and robust to changing documents, we began a design probe on how to modify Catseye to support large-scale curation. Our curation work is two-pronged – one prong explores how to deal with annotations *en masse* as a way of cutting down on potentially hundreds of irrelevant annotations, e.g., to throw away after their moment of utility is over. Another prong explores annotation's connectivity to a source document and how to algorithmically re-attach the annotation across numerous scenarios, as an additional form of curating one's set of annotations (see Figure 5.1). We then implemented these designs into Catseye– all screenshots in this chapter represent the current version of the Catseye system (in contrast to the screenshots in the previous chapter). Through this exploration, we created a set of prototypes and design insights that informed our later work, especially with respect to Sodalite.

5.2 Background and Related Work

Management of lightweight information, such as notes and annotations, has been extensively studied in both the fields of psychology and Human-Computer Interaction [14, 21, 22, 25, 54, 133, 213]. Researchers in psychology have largely studied this phenomenon with respect to both the cognitive underpinnings of how humans make sense of and organize this information [14, 82, 134, 213] and in terms of the motivation behind creating these information scraps [82, 113, 133, 134]. HCI research has explored this phenomenon from a more applied standpoint, including how users collect and create this information [54, 81, 113], annotate it [81], and use it for later tasks [25, 83, 121]. A predominant theme across most of this work is that the cost of externalizing this information must be kept low [54, 113] (otherwise, users have been known to favor pen and paper [54]) and mechanisms for organizing and curating the information must be flexible [54, 114, 248]. Curation typically occurs when a user wants to transform their information scraps into something that can be used by another person, such as a blog post [154] or decision table [148]. A challenge in curation is that it is typically cognitively expensive and time-consuming [149] and a user's mental model of a problem space is often evolving, so the curated representation can become out-dated [69, 95, 121]. However, failing to curate these sets of information scraps can result in the information becoming intractable [54].

Less work has explored annotations, specifically, and their relationship with source documents (i.e., anchors), especially with respect to curation. A prior literature review [5] found that the flexible nature of annotations allows them to serve a variety of purposes, including supporting in-context commenting and creating connections among parts of text. Other work noted that annotations may be seen as a conversational tool among the document users, as well as with the document creators [72, 85]. The nature of annotating and its tight coupling of user commentary and source document has made it useful in a variety of crowd-sourcing contexts, such as for question-answering on web pages [42, 99, 101] and in classrooms [67, 74, 275]. However, most of these works have been in the context of static materials, such as books, and *not* dynamic materials, such as source code, which inspired us to further explore this often-overlooked challenge in utilizing annotations for long-term information collection and sharing. Our work expands on this annotation and information management work through exploring mechanisms for both automatically re-attaching annotations given document changes and curating annotations with user interaction techniques.

5.3 Preliminary Study of Catseye Annotations

As an initial exploration of what annotations to curate and how, we explored annotations already created naturally by myself as part of dog-fooding Catseye (see Section 4.6). A subset of these annotations (122 out of the 181) were created from the fall of

2021 to the spring of 2022 and were developed while implementing and debugging various parts of the Catseye code base. In anticipation of our summer undergraduate researchers joining the research team, I hand-coded each authored annotation by whether or not I believed it would be useful to someone unfamiliar to the code base. This was done in service of finding patterns in terms of the annotation's structure and content and whether or not I believed it would be beneficial. I chose to do this coding independently for multiple reasons – for one, this internal decision-making of whether or not to "share" a potentially "helpful" annotation is normally how such curation would happen "in the wild" [74], as people are not typically working together to decide whether or not to make an annotation public. Further, determining whether or not an annotation may be helpful requires an intimate knowledge both of the annotation (which can be difficult to understand given their terseness) and the code base (which, at that point, had only been developed by myself). Only annotations which were not already-deleted were included in the analysis since deletion suggests the annotation was no longer relevant to me, let alone relevant to someone else. Signals of usefulness included whether or not it documented some part of the code, how comprehensible the annotation was (with many of my annotations being informal in nature), and whether the code and annotation content were still relevant or not. Considering I could not definitively say whether or not each annotation would be helpful for the summer students, the codes were indefinite and included the codes "Probably", "Maybe", and "Probably Not". For each determination, I included an explanation as to why I believed the annotation was probably, maybe, or probably not helpful, and, for situations in which I thought the annotation could be helpful (i.e., "Maybe" and "Probably" codes), I included an additional code for in what context I believed the annotation could be helpful. Working contexts in which I anticipated annotations could be helpful for included software maintenance, extension testing, API usage, understanding design rationale, or feature requests.

Through this preliminary analysis of the 91 non-deleted annotations that existed in the code base at the time, the majority of annotations were "probably not" helpful (49), while 33 were "maybe" helpful, and 9 were "probably" helpful. The most common reason for annotations not being helpful was due to the content no longer being relevant, either because the code had changed enough such that annotation's content was out-dated (18) or because the task or bug the annotation was referring to was complete (16), suggesting that a mechanism for easily removing completed task annotations, like Adamite, would be useful in Catseye. However, detecting code-versus-annotation content inconsistencies is less trivial and part of the inspiration of our re-anchoring work. Additionally, some annotations were marked "not helpful" because the way in which they were written was too informal to be easily shareable by myself and comprehensible by another developer (11). Prior annotation research in the classroom context has discussed that sometimes, when moving an annotation from private to public, annotation authors had to go through a pre-processing

step prior to making their annotations public, in which they typically make the annotation longer and more formal by expanding on ideas that were previously just terse reminders to themselves [74]. Such a step may also be required when curating annotations prior to sharing them for code comprehension.

The 9 annotations I had the most faith in potentially being helpful were annotations related to specifications on how to implement or maintain some part of the code base (5), how to use some part of the VS Code API (2), or documented some unintuitive part of the code base (2). Similar to what we saw in the Adamite study, the majority of the potentially helpful annotations were answers to questions I had at some point (4/9) and, like higher-quality Stack Overflow answers [175], 2 of the 4 referenced external resources I used that informed my answer. 2 of the 9 annotations used the annotated code itself as an example of how to complete a task (e.g., "it's really this simple to authenticate with github") while 2 other annotations bound a task to the code as a reminder (e.g., "Need to update this [code] too whenever we update the annotation model") – all 4 of these examples leverage the tight coupling of information and code to transform the code into a representation of an implementation constraint or code example. These qualities make these annotations seemingly more appropriate to be shared than others, thus ones to curate for other users.

The remaining 33 annotations were less obvious as to whether or not they should be shared. A common theme across these annotations was a lack of certainty in the subject matter, with 8 being questions and 5 hypotheses. Notably, this is similar to a finding from the Adamite study, in that unanswered questions were one of the least helpful types of annotations. Other annotations would only be potentially helpful if the student was working on that particular part of the code base and would otherwise be too confusing to be of any use (10). For example, 6 of the 10 annotations were related to how the anchoring algorithms did or did not work. The mechanism for keeping anchor points up-to-date while a user is actively editing code is easily the most complex part of Catseye and also the most prone to bugs, leading to many annotations about those parts of the code base – if the student was working on anchoring, then these annotations could be helpful, otherwise they may draw attention to this part of the code base that is difficult to understand. In this way, curating annotations for later developers may need to change dependent upon what task the later developer is performing and whether or not the annotations are relevant to them, with an annotation's location a particularly powerful signal of relevance. Other reasons for annotations being "maybe" helpful included acknowledgements of potentially unintuitive design and implementation decisions (9) and potential feature ideas (4).

Through this exercise, we came up with the following design considerations when supporting annotation curation:

- Annotations, in order to be shareable, must be ***relevant*** to both the current code in the IDE and to the future reader.

  - This suggests a need for both author-side and reader-side curation functionalities.
  - This suggests a need for the relationship between the annotation content and code content to be algorithmically or deterministically evaluated, prior to showing the annotation to readers.
  - Relevance is often lost over time as code is changed, tasks are completed, and so on.

- Some amount of pre-processing must be done by the annotation author as part of the curation, prior to sharing with later readers.

- Signals such as answered questions and annotations that contain reference URLs suggest higher-quality annotations that may be candidates for sharing. Signals of lower-quality annotations include lack of certainty (e.g., "I'm not sure about this...") and a lack of formality in tone.

5.4 Design Probe: Catseye Annotation Curation through Re-Anchoring

Initially, we did not consider the act of (re-)attaching or removing an annotation given the source code changing (hereafter referred to as "re-anchoring") a form of curation. Instead, we considered it a technical problem – how do we find the most appropriate point to re-attach an annotation given a change in which the original code anchor was removed, moved to a completely new location, or changed in such a way the code is semantically similar but textually different (e.g., `myList.forEach(t => t + 1)` to `for(const x of myList) x + 1`)? While this technical component of the project is, indeed, a challenge, upon reviewing the Catseye annotations (including how many were no longer relevant due to code changes) and considering various scenarios in which re-anchoring is necessary (Figure 5.1), it became clear that the mere act of choosing to either remove an annotation or re-anchor it given a code change is a form of curation. With this in mind, we sought to both design an algorithm that could find potential re-attachment points and a user-interface for re-attaching or removing the annotation, given uncertainties around whether the annotation should persist or not.

5.4.1 Algorithmic Re-Anchoring

We designed our algorithm with the goal of finding the most-likely re-attachment points, given an unanchored annotation. We considered an annotation to be unanchored if its last logged anchor point that we stored in the database was either invalid[2] when attempting to attach on system launch or the text that the anchor contains is too dissimilar to the last-logged anchor content.

If an appropriate match is not found at the original anchor location (Figure 5.3-1), the search expands outwards to the 5 lines above and below the last-known location (Figure 5.3-2a and 2b), given research that source locations, given a change, are often within a small line range [206][3]. At this point, we begin utilizing our text and distance-matching algorithm to determine whether the searched range of text contains a reasonable, potential match.

The algorithm begins by tokenizing the last known anchor string by stripping white space and syntactical symbols. This same tokenization is performed on each code line within the search space. We first see whether there is an exact match in terms of token content and order within the search space – if so, we return the source code line's line numbers and offsets. If no exact match is found, we attempt to find the closest match.

For the closest match, we combine a variety of heuristics in order to calculate closeness. This includes the edit distance between each token in the anchor text and the source document text, the distance from the last known location of the anchor and the source location, and the similarity of the surrounding text in the source document to the surrounding text that is saved in our database. The surrounding text is used in combination with the saved anchor text in order to prevent the algorithm from favoring common keywords, in the case the user annotated code composed primarily of said keywords (e.g., `for`, `while`, `function`, etc.). Given these heuristics, a weighted average is created for each token, given its comparison point, and these weights are averaged by line. The algorithm then chooses the sequence of contiguous tokens that yields the highest average value – if the anchor is above a certain threshold, the anchor is added to a list of candidate anchors for the user to review. This process is repeated for each line within the search space. If no anchors meet the quality threshold, the search is expanded to the code within the parent node of the AST (Figure 5.3-3) and, if that search also does not yield sufficient anchors, the search is expanded to the whole file.

There are situations in which our algorithm may not work. One feature of Catseye is that the user can choose to annotate any code of any size, whether or not

[2] An anchor can be considered invalid when one or both of the bounds of the anchor (i.e., its starting line, starting offset, ending line, and ending offset) are not contained within the document's starting and ending lines and offsets.

[3] Note that this is contingent upon Catseye having been run semi-recently, thus having a recent "memory" of where the annotation should be anchored. In situations in which the code has changed many times but Catseye has not been used, we additionally keep track of the last known Git commit the annotation was on, making it possible to move forwards in time along the commit tree and run the re-anchoring algorithm at each commit.

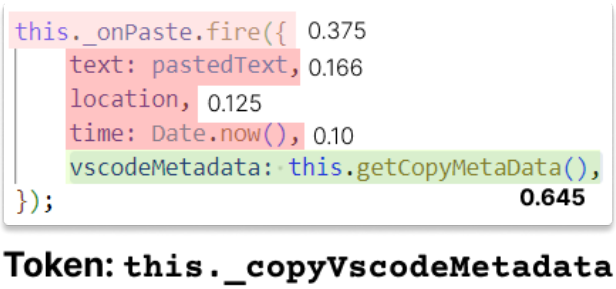

FIGURE 5.2: How running the matching algorithm at the token level with `this._copyVscodeMetadata` across multiple tokens in multiple code lines would yield different results, given the string difference and location difference. Note that the most reasonable re-anchoring spot from Figure 5.1 has the highest match score.

it is syntactically correct – this means that the user can choose to annotate, for example, just the letter "u" in the word "function". While it is unlikely a developer would choose to do that, the terseness of the anchor would make finding candidate anchors more difficult. We attempt to mitigate this issue through utilizing the surrounding code and taking the anchor's original location, including line and offsets, into account.

Conversely, a user may choose to annotate a large amount of code, such as a whole file or function. Fuzzy anchoring [50] the text, given that amount of content and taking into account the additional code-structure information we have (such as what the parent AST node is and what the original name of the method was) should yield reasonable results, but it unlikely the anchor content will ever be exactly the same in these cases. The algorithm may not find reasonable candidate locations if the file has changed too significantly since the last time the annotation was anchored. This is why we chose to show the user candidate locations and allow for other reanchoring options, including manually choosing a new anchor or leaving the annotation unanchored, given that it is difficult to determine if the annotation should be deleted or not in that case. Additionally, we have reason to believe that both of these types of anchors (too small or too big) are not particularly common, given that most anchors in our Catseye evaluation (Section 4.4) averaged around one line of code.

5.4.2 User Interface for Re-Anchoring

Upon finding a set of reasonable new anchor locations for an unanchored annotation, the user interface for Catseye updates such that the user can curate these annotations (see Figure 5.4). When reanchoring an annotation, a user can view what anchor candidate(s) Catseye automatically found on launch (Figure 5.4-1 and 5) or choose to leave the annotation unanchored (Figure 5.4-2). In this way, we wanted to support the lightweight nature of Catseye by not forcing a user to act on the annotation if they do not want to. Catseye also sorts the candidate anchors by how

confident it is about each anchor given the weighted averages, with the highest-weighted anchor shown first. In the case of many candidate anchors, the system only shows the top 10, such that users are not overwhelmed with many potentially incorrect choices. The system additionally states its confidence in the top-left corner of the suggested anchor (in this case, it determined this anchor to be a "very similar match"). Additionally, if the code has either changed so significantly that none of the system-suggested anchors are correct or the user wants to re-contextualize their annotation to this new working context, they can choose to manually reanchor the annotation at any arbitrary point within the code base by selecting some code and clicking the "Manually Reanchor" button (Figure 5.4-3). As discussed in our earlier scenarios (Figure 5.1), there may be situations in which there is no longer an appropriate anchor point, in which case the annotation can be deleted or left unanchored.

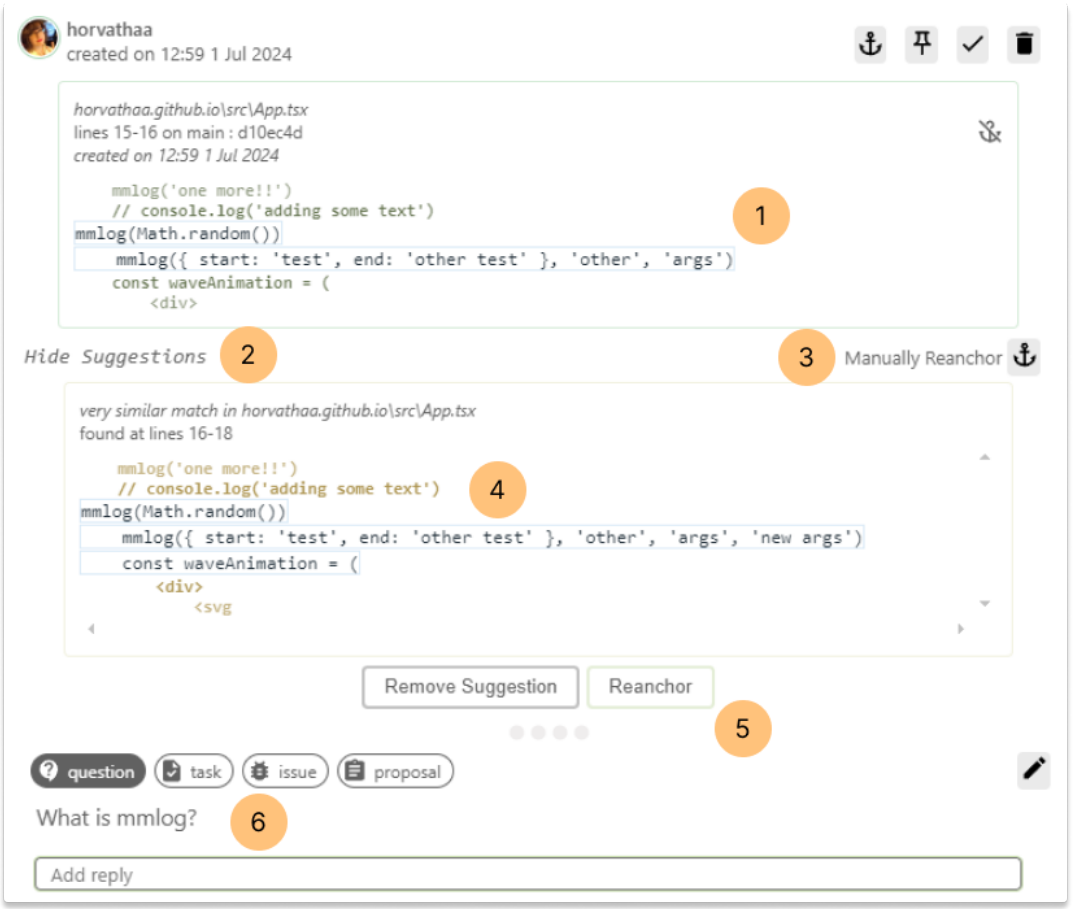

FIGURE 5.4: The user interface when re-anchoring an annotation. (1) shows the last-known anchor, along with the surrounding code in green. (2) is the Show/Hide Suggestions button, allowing the user to leave the annotation unanchored, if the user so desires. (3) is the Manually Reanchor option – the user can select some code in the editor, then click the "Manually Reanchor" button to set the anchor to their selected code. (4) is the first of multiple candidate anchors. (5) are the options for what a user can do with a candidate anchor – they can either remove that particular anchor as an option, "Reanchor" their annotation to that anchor, or click through the carousel (i.e., 4 gray dots) to view their other options. (6) is the annotation.

FIGURE 5.3: How the re-anchoring algorithm expands its search outwards, given no sufficient match at each step. (1) is the original anchor location, (2a) and (2b) are the 5 lines above and below, respectively, the original anchor location, (3) is all of the code belonging to the parent node of the original anchor in the AST, and (4) is the whole file.

To further support these anchoring activities, the design of Catseye anchors was updated to include additional meta-information about the anchors to assist in curation. Anchors now display their state via the anchor icon in the top right corner to further clarify to users whether their anchor exists and is being tracked currently (i.e., anchored) or not. Catseye will also show the last-known location of the anchor, along with what Git branch and commit this anchor was last seen on and the time at which it was created (Figure 5.4-1, top left corner). While the text surrounding the anchor is primarily used as supplementary data for the reanchoring algorithm, we also show it to the user to help them in placing their anchor in context when reviewing their annotations – candidate anchors also include this information to help users in visually assessing whether or not the anchor seems correct.

In designing re-anchoring capabilities for Catseye, a number of design challenges arose. One interesting consideration was whether or not to even have the user review candidate anchors when their annotation becomes unanchored. Indeed, given high enough confidence in a candidate anchor, it may be an overall better user experience to simply have the system place the annotation at a new anchor without requiring the user to confirm the new location or informing the user it performed this re-anchoring. Ultimately, we decided to loop the user into this re-anchoring process whenever the system does not find an anchor that exactly matches the last-logged anchor. We did this not only to confirm that the resulting anchor is correct, but also as a forcing function for users to perform curation on their annotations. We hypothesized that the point at which an annotation becomes unanchored, such as at a `git merge` or `git pull`, is also a likely time at which the annotation is no longer

relevant. We believed this due to the fact that the most common reason an annotation was marked as "probably not" helpful during review was due to the annotation being no longer relevant and this lack of relevance most often stemmed from the code having changed too much. By allowing the user to actively participate in the curation process when this change occurs, we expect that problems noted in earlier PIM and "information scrap" research where users get "lost" amongst a sea of uncurated and irrelevant notes can be avoided [54].

5.5 Design Probe: Catseye Annotation Curation through Batch Processing

As discussed in the introduction to this chapter, a key feature of information scraps is their ephemeral nature. However, when programming, a developer is already under a large amount of mental stress and, thus, doesn't necessarily have the mental bandwidth to not only manage their code but also their annotations to throw them away. Thus, we sought to improve Catseye's design to make this activity even more lightweight.

With this in mind, we sought to provide functions for either increasing the utility of an annotation for a later reader or easily removing it from consideration. In this way, we see annotations as a mid-point in terms of a developer's individual sensemaking journey in which there is a chance the information they gain will no longer be relevant, thus should be thrown away, or is a piece of evidence in a longer stream of information (or information "quest" [87]) that can be later transformed and structured into something useful. To lessen the cost of performing either of these activities, we introduce batch processing of annotations into Catseye. This batch processing works both in tandem with structuring activities, such as adding annotation types (taken from Adamite), and with culling operations such as deletion.

We chose to focus on batch processing over other types of potential information management processes for multiple reasons. The predominant reason is batch processing lends itself nicely to the information management practices we saw in our studies of Catseye and Adamite along with prior research (e.g., [121], [148]) – there is a common pattern in which people performing sensemaking forage for and generate information for a while, then, upon reaching a reasonable stopping point, pause and revisit the set of information amassed during their sensemaking episode. For example, in our Catseye study, one participant, upon fixing a bug, would go back to each annotation they created related to that bug, and either delete the annotation or respond to it to "conclude" that thread of inquiry. Batch processing works well in this paradigm in that each "set" of annotations created during some sensemaking can be treated as a batch and operated on to perform the types of activities we hypothesize would be helpful. Other reasons for choosing to explore batch processing includes the ability to adapt already-known successful information management UI

techniques such as filtering and because Catseye's list view of annotation lends itself nicely to acting on items in aggregate.

5.5.1 User Interface for Batch Processing

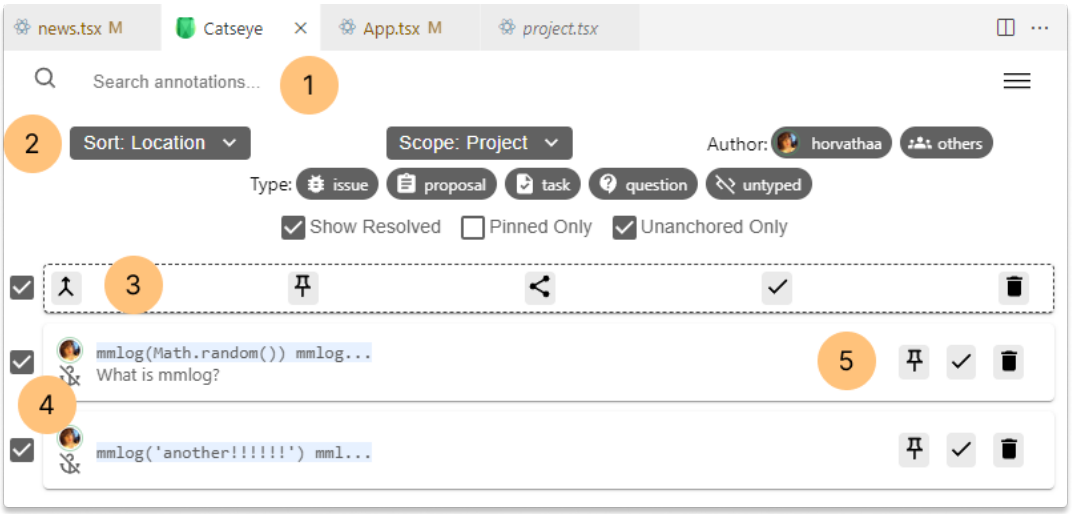

FIGURE 5.5: The user interface for batch operating on annotations in Catseye. (1) is the search bar, (2) is the set of sorting and filtering operations including sort by location or time, scope of annotations to include (including annotations made on this file, this project, or across all projects), who authored the annotation, what annotation type it is (similar to the Adamite annotation types), and whether or not the annotation has been marked as "resolved", is pinned, or is anchored. (3) is the set of batch operations a user can perform on any selected annotations, with options for (from left to right) merging (Figure 5.6), pinning, sharing, resolving, or deleting. (4) are collapsed annotations which are currently checked, meaning they will be included in any batch operations selected at (3). (5) are the buttons for operations a user can perform on a single annotation.

The user interface for managing annotations *en masse* draws from other systems that utilize large-scale filtering and sorting of information (e.g., shopping websites). The Catseye pane now includes a top bar comprised of operations a user can perform to modify their set of visible annotations through searching (Figure 5.5-1), sorting, and filtering (Figure 5.5-2). Some specific filters, such as "Unanchored Only" and "Pinned Only", were designed such that users can find annotations that they may want to follow up on, such as for reanchoring an annotation. Others were designed to remove annotations that may not currently be relevant, such as "Show Resolved", in that the default is to not show resolved annotations, which are expected to represent finished tasks or answered questions. When finding annotations that may be useful for a *later* developer, however, finding such resolved annotations may be useful to isolate answered questions that could be useful, when used in conjunction with the *type* filter to select only question-type annotations. In this way, we sought to improve upon Adamite's typing, filtering, and sorting functions through combining multiple filters and sorts at once to isolate potentially useful annotations.

Once a user has created a filter or search to show annotations they want to do something with, the user can perform *batch operations*. The currently-supported

batch operations (shown at Figure 5.5-3) include merging, pinning, sharing, resolving, and deleting annotations. These batch processes will operate on any currently-selected annotations (Figure 5.5-4). Most of the functions are taken from similar information management systems, such as E-mail inboxes, with the rationale that such functions would similarly make sense in moving an annotation to the next stage of processing, whether that be when the annotation is no longer useful (batch resolution, un-pinning, and deletion), still useful (batch pinning), or ready to be used by other teammates (batch sharing). However, we found a need to also *transform* annotations prior to their usage by other teammates, leading to the development of our novel merging interface.

5.5.2 Merging Annotations

Merging annotations is the act of taking two or more annotations and combining them to form a single new annotation, while deleting the original annotations (see Figure 5.6). Considering a large part of curating content to be usable by later users is synthesizing information into a more connected and comprehensible form [148], we sought to provide an interface for that process that not only supports this synthesis but also removes the "leftover" content after that work is done, thus supporting two aspects of the curation process in one interface.

The user interface for merging annotations allows for a high level of specificity in terms of allowing the user to decide what parts of each annotation they want to keep and what they want to remove. Each core text-based component of an annotation (i.e., anchor, annotation content, replies) can be imported into the merged annotation. When text from an annotation body or reply is added to the annotation content (Figure 5.6-5), the content is automatically marked with additional meta-information including who authored the information and the time at which it was originally authored such that the original context is available. We chose to support adding reply content to the main annotation body considering many of our most useful annotations, both in this study and in the Adamite study, are question-answer pairs in which the answer was typically in a reply that may be missed, thus adding that relevant and important information to the main annotation content could help a later user's sensemaking. The user can also choose to expand upon, pare down, or otherwise transform the text, e.g., to be more formal since the information is imported directly into an editable text field.

We also support batch operations within the merge interface itself. Users can choose to add all annotations or add all replies (Figure 5.6-1) to their resulting annotation. Given the level of complexity of this UI, one can imagine a developer being reticent to use it since the cost of authoring new annotations is relatively low, thus creating a completely new annotation that includes content from other annotations may be easier than using the merge functionality. However, we expect that in the case of more complex merge cases, such as merging many annotations or merging annotations with multiple anchors across different files, these batch operations in

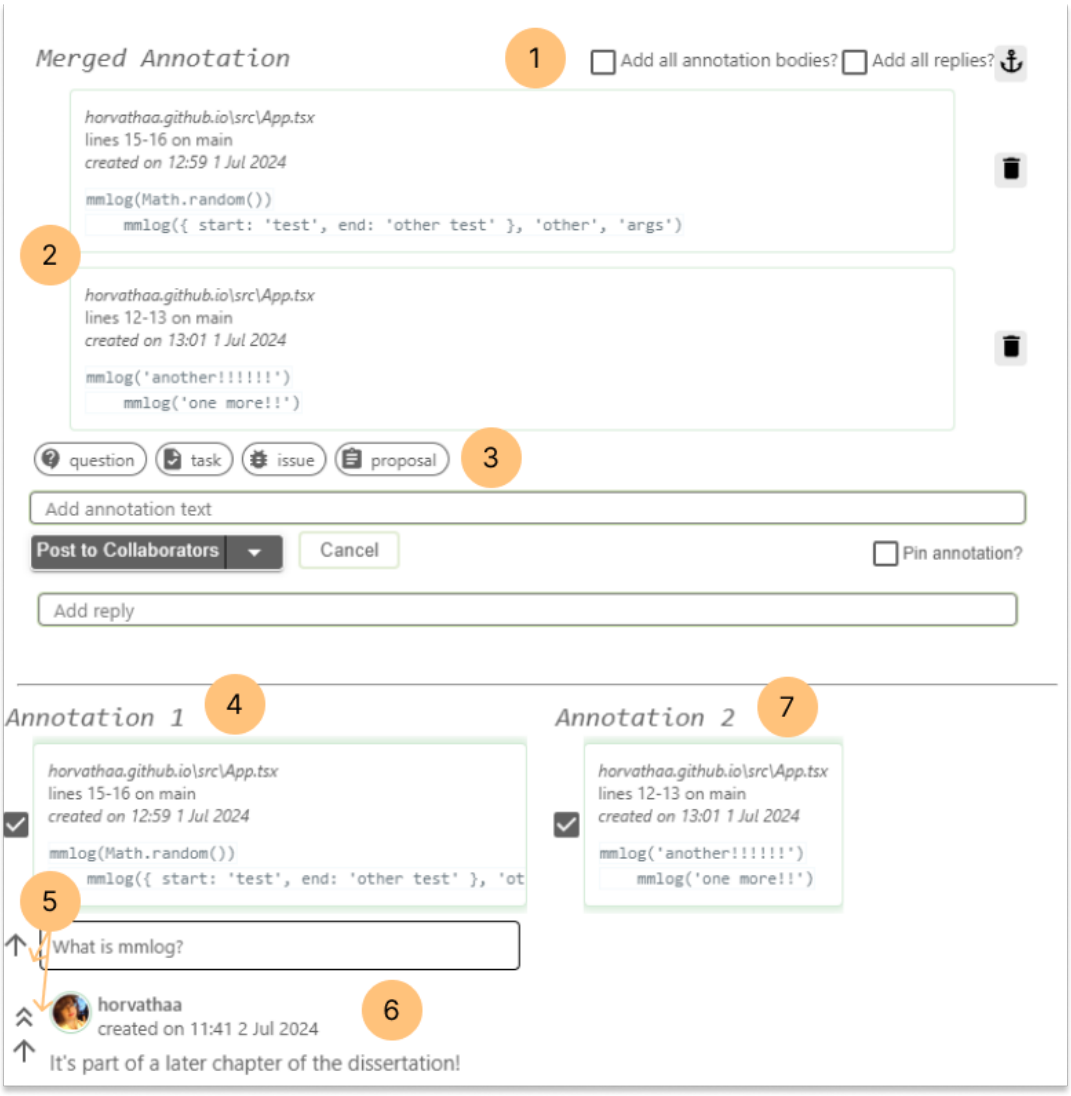

FIGURE 5.6: The UI for merging annotations. (1) are batch operations for creating the merged annotation through importing all of the content from the original annotations. (2) is a preview of the annotation anchors, with the option to remove them (trash can icon to their right). (3) is the regular annotation authoring UI, such that users can add types and text that may not be in the original annotations. (4) is the first annotation that is being used to create the merged annotation – the checkmark shows the anchor will be included. (5) points at the icons for importing annotation content and replies into the resulting annotation – annotation replies can be added to the merged annotation's main body text by clicking the double chevron icon, while the single upwards arrow will bring the reply's content up as a reply. (6) is the reply content for this annotation. (7) is the other annotation to be included in the merged annotation – since it is a highlight annotation, it has no annotation text or replies.

conjunction with the high-level of specificity afforded by the interface would be a better user experience than attempting to make the annotation from scratch.

5.6 Discussion and Future Work

In designing for annotation management, the user interface had to support potentially-competing goals – support complex activities while still feeling lightweight. In my personal usage of Catseye and in our user studies, developers wanted to be able

to jot down an otherwise throw-away thought about their implementation, bug, or something else they discovered in order to offload some of the intense cognitive load they were under. However, that in-the-moment need subsequently leads to more work later through either culling or transforming this information. We do not want to deter developers from generating information due to anxiety about later work, given that this information can be useful. This tension between supporting in-situ lightweight information generation and the subsequent work that lightweight information may incur inspired our exploration of this design space.

A particularly interesting aspect of this design space is the explicit connection between information scrap and document (i.e., annotation and anchor). Prior work in lightweight note-taking and PIM highlighted the importance of information location [21] and how that spatial component of, e.g., placing a sticky note with a grocery list on the fridge can be particularly powerful in reminding and recalling information. This connection between note and artifact has been less-explored in traditional digital spaces due to the 2-dimensional nature of a desktop environment leading to less "space" to work with and a lack of support for general-purpose user-facing customization of digital spaces. With our re-anchoring work, we further explored how this connection between information and source can be complicated in a dynamic space, such as a text editor, where information is constantly changing. This leads to questions of how to best retain the benefits of spatially organizing information when the space is evolving over time – returning to our scenario shown in Figure 5.1, in the case that the code is changed, our reanchoring algorithm would correctly find the most reasonable new anchor given content and location (Figure 5.2). However, it is still unknown whether the annotation should persist at all – the user asks "where does this value come from?" about the class property `this._copyVscodeMetadata` and, after the `git pull`, the class property is replaced with a method call (`this.getCopyMetaData()`). Perhaps the addition of a function call answered the user's question as the call implies the "value" the user was referring to is returned from the function. However, an alternative scenario is that the initial question is expressing confusion around the whole `vscodeMetadata` property, in which case the question may stand. Given annotations typically terse nature and the level of cognitive overload a user is typically experiencing when choosing to write such a note, it is unreasonable to expect more exposition that would make classifying these annotations possible. Some very recent work has explored using LLMs to automatically and intelligently re-anchor annotations with some success, but the approach is expensive and does not fully address the disconnect between ambiguous annotation content and code [166]. In this prototype, we chose to keep the human in the loop given this issue of text-versus-code ambiguity, but our next work, Sodalite, explores additional mechanisms for highlighting text and code inconsistencies with the goal of leveraging the anchoring relationship for document maintenance.

Sodalite additionally builds upon the idea proposed with the batch processing

work, namely that annotations can serve as "building blocks" for larger, more curated documents. Merging annotations is one way of taking intermediary information fragments and transforming them into a more shareable and appropriate form. Sodalite, similarly, allows the user to take locations within the source code and write about them using its specialized mark-up system, but no longer "imports" annotations as a starting point. Sodalite was primarily focused on long-form document maintenance, so leveraging and curating annotations was less relevant. Nonetheless, given how LLMs have evolved since this work, generating documentation through using annotations and their associated code may be a viable alternative to writing documentation from scratch (Sodalite) or merging and editing annotation content (Section 5.5.2).

The connection between document and information is particularly powerful in immediately placing information in context and, through that context, natively supporting the evolution of information over time. By abstracting this information onto its own plane, separate yet connected to the source code, annotations can exist without interfering with the "business logic" of the source code or development environment. This is in contrast to other systems which change the overall structure of the IDE [29] or interfere with the source code [244] in order to support spatial orientation and connectivity between meta-information and code. The invisible binding between source code and information that the re-anchoring work and, more broadly, this thesis explores highlights how developer's rich and textured inner world of tacit knowledge about code can be translated into meta-information and bound to the code itself, given proper tooling.

Chapter 6

Sodalite: Meta-Information to Support Documentation Management

---

This chapter is adapted from my paper:

> [103] Amber Horvath, Andrew Macvean, and Brad A. Myers. 2023. "Support for Long-Form Documentation Authoring and Maintenance". In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Washington, D.C., October 2-6, 2023. IEEE.

6.1 Overview

The connection between code and text written *about* code has long been studied and explored. From literate programming [48, 122, 222], which has had a recent resurgence with the rise of Jupyter notebooks [118], to various forms of documentation [1, 45], there exists a tension between making code comprehensible through expressing the design intent in a more naturalistic and human-readable form, while allowing for the control that traditional programming languages provide. Our approach with Sodalite[1], sits at the intersection between these two approaches by allowing the source code to be *anchored* to the corresponding written text that seeks to e.g., describe how it functions, or how it came to be.

Once the text is anchored, this relationship can be utilized in new ways. While Catseye and Adamite automatically attempted to update and repair any links between annotation and source text or code that appeared broken, this process sometimes failed and the annotation would be thrown away. This often occurred when the original source document changed in such a way that there was no longer an appropriate anchor location for the annotation, such as during a Git pull or if the

[1]Sodalite is a royal blue mineral and stands for **S**tories for **O**n-boarding as **D**ocumentation **A**uthoring, **L**everaging **I**DEs for **T**ext **E**nahancements.

API documentation changed between releases. While the fact that the annotation is no longer anchored, thus, may no longer be appropriate for the author or a reader to reference, the fact that it became un-anchored itself may be useful. For example, if I have annotated an API method's parameters with the specifics of my particular implementation, it would be useful to know that, in the new release of the API, these parameters no longer exist and my implementation will need to be changed. The challenge of keeping code and documentation in sync with one another has been a long-standing research problem [4, 210, 246].

This chapter explores utilizing the connection between code and text to overcome longstanding barriers in documentation authoring and maintenance. By leveraging code meta-information including the structure, existence, and location of code, Sodalite is able to automatically determine the likelihood of the documentation being out-of-date given how well the system performs at re-attaching each code-text pair to the code in the IDE. Sodalite also uses this meta-information to help the initial documentation author in writing their "code story" through suggesting code patches to document, given the author's already-documented code, and providing templates for bootstrapping the authoring process. We expect this documentation authoring and maintaining process to be particularly useful for developer teams or open source projects, where the documentation and corresponding source code can "live together", with team-oriented documentation suffering from many of the same barriers that end-user facing documentation does [110, 208], including out-of-date information [208], while typically lacking a dedicated team or process for updating the documentation [3, 53].

6.2 Background and Related Work

Prior work about writing software documentation have ranged from understanding what information is in documentation [94, 155], what the problems are with that information [3, 4, 35, 65, 140, 160, 178, 210, 212, 246], how software documentation is authored [53, 219, 233], and automating documentation processes to offset authoring costs and standardize the information present in documentation [2, 88]. Notably, the majority of this documentation work is in the context of software documentation for end users of a software library, e.g. API documentation [172]. Most relevant to our work are studies about documentation created for other developers working on the *same code base*, such as internal documentation about the code base [208, 219] or a code tour for an open source repository [241], where the primary goal is to help other developers understand and contribute to the code base. Our work expands upon this work by contributing a system designed specifically to help create this type of documentation, with a focus on combating some of the known issues in authoring and maintaining this documentation including out-of-date information.

Once some software documentation is made, researchers have studied how developers maintain those documents and use that information. In studies of usage,

researchers have identified many problems of documentation that lead to the documentation being less trustworthy [150], including questions about how up-to-date the information is [4, 140, 246] and completeness [210, 212]. A survey of developers at one company reported that they rarely updated documentation and that they correctly assumed that documentation content is out-of-date [140]. One reason for this lack of maintenance is that finding the appropriate places to update given a change can be challenging, with developers reporting that they would value a tool that helps identify those locations [73]. Our system attempts to reduce some of these costs by having maintenance be a core design consideration by automatically locating and highlighting the out-of-date portions of the document given which code was changed.

6.3 Sodalite

In order to use Sodalite, a user begins by opening the Sodalite webview [165] which appears and functions like any other file in the editor (i.e., it can be dragged, resized, closed, etc.). The user can then view the stories that have been associated with the user's currently-open GitHub repository or choose to author a new story (see Figure 6.1). When authoring a new story, the user will be presented with a rich text editor, hereafter referred to as the "story editor" (in contrast to the "code editor", which refers to the Visual Studio Code IDE), and users can utilize code story *templates* and *code links*. For already-authored code stories, the system performs its maintenance algorithm to determine what code links are potentially invalid.

6.3.1 Templates

We included templates for stories to help guide authors about what should be documented (a common problem with documentation authoring is not knowing what is important to document [53]) and to provide built-in mechanisms for allowing developers to find the right code and code-related information to include in the documentation.

We began by identifying various types of developer long-form writings that are not adequately supported by current tooling and would benefit from leveraging the context of the source code. We compiled this list through a combination of reviewing literature and informally consulting with software engineers. For each of these types, we developed a "template" which contains a list of headings, subheadings, and guided instructions with prioritized code links (see Section 6.3.2), that would most likely be included. This list includes:

- **Overview documents**, which serve to introduce a newcomer to a code base [208, 241]. This template prioritizes higher-level information, including references to functions and classes, with the assumption that developers most likely do not want to cover lower-level specifics in an overview. "Overview" also

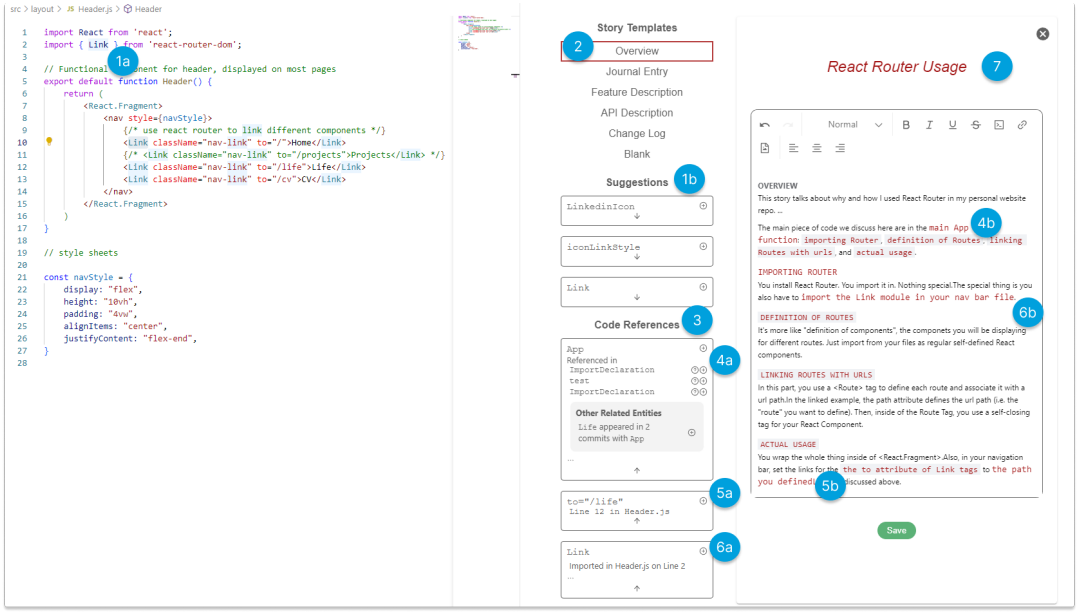

FIGURE 6.1: The editor for authoring a story using Sodalite – this is a simplified and anonymous recreation of P2's story for demonstrative purposes. (1a) and (1b) show the relationship between the code editor and the story editor – in this case, the user has clicked the code `Link`, prompting the suggestions pane to show identifiers related to `Link`. (2) is the template list with the selected "Overview" template highlighted. The "Code References" at (3) are represented in a list (4a, 5a, and 6a) and their locations are shown in the story text at (4b), (5b), and (6b), respectively. The title of the story is (7).

prioritizes identifiers with certain keywords in their names that are commonly associated with control flow in a web development project, such as "listen" or "handle", with the rationale that the logical flow of a project may be discussed in an overview.

- **Journal entries**, which serve as logs about what a software engineer completed in a workday and are sometimes required by management [154].
- **Feature descriptions**, in which a developer discusses what a particular part of a code base does and is has been claimed to be one of the most important types of documentation for developers [140].
- **API descriptions**, where a developer catalogs what they have learned about an API to benefit later developers [101].
- **Change logs**, in which a developer enumerates parts of the code base that have changed between releases of a software project [154], but, due to authoring costs, can be incomplete [38].

Note that we are not claiming that this is an exhaustive list and our system allows developers to define their own template if the built-in ones are not adequate by using a predefined JavaScript object notation (JSON) structure that Sodalite reads. The system also defaults to a "Blank" template, such that the user can define their own structure without extending Sodalite.

6.3.2 Code Links and Suggestions

A key feature of Sodalite are *code links*, where the story references back to code. Links can be made manually or using the "Suggestions" pane (see Figure 6.1-1b), which lists code links that the author may include in their code story. Code links come in three varieties:

- **Identifier definitions**, which link to where a specific code entity like a method or variable is first defined. A user can create an identifier definition link by selecting the "plus" button in the suggestions pane at the top right corner of the code link box. Identifier definitions also include additional information about the identifier, including places in which it is referenced, and other classes or functions it references (see Figure 6.1-4a).
- **Identifier references**, which link to a specific instance in which a particular identifier is referenced or used. An author can make an identifier reference by selecting a reference that is listed in an identifier definition's list of referenced locations. Figure 6.1-5a shows a reference.
- **Code ranges** (see Figure 6.1-6a), which can be any arbitrary range of code that the user has selected in the Visual Studio Code editor. Selected code in the code editor will always appear at the top of the user's "Suggestions" list, given that selecting code is a strong signal that the user wants to include that code in their story. Code ranges are fundamentally the same as the "code anchors" that Catseye supports.

Code links can be added to a code story in two different ways. If the user has no text selected in the story editor, the code link will either insert the name of the identifier (identifier definitions and references) or the selected code (code range) at the location of the user's cursor. If the user has selected some text in their code story, that text will be linked to their code (see Figure 6.1-4b, 5b and 6b). Either way, clicking on a code link within the story editor will navigate to wherever that particular code link is located in the code editor. Once a code link has been added to a story, it will appear in a "Code References" list (Figure 6.1-3), such that the link may be used elsewhere in the story.

Code links can also contain additional meta-data Sodalite was able to determine about that part of the code. This includes, for the identifiers, where they are defined, and referenced in different parts of the code. Once a code link has been included in the code story the "Suggestions" pane will include other identifiers that were commonly edited at the same time as that particular identifier. We identify these "co-edits" by parsing the Git commit history for that particular identifier and count when other identifiers appear in the same commit. In this way, we attempt to identify parts of code that are related but not in a way AST parsing would find.

Given these different types of code links, the "Suggestions" pane uses different sources of information to populate its list. Sodalite examines both what the user

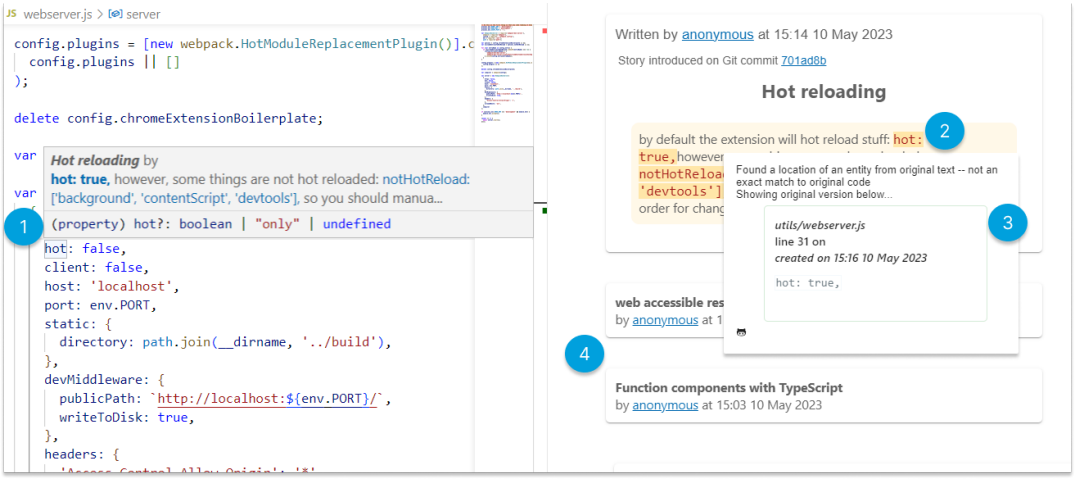

FIGURE 6.2: How Sodalite appears after a story has been authored. (1) shows the hover text when a code link is interacted with in the code editor. (2) is a code link that has been marked as "needs review" given that the original code (shown in (3)) and the code in the editor (at (1)) are different. (4) are two other collapsed code stories.

is doing in the story editor and what they are doing in the code editor. The information the algorithm leverages from the story includes what references are already included in the story and what the user has most recently typed. Identifiers related to references already within the code story will be prioritized, as will identifiers that match some part of the most recently-typed text. The system then complements that information with what it knows about the user's current location within the code, including if the user is currently selecting an identifier, and, if so, what identifiers are related to the selected identifier, along with other identifiers referenced in the file and, if applicable, in the user's currently selected scope. Some templates like the "Change Log" prioritize certain types of code links (e.g., code links that have been heavily edited), in which case the system favors those identifiers in the Suggestions list. Identifiers that appear in multiple information sources are pushed to the top of the list. To prevent the user from being overwhelmed with suggestions, we limit the amount of suggestions to the top 10.

When a story is saved, Sodalite generates a JSON file in a system-generated folder, which can be committed to the user's Git repository and used by other programmers when they read or edit the source code. This JSON file is also used by the system when determining what code links are out-of-date.

6.3.3 Support for Reading

Sodalite has some features designed specifically to help readers of code stories better understand and utilize the documentation. A core feature of Sodalite for readers of code stories is the fact that it is situated within the context of the code. Developers have previously stated they value source code and code comments more than other types of documentation [229] – we hypothesize that bringing the type of information typically in external documentation into the source code will allow for "the

best of both worlds" by staying within the developer's working context while also supporting longer-form text.

Another feature of Sodalite that we expect to help users of code stories are the bi-directional nature of the code links. When some code has been linked in a story, the code will be highlighted within the IDE, such that users of the code can discover pertinent documentation that is relevant to that code (see Figure 6.2-1). The highlighted code brings up a hover text that shows a preview of the code story, including the surrounding text from the story, the part of the story that is linked to the code in bold, and the name and author of the story. If the surrounding story text contains code links to other parts of the code, those links will navigate the user to the code link's location. If the user clicks on the link in the hover text, the Sodalite pane will open and scroll to the correct part of the story. This does not interfere with the regular links in Visual Studio Code's hover text, which still work like normal.

Users of code stories are also presented with additional context about the originally authored story, to further help them validate what information is still relevant. The story shows who authored the story, the date and what Git version the project was on when the story was last authored or modified, and each code link in the story displays additional metadata about what version of the code it was on when authored, along with its original code content and context. This metadata complements the information the out-of-date system provides when the story is validated.

6.3.4 Support for Maintenance

Sodalite leverages being in the code editor and having access to Git versioning information to be able to mark parts of stories as "valid" (in which all code links are valid), potentially in"need of review" (in which the system found a potential match for the code link, but it is not positive – see Figure 6.2) or definitely "invalid" based on how well the system is able to match the code references in the story to the code currently in the user's project in Visual Studio Code. There are situations in which the documentation may go out-of-date that our system would not capture, but, in a study of documentation problems, [4] found that most cases in which the documentation went out-of-date was due to the code changing, so we focus on that case with Sodalite.

On launch, Sodalite parses every code story file in the user's current project and builds an internal AST representation of the code in the project to compare the code links against. We use the different types of code links to inform how to re-attach a particular link and whether that attachment is valid or not.

For code anchors, given their flexible code structures (e.g., anything from a string to a full multi-line expression), we begin by evaluating the most optimistic condition, in which nothing has changed, through checking whether the position we have saved contains the same code as the code link. If not, we then look for whether the code from the code link exists anywhere within the document. If that does not

work, we use purely text-based matching mechanisms to discern candidate matching points, since that was the most successful method used in [206]. We use AST information to restrict our search spaces, using a recursive function from most within the last-known scope location.

In the case of multiple matches or no matches, we run our re-anchoring algorithm, partially adapted from [206]. This algorithm uses the last-known code content, the original location within the file, and the original surrounding code (all of which are saved with each code link) to find the most likely candidate code reference. We then weigh the probability that the location is correct through a combination of calculating the edit-distance between the two versions, the edit-distance between the surrounding lines of code, and the difference between the candidate line(s) of code and the original location, with a closer location being weighted higher.

If there is a low score or no matches, we mark the anchor as invalid and the corresponding text in the story as in need of review. To assist the author in finding either a new location to link the story text to in the code or to discard the link, we present additional metadata about what the code looked like and where it was last located (see Figure 6.2-3, where it says "Showing original version below...").

For definition references, the system searches the internal representation to see whether the identifier is defined at its last known location. If the definition is not there, the search will expand outwards to see if the identifier is defined in a different file and, if so, attaches to that location, but marks the link as potentially invalid, considering it may be a different identifier that just happens to have the same name. If it is a local definition (e.g., a property named `bar` that is part of class `Foo`) the system will check that both the class `Foo` and the property `bar` exist. If the system cannot find a definition for the particular referenced code entity, the system uses the text-and-location based re-anchoring algorithm. If the algorithm does not return a result of a sufficiently highly-weighted likelihood, we mark the reference as invalid, and the surrounding text within the code story as in need of review by the author.

A similar approach is used for checking identifier references in the code. We begin by seeing whether there are one or more references in the code in the last-known scope in which the reference was used. If there are multiple candidate matches, we find the reference with the most similar AST path to the saved code anchor information. If there are no candidate matches, we once again fall back on the text-matching algorithm and, if the result is inadequate, mark the text within the story as potentially invalid and in need of review.

By attempting to automate the process of re-matching the story content to the corresponding code and suggesting potential edits to make, we attempt to lower the workload for the author in maintaining the documentation. "Valid" code links have no highlights. "In need-of-review" code links (shown in Figure 6.2-2) are highlighted in yellow as a warning to readers. "Invalid" code links are highlighted in red to draw attention to this part of the story.

6.4 Evaluation of Sodalite

6.4.1 Study Design

In order to assess the usability and usefulness of Sodalite, we ran a small user study to test both the authoring and maintaining aspects of Sodalite. Participants were instructed to use Sodalite to document some code of their choosing. We then observed them as they chose to document whatever information they saw fit.

To evaluate the maintenance features, participants took their documented code and reverted to an *older* version of their code. We considered requiring participants to document an old version of their code, but decided against this since we did not want the study to become a test of how well the participant's remembered their old code and developers are not typically documenting their old code. Therefore, they documented the current version, and we pretended that an old version was a newer version.

Participants were instructed to look through their project's GitHub history to find a version that was many commits behind the version they documented and involved edits to any files that they ended up documenting during the authoring portion of the study. Upon finding an appropriate commit, participants checked out that version of the code and re-ran Sodalite, which triggers the maintenance system to try and find appropriate re-anchoring points. The first author and the participant stepped through each code link within the stories to determine whether the system chose the "correct" action, meaning the participant agreed with the system's determination of whether the link was "valid", in "need of review" or "invalid".

We recruited 4 participants (3 men and 1 woman), hereafter referred to as P1 through P4, using study recruitment channels at our institution and advertisements on Twitter. All participants were required to have some amount of experience using JavaScript or TypeScript, to have a project written in one of those programming languages that they were willing to document, and to regularly use Visual Studio Code. All sessions were conducted over video conferencing software and the sessions were recorded. Participants had, on average, 12.25 years of professional programming experience and considered themselves proficient in JavaScript and TypeScript.

For analyzing the authoring experience, we captured how many stories the participants authored, how long each story was, what type of templates the participants used, how many code links they chose to include across how many files, and what types of code links they were. We additionally reviewed the recorded videos to count whenever the participant navigated through the document using the code links, and counted whenever a participant copied and pasted or deleted a code link. We additionally noted any usability issues participants discovered with the tool.

To assess how well Sodalite did at *maintaining* the code stories, we computed the false positives, false negatives, true positives and true negatives for each code link given the participant's evaluation of whether or not Sodalite correctly re-attached the code link. We additionally logged what the offset was from the code location

during the authoring session to the maintenance session, along with the total diff between versions, and what type of re-attachment strategy the system used (e.g., text-based, AST based, etc.).

6.4.2 Study Results

All participants were able to successfully use Sodalite to author at least one code story that utilized Sodalite's features, and agreed with Sodalite's re-attachment decisions, on average, 86.5% of the time. We further discuss participants' experience using Sodalite's authoring and maintenance features.

Authoring

The 4 participants authored, in total, 15 stories, with 2 participants each writing one story, and the other 2 participants authoring 6 and 7 stories, respectively. On average, participant's wrote 238 words (min = 136, max = 312, std. dev. = 75.4) during the 45 minute authoring time. The participants in our study chose code repositories of varying purposes, sizes and complexity to document. Participants, on average, chose to document 3.25 files. Our participants chose to document a Google Chrome extension, a Visual Studio Code extension, a browser game, and a personal website.

Across all stories, participants created 52 code links, with each participant, on average, creating 13 (min = 11, max = 18, std. dev. = 3.36). 18 of the code links were pointers to definitions of identifiers within their code[2], 7 were identifier references, and the remaining 27 were "code ranges". In their stories, participants, on average created 12.75 code links.

The code links were represented in different ways within the code story. 29 of the 51 code links were attached to text in the code story that was just the name of the identifier or the expression the user had selected. Notably, 18 of those 29 code links were definitions of particular identifiers, where the story sought to explain some detail about that identifier. The remaining 22 code links were attached to text that was part of a sentence within their code story. This split in usages suggests that supporting both of these ways of referring to code helped support different ways in which participants wanted to talk about their code.

Participants sometimes used their code links to help them navigate through their code, using, on average, 41% of their code links at least once to navigate through their code (min = 0, max = .722). Participants would sometimes check, after including a code link, where that link went as a way of assessing the reader experience. P4, in particular, used the navigation features often, revisiting the majority of their code links to review the other places in which the code was referenced and used their links to navigate 13 different times across the session. This suggests that the code links may be useful for navigating through a code base, even when authoring documentation.

[2]P1 made exclusively this type of code link, comprising 11 of the 18 total definitions.

Maintaining

Sodalite made the correct decision of either re-attaching or not re-attaching 44 out of the 52 code links, a success rate of 86.5%. Of the incorrect links, 7 were because the system could not find an attachment point, despite there being one within the code, and 1 was because the system erroneously found a location that it thought was correct when it was not. The majority of cases were that Sodalite successfully re-attached a code link, at 36 instances. Eight code links had changed in such a way the system correctly marked them as needing review. The reversions averaged 100 lines of code lost (min = 53, max = 137, std. dev. = 38.92) and 49 lines of code added (min = 3, max = 95, std. dev = 51.97) per file.

Sixteen of the correctly re-attached links were definitions, in which the re-attachment strategy was to find a point within the file in which the named entity was defined. Given the relative broadness of this strategy and the specificity of the named functions or classes participants wanted to document, all but 2 definitions were able to find an appropriate point somewhere in the document. This strategy may not work as well if a method or class is renamed, but that did not happen to occur in any of the participants' versions. The reference re-attachment strategy also worked well, with 8 of the 10 references being successfully re-attached in an appropriate location.

Perhaps more interestingly, the 8 cases that the system correctly marked for review occurred in different situations, including when some code had been commented out, the underlying semantics of the code changed significantly enough such that the text in the story was no longer valid, and when the code links' corresponding definitions did not exist. One such instance is shown in Figure 6.2 in which the original code link and surrounding text states that a value, `hot`, should be set to true. However, the code in the version the user has opened has `hot` set to false. Two of the cases occurred because the definitions for the linked methods did not exist in that version of the code.

The single false negative case, where Sodalite missed a change, also occurred using the text-based matching algorithm. The anchor was originally placed on an `else` keyword, but, when the prior version was pulled in, the system placed the code anchor on a completely different instance of `else`. This other location was closer to where the original link was and, since `else` is such a common keyword, the system found multiple reasonable locations but selected the incorrect one. For identifier references, Sodalite flags situations like this, but the system does not currently support that for keywords. Future versions of Sodalite would benefit from introducing more heuristics about the context in which the code appears, including any logical dependencies, such as an `else` statement's corresponding `if` statement.

6.5 Discussion and Future Work

Our evaluation lends evidence to the claim that in-editor authoring and maintaining of documentation can be supported with Sodalite. Participants succeeded in creating code stories and, when faced with changed code, the system was very successful in identifying problematic references, and relatively successful in choosing the appropriate action to take.

A current limitation of Sodalite is that all of the mechanisms for determining whether or not the story is out-of-date are contingent upon the story including code links. The expectation is that, given Sodalite being located within the IDE, developers will naturally utilize that context and reference their code within their code stories. Indeed, all participants stated in the post-task survey that they valued the ability to link their text to their code. Nonetheless, future versions of Sodalite may benefit from additional mechanisms for assessing the validity of the text in comparison to the code, for example, through leveraging crowd-sourcing mechanisms for quality control [6, 53].

Another benefit of having even better mechanisms for identifying out-of-date code stories is to notify documentation writers and/or developers when their story has gone out-of-date. It is a known problem that updating documentation is a task that developers typically put off – while Sodalite attempts to make updating easier through marking places for review, the system does not require that the author take any action. One can imagine an additional feature to Sodalite that allows stories to be marked as "very important", in which case developers could immediately be notified if they make a breaking change, while lower priority stories can be addressed on the developer's own time. In the proposed work which involves integrating Sodalite with the Meta-Manager, Meta-Manager's constant listening for change events may be leveraged for real-time detection of code links going out-of-date.

Chapter 7

Meta-Manager: Meta-Information for Question-Answering

This chapter is adapted from my paper:

[102] Amber Horvath, Andrew Macvean, and Brad A. Myers. "Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code". In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*, May 11–16, 2024, Honolulu, HI, USA. Association for Computing Machinery, 2024.

7.1 Overview

When developing and understanding code, as previously discussed, developers are managing many types of information that come in different forms. The previous chapters explore how this information can be externalized and made more useful through tooling approaches that leverage the context of the source code. However, this externalization requires additional cognitive and physical effort by the initial author [15, 34, 96, 121, 216], which can be a deterrent from using these types of tools. Further, these tools also only capture what is expressly written by the developer, which, while useful for capturing information that may exist only within the head of the developer, can miss other interesting or useful contextual meta-information that was created during the sensemaking [148, 149]. Nonetheless, information related to the history and implementation of code, while often not expressed by the developer [144], is often useful in answering historically "hard-to-answer" questions about code, specifically those related to rationale [129, 135].

In this chapter, we explore capturing code editing and history information traces at scale through an automatic, event-driven tooling approach. We isolated editing events that we expected would help developers in answering some of these questions, such as copy-paste events to help with understanding *where* some code originated from and web search events and additional meta-information from web pages

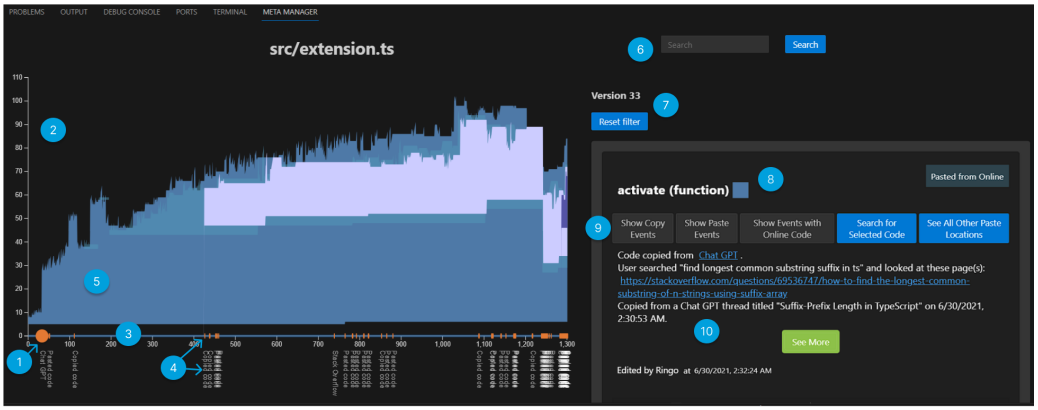

FIGURE 7.1: Meta-Manager as it appears within Visual Studio Code: the pane appears in the bottom area of the editor, with the left area displaying a visualization of the history of the code file over time, while the right area displays information about a particular code version.

including Stack Overflow and ChatGPT for answering *why* some code is written a certain way.

We instantiated this event-driven code history and provenance model in our tool, Meta-Manager. Meta-Manager translates this information into key events along a code history timeline visualization that allows developers to explore how, when, and where some code came to be. We focus on supporting *later developers in answering questions* related to code design rationale, along with code history, provenance, and relationships, given that these questions are significant blockers for developers maintaining code [129, 135, 156]. We specifically designed Meta-Manager with the goal of not only supporting developers in answering these questions but also with respect to dealing with challenges of *scale* (given the sheer amount of edits made to code that may not be of interest but are nonetheless logged [7, 225]), *navigation* (to support developers in finding the edits that are of interest), and *unwritten design rationale* (given developers' reticence to stop their work to document particular decisions [148, 156], despite their generated information trail's potential to be useful [121, 150]). In our evaluation of Meta-Manager, we found participants were able to successfully use it to answer otherwise unanswerable questions about an unfamilliar code base's history.

7.2 Overview of Meta-Manager

In order to capture code provenance, history, and rationale information at scale in a manner that is navigable and informative, we developed Meta-Manager. We begin our discussion by discussing what questions we believe Meta-Manager is well-poised to answer, then show a user scenario on how that history may be generated and used by a later developer to answer a question with Meta-Manager. We then discuss how each feature in Meta-Manager instantiates our design goals and addresses

significant questions developers have about code.

7.2.1 Developer Information Needs

In designing and creating Meta-Manager, we began by reviewing related literature on information needs of developers when working with unfamiliar code [51, 60, 75, 110, 124, 135, 156, 176, 207, 208, 223]. As discussed in Chapter 2, this can happen in many contexts, such as when adopting a code base after a coworker leaves [176, 207, 208] or when maintaining a large code base [75, 124, 135, 156, 223]. In reviewing the literature, we were particularly interested in questions about the *rationale* behind code's design, given the lack of support for that information need [156], despite its ubiquity and importance [135, 156]. Through our literature review, we identified the following questions as related to code rationale and provenance, and as potentially answerable through directly supporting developer's sensemaking of code history with tooling:

- ***History*: How has this code changed over time?** [60, 135, 156, 223] Developers often try and understand the evolution of some code in service of answering a question that is pertinent to their current task. For example, this may help while investigating when a bug was introduced [226], finding when some code was last used in service of understanding how a feature changed over time [262], getting "up-to-speed" on a new code base [110], or finding a snippet of code that was edited repeatedly to understand where the original developer had issues [135, 144, 226]. Isolating when these particular changes happened can be impossible in the case that the intermittent version is not logged in a version control system (which is often the case in situations where a developer is trying out multiple solutions), or very difficult to find even if there [116].

- ***Rationale*: Why was this code written this way?** [110, 124, 135, 156, 208, 223] A commonly-reported activity among developers when understanding unfamiliar code is reasoning about why it is written the way that it is. This information is typically only known by the original author during the time at which the code was written and, if not written down (which is the majority of cases [148, 156]), is lost. On the off chance it is recorded, it is most likely preserved in the form of a random Git commit message or code review comment [191], which are often too difficult to forage through [230]. Developers have stated that attempting to answer these questions are "exhausting" given the lack of tooling support and reticence to ask co-workers [156], yet they must be answered in order to understand design constraints and requirements which will inform later implementation decisions.

- ***Relationships*: What code is related to this code?** [51, 75, 135, 223] Oftentimes, when contributing a change to a code base, developers must reason about how their new code is related to many other parts of the code beyond simply what

could be found in a call graph. Other relationships that developers reason about are what parts of the code are commonly edited together (the "working set" [27, 45]), and, if introducing a change or refactoring some code, what other parts of the code must be updated. Developers also sometimes wonder what solutions a previous developer already tried when introducing a change, another otherwise untraceable relationship given that such prior solutions are usually commented-out or deleted [135].

- ***Provenance*: Where did this code come from?** [75, 223] In 2021, Stack Overflow reported that one out of every four users who visit a Stack Overflow question copy some code within five minutes of hitting the page, which totals over 40 million copies across over 7 million posts in the span of only two weeks [198]. Given this ubiquity of online code and developers reliance upon it, researchers have investigated the trustworthiness of code that is sourced from online resources [12], ability to be adapted to a developer's own working context [273], and correctness of the code in terms of API usage, syntax, and so on [236]. With the rise in LLMs for code generation, research is beginning to focus on the quality of AI-generated code as well [55, 68, 120, 145, 147, 239, 263, 268]. Typically, it is not easy to see what code came from AI or from an online source, versus what was written by developers themselves. While developers occasionally add code comments that cite where some pasted code came from, this does not happen very often [11] and, when it does happen, the links have a tendency to break over time and recreating the context in which that code was initially added and determining whether it is still valid is laborious [90].

It is also worth noting that all of these questions are phrased about "this code", which is the language used in the original research. This phrasing suggests that developers are typically discussing these questions at the snippet or block level [98], as opposed to the file level, which is what typical version control systems operate at. Our system differs by tracking code at the block level across files and supports drilling down to a specific line or lines of code interactively.

7.2.2 Scenario

Ringo, a software engineer, is working on implementing a calendar widget into his team's scheduling software. Ringo is using an off-the-shelf React component that provides most of the calendar widget's functionality and visuals – yet, as he is implementing some of the date verification, he notices that the returned time is incorrect. He begins by *searching Google for how the date verification API works*, *visits the documentation but does not find any useful code examples*, then *asks ChatGPT what is wrong with his usage of the API and how to get the API to verify the date correctly*. ChatGPT provides him with a code example, which Ringo *copy-pastes into the code base*. Upon re-running the code, he sees the snippet works and thinks nothing more of it. He, then, *pastes this code into the other parts of the project requiring date verification*.

Many months later, Jeremiah, a software engineer who has recently joined this project team, is working on one of his first pull requests. In doing so, he spends time familiarizing himself with the code base by reading through the code. While reading, Jeremiah notices an odd implementation choice – a particular function uses an earlier version of an API's method for checking the time of a calendar widget, despite the current version of the calendar API being used elsewhere. Jeremiah is not initially certain whether this confusing implementation decision is intentional or not, as there is no documentation on this line of code, and, given this uncertainty, he is reticent to change the code out of fear of some undocumented design criteria. Jeremiah wonders *"why is this code written this way?"* and launches Meta-Manager to investigate.

Jeremiah notices in the Meta-Manager pane that this particular file has many hundreds of edits and, through the visualization, notices that the particular block with the confusing code was introduced many edits ago. This suggests that Jeremiah's current teammates would most likely not know why this particular API method is used. Thus, Jeremiah begins using the Meta-Manager by selecting the line of code in question and *searches backwards in time* to see when this line of code was introduced. When the Meta-Manager timeline updates with places in which the line was edited, Jeremiah notices that the line was added with minimal subsequent edits and its first appearance corresponds to a paste from ChatGPT. This tells Jeremiah that *the code has not evolved much over time* suggesting that it was a solution that did not require much tweaking by the author. Jeremiah *inspects the code version by clicking on the "ChatGPT" paste event*. The ChatGPT code version has *additional meta-information including the original developer's Google search and visited web pages* which shows that they were looking at the API documentation. The thread shows that they asked ChatGPT for a code example that uses the API to verify a date and ChatGPT provided the code using the earlier API method. With this additional context provided by Meta-Manager's meta-information, Jeremiah now knows *where this odd code came from*, as the older API usage was provided by ChatGPT, and *why the code was written the way it is* – namely, to meet a specification that the newer version of the API does not provide. With this information, Jeremiah no longer needs to ask his teammates about the usage of the old API and feels comfortable leaving it as is – he adds a code comment to the line stating that this line should be updated if the calendar API updates with new date-checking functions.

Jeremiah, lastly, wants to see if there are any other parts of the code using the older version of the API, such that he can similarly mark those parts of the code for updating. In order to *find any code related to his current code*, he looks to see whether this code has been copied and pasted anywhere, and finds that the code was copied and pasted 4 times across history. When looking at those copy events, he navigates to the corresponding pastes and sees that 2 of the 4 pastes no longer exist. For the remaining pastes, he adds a code comment stating the lines should be updated.

7.2.3 Detailed Meta-Manager Design

We now discuss features (labelled with "F" below) of Meta-Manager in terms of its design goals ("D"), and how these features support answering the history and rationale questions about code we have identified in our prior literature review (Section 7.2.1).

[D1] Automatic Code History and Provenance Data

We developed Meta-Manager to support better navigation and sensemaking of code history through a scalable and visualized history view (see Figure 7.1). Meta-Manager supports automated history and provenance data through its organization of data and its history model ([F1]), along with extending its historical data capturing outside of the IDE ([F2]).

[F1] Data Organization. On system launch, Meta-Manager creates an index of the entire code project by traversing through each file and creating an abstract syntax tree (AST) representation of each TypeScript or JavaScript file and, if Meta-Manager has been used with the code project previously, searching the Meta-Manager database to find what code blocks in the current project correspond to the code blocks saved in the database history. In the case that the block does not exist in the database, Meta-Manager will begin tracking its history.

We chose to track code history at the block-level, as opposed to the file level, in order to better align with developers' mental models of code [97] and given that our supported questions are often asked at the block or snippet level. This approach also complements our design goals of combating scale, considering each code block is in charge of its own history, meaning code versions are only captured when a block has changed. By deconstructing the versioning space to each code block and allowing each code block to manage its own history, we can support more fine-grained answering of questions related to *history* and *provenance*.

In order for Meta-Manager to begin logging code versions, the user does not need to take any actions beyond installing the extensions. On each file save, Meta-Manager will log a new version and perform an audit of the file to see if there are any new blocks of code to track. To investigate the code history, the developer can navigate to the "Meta Manager" tab in the bottom area of the editor — doing so will render the edit history of the user's currently-open file. Whenever the user opens a file, the Meta-Manager will render that particular file's history. Each code block's history appears both within the visualization as a colored stream (Figure 7.1-5) and, given the location of the scrubber (Figure 7.1-1) along the timeline, a code box version (Figure 7.1-8) is shown that represents that particular code block at that point in time.

[F2] Development Traces Online and in-IDE. Meta-Manager tracks code-related

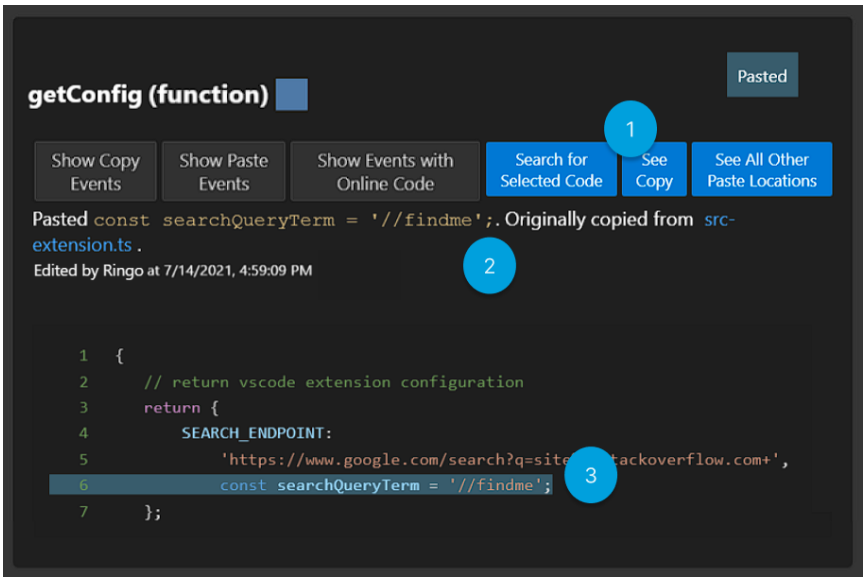

FIGURE 7.2: How the code box looks when expanded to show a code version – in this case, a "Paste" event version. (1) shows the buttons specific to a "Paste" code version, including the"See Copy" button which will navigate the user to the corresponding copy event on the timeline (if the copy happened in a different file, then the code box will update with a preview of how the code in the other file looked at the time of the copy, which can be clicked on to change to that file); (2) shows the text explaining what happened with this particular paste event — clicking in this area will open the editor tab showing what the code file looks like now; (3) shows the code for this version, along with a light blue highlight on the code that was pasted.

development events within the IDE and online. For certain events, additional meta-information will be shown on those particular code versions with additional affordances. For example, in Figure 7.2, this particular version of the method `getConfig` had a paste event, where the user pasted in the code on line 6. The version adds additional information such as where that copy came from (in this case, the file "src-extension.ts") and buttons for relevant actions, such as seeing the original copied code (Figure 7.2-1).

In cases where code was pasted from an online source, Meta-Manager will provide additional meta-information about the web page that the code was pasted from, and, if available, what the original user was attempting to do. Meta-Manager's supplementary browser extension is designed to work with some popular programming learning resources, including Stack Overflow, GitHub, and ChatGPT[1]. If the browser extension detects that the user is on one of these web pages, it will extract website-specific information (e.g., the name of a ChatGPT thread) and listen for copy events. If the Visual Studio Code extension detects a paste which matches the content of the browser extension's copy, this additional information will be transmitted to the Visual Studio Code extension to be associated with that paste. The hypothesis is that the query text can be a good signal of the developer's original intent for the code, which has been supported by prior work [121, 150] and our observations.

[1]We envision this list being substantially expanded to include other commonly-used resources where code is copied from, such as the official documentation for languages and APIs.

Similarly, if the user makes a programming-related Google search[2] prior to visiting these websites, their initial query and visited web pages will be included with the meta-information about the pasted code (Figure 7.1-10). Clicking the "See More" button will pull up a preview of the web page in the Meta-Manager pane of the editor, and highlight the part of the code on the web page from where it was copied.

Through automatically capturing this development context that would be too laborious to capture manually, we hypothesize that these pieces of information, when combined and contextualized to when the edit happened, can help developers reason about the *rationale* behind a change and the relevant *provenance*. These features work in conjunction with our data model, which allows each block to track this information. A problem with other methods for keeping track of provenance information, such as code comments that contain links to where some code came from, is that the information can go out-of-date, either in the case the link breaks or the code changes enough such that the code comment is no longer accurate [90]. By having this information versioned, we give the developer the tools to reason about this rationale across time.

[D2] Scalability.

Given the sheer amount of information we are tracking with Meta-Manager, Meta-Manager is designed to support managing large amounts of information. We do this in multiple ways – both *collapsing* information into a visual representation ([F3], [F5]) and *prioritizing* different types of information ([F4]).

[F3] Visualization. The chosen visualization, linked to our data model ([F1]), allows each block to *manage* and *display* its history effectively. The x-axis represents edits, while the y-axis corresponds to file line count, collapsing all edits to illustrate block changes over time. For example, in Figure 7.1, the dark blue stream represents the `activate` function. In the case of nested blocks (e.g., a method within a class), the colors in the visualization will overlap, such as the violet area on the chart covering the dark blue. At the scrubber's version, the `activate` function grows by approximately 20 lines, reflecting a paste event from "ChatGPT", suggesting to a user that ChatGPT provided a significant contribution at this time. In this way, the visualization itself can serve to answer some questions about the code's *history* on its own. The visualization also contextualizes the annotated timeline of events (see Section 7.2.3-[F7]).

[F4] Significant Edit Events. Meta-Manager manages scale by prioritizing certain versions over others. Each code block listens for specific edit events that occur during its history, such that these events may be annotated along the timeline (Section 7.2.3-[F7]). Edit events of interest include copy-paste events, both from online

[2]We consider a programming-related Google search to be one in which popular programming-related websites appear in the search result list. We acknowledge the potential privacy problems with this feature, and consider the current prototype to mainly be an evaluation of the advantages of doing this, and expect that a more complete tool can provide more control over what is saved from the browser, as in [150].

and from within the IDE, block commenting code, and, given a specific code snippet, when that snippet was edited, added or deleted. When these particular edit types happen, additional meta-information will be captured and shown on the code version, as is the case for the version in Figure 7.1-10 which shows where the code came from, what the user was doing online, who performed the edit, and when it occurred. This meta-information will change given the type of edit (see Figure 7.2 for an example of an in-IDE paste event).

We hypothesize that these edit events will be useful to later developers due to the diverse meta-information they generate, aligning with our earlier discussions on developer information needs. As discussed in Section 7.2.3-[F2], web activities of developers can elucidate code design *rationale* when viewed alongside code versioning. Within the IDE, copy-pasting aids in understanding hidden code *relationships* between the original and pasted sections, assisting in tracking code *provenance*. Block commenting reflects developers exploring different solutions or altering implementation, a code *relationship* typically challenging to trace.

[F5] Zoom and Filter. Another feature Meta-Manager provides to manage scale is through directly interacting with the visualization to reduce the history-space through zooming. Since the number of code versions will increase over time, Meta-Manager allows developers to zoom in to parts of the visualization that they find particularly interesting. The visualization will update to show a slice of the editing history (Figure 7.3), which can be dismissed with a "Reset" button. Users can also *filter* the timeline representation to only show specific edit events in order to further reduce the search space.

[D3] Support Navigation.

In order for the code history to actually be useful for question-answering, developers must be able to *find* the relevant information pieces in service of their questions. Meta-Manager presents this information as code versions that contain meta-information and supports finding these versions through multiple ways.

[F6] Search. Meta-Manager supports searching by both content and by code versions across time. Users can search across time using either code that they have selected in their current code version (Figure 7.2-1, "Search for Selected Code") or directly through the code editor by selecting some code in their file, then using the context menu to select "Meta Manager: Search for Code Across Time". These two searches differ slightly from one another, in that the search using the code box will search *forwards* in time from the specific code version, while the search from the code editor will search *backwards* in time (since the editor always shows the current version of the code). Both searches utilize the edit history by modifying the query given how the code changes across each version. This means that the search will attempt to expand if the selection grows, shrink if the selection shrinks, and update the code query content to match on given variable names and other constructs changing over time.

When a search is performed, the timeline will update with events marked "Search Result" for events affecting the specified code, where the code differs in some way from the previous version. This is to prevent the search results from being flooded with events where the code is exactly the same, but has moved as a result of other code above it being edited. When looking at a search result code version, the part of the code that matched the user's query will be highlighted in orange. The search will also detect significant edits made to the code. This includes when the searched-upon code is initially added, removed, commented out, or commented back in. These events are specifically marked on the timeline with a label corresponding to the type of edit. Searching by content works similarly in that the user can type a query into the search box (Figure 7.2-6) and each code version which includes the searched-upon string will be annotated on the timeline. Searching is fundamental for finding a version that may answer questions of *rationale*, *provenance*, or *history*.

[F7] Annotated Timeline. Meta-Manager leverages the listened-for significant editing events of interest (Section 7.2.3-[F4]) by annotating these events along the visualization's timeline (see Figure 7.1-4). Clicking on these annotations will navigate the user to that particular code version, further reducing the amount of code versions a user needs to look at in order to find potentially useful information, given our hypothesized information needs that will be met with the meta-information captured during these editing events. The timeline will also be annotated with versions to look at when a user performs a search (Section 7.2.3-[F6]). A large barrier to making sense of code history is the challenge of searching through large histories [230]. Meta-Manager attempts to mitigate this barrier through pulling out the most interesting versions using both its data and history model and through leveraging the user's interest given a search query.

[F8] Scrubbing. Within Meta-Manager, users can scrub through code versions (Figure 7.1-1). The scrubbing functionality serves multiple purposes: enabling movement between un-annotated versions along the timeline and providing a quick overview of code changes over time. When the code box is expanded and the user is scrubbing, the code will update for each version. This view complements the visualization's high-level representation of history with its lower-level code history representation and supports varied speeds of historical sensemaking, akin to how a user can scrub through, e.g., a YouTube video and speed up or slow down for targeted viewing. Users can comprehend the code *history* at different levels, aligning with where they are in their sensemaking journey.

We hypothesize that supporting search both by content and across time will help with further bridging the connection between the user's current working context and the *history* of the code. By supporting this more micro-level investigation, in conjunction with the more macro-level scrubbing and visualization mechanisms for understanding history, users of Meta-Manager can answer their questions at varying levels of granularity.

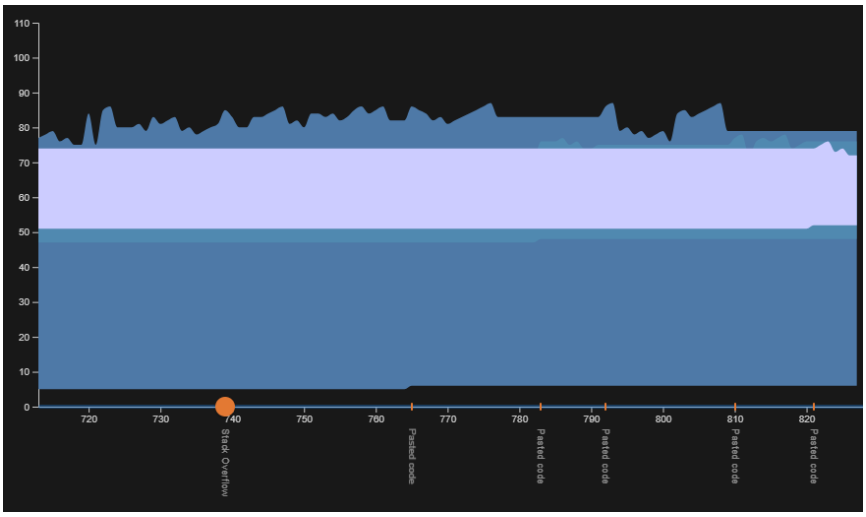

FIGURE 7.3: A zoomed-in portion of the timeline shown in Figure 7.1. This zoomed-in portion shows around 120 edits between Version 710 and Version 830, with the scrubber set around Version 740, when a user pasted code from Stack Overflow.

7.2.4 Implementation

The Meta-Manager, both the Visual Studio Code editor extension and supplementary browser extension, utilize TypeScript for the logic and React [66] (with D3.js [179] for the chart in the Visual Studio Code extension) for the front end. FireStore [58] is used for authenticating the user, establishing a shared connection between the browser extension and Visual Studio Code extension, and logging the code revisions and metadata in the Meta-Manager database.

The code logging in the editor works by utilizing the TypeScript abstract syntax tree (AST) on system launch to parse each file in the user's currently-open repository. The system then attempts to match each block within the parsed-AST to known code entities stored in the database. This matching uses a variety of heuristics including text-matching using the "bag of words" approach discussed in [257], the last-logged structure of the AST (such that known relationships between blocks are prioritized), the user's current Git commit and the logged versions' commits, and the line difference between blocks. In the case that a block in the history is not found in the current version, we consider it deleted, and, to cut down on superfluous versions, do not pull in its history[3]. If an unknown node is found in the current code AST, we assume it is newly created and begin tracking its history. We additionally listen for any document change events to capture whether or not a new block is added or removed to begin tracking or stop tracking the node's history. Each node within the system subsequently manages its own version history and the events that happen to it, such that users can drill down even further to see, e.g., when a particular code block was introduced or removed. While the user is editing their code, each node keeps what we call a "change buffer" of edits that happen to the node, with each

[3]Note that, given the nested structure of the AST, if a missing node's parent is present in the user's code, the user can still see the missing code in the parent's version history and query upon it, if they want.

edit parsed to determine whether it is an edit of interest, such as a paste event or a block code comment. In the case of a copy, the system identifies which node experienced the copy such that a connection between the respective versions can be made between that node and the node that receives the paste event, even though the copy event itself doesn't actually change the code.

For a more comprehensive discussion of the Meta-Manager architecture and data model, please see Appendix A.

7.3 Lab Study

In order to assess how well Meta-Manager performs in helping developers answer historically "hard-to-answer" questions about code history, we ran a small user study. Participants were tasked with using Meta-Manager to explore an unfamiliar code base while using the system to answer questions we asked them about the history of the code, without modifying or running the code. We chose to have a single condition (as opposed to a between or within subjects study design) in which participants used the tool since the questions we asked participants would, without the tool, be unanswerable, meaning there is no real control condition we could grade the experimental condition against. This was done deliberately considering we specifically designed our tool to support answering these types of questions. Thus, ensuring that the tool succeeded in that regard was our primary goal of the study, along with assessing the usability and utility of the tool.

The lab study consisted of a tutorial with Meta-Manager in which the experimenter and participant walked through each feature. Then, the participant and experimenter walked through different parts of the code base and the participant would use Meta-Manager to try and answer each of 8 questions (Table 7.1). Once the participant answered each question, the study ended with a survey to capture participant demographic information, along with their experience using Meta-Manager, and their own history in attempting to answer the types of questions Meta-Manager is designed to help with answering.

7.3.1 Method

Code History Creation

Given that Meta-Manager has not existed long enough to naturally accrue a history log that would be in line with real, prolonged use of the tool, a code history was artificially created. We did this because we did not want to bias the study in favor of the tool purely because there are a small amount of code versions, thus finding an answer to a question is trivial. The artificial code base is based upon a real code base [231] for a Visual Studio Code extension created by an external group unaffiliated with Meta-Manager, which functions similarly to CoPilot. This repository was chosen due to the fact that much of the code centers around the Visual Studio Code API,

which few developers are familiar with, thus lowering the likelihood of a participant performing well purely due to having more background knowledge in the domain.

Our methodology draws from prior approaches that similarly explored developer sensemaking of code history by using a variety of online sources along with the the experimenter's rewriting of the code to create the synthetic code base [117]. Code sourced from different online sources ensures that the code base is unbiased since it does not just use code sourced from one individual who has one implementation style. To create the artificial edits, the first author independently rewrote the code base, following along with the Git commit history in order to capture "real" versions of the code. While writing the code present in each commit, the tool was logging these real versions, but was also recording individual edits (e.g., add 1 line that says `const searchResults = match(searchResults);` in file search.ts on commit 4acb) that were then artificially inserted at realistic intervals across each code's history, given the correct file, time period, and node. The first author intentionally did not write "perfect" code that matched what was in each commit, to account for the more realistic intermittent versions the tool would capture in real usage. The author also intentionally added events that we are particularly interested in investigating, such as copy-pastes, across each file's history, along with simulated copy-paste events that match the frequency reported in prior literature on how often developers copy-paste during a normal programming session [106]. We also added realistic copy and pastes from Stack Overflow and a few from ChatGPT (even though the code was actually written before ChatGPT was available) since these will be increasingly important, with these events occurring less frequently than within-editor copy-pastes. To further validate the realism of the code, we followed the same approach as [104] and asked participants how similar the code was to code they had seen in their own work, with participants reporting the code is, on average, similar to code they have encountered before[4]. Using this technique, we generated a code base consisting of 5,661 edits in 1,328 lines of code across 10 files and 28 different code blocks.

Tutorial

The study session began with the experimenter showing the participant how to use each feature in Meta-Manager. This included an explanation of the visualization (including how to zoom in to the visualization), how to use the scrubber to move through the code versions, how to search from both within the code editor and within a code box, how to filter to view only copy events, paste events, or paste events from online, and how to view each corresponding copy and paste between code versions. This tutorial was done in one of the files within the created code base, such that the participant could see and understand the context of the code base, but none of the code history task questions related to anything in that particular file.

[4]average = 3.8 out of 5, using a 1-to-5 point Likert scale from very dissimilar to very similar

Task

Our main task draws from similar related work [57, 117] in that each participant was required to use Meta-Manager to answer 8 questions. Each question was designed such that it would represent at least one information need we are interested in (see Section 7.2.1) and would require the participant to use some feature of Meta-Manager to answer. Questions also required the participant to perform multiple steps using the tool, such that they would be non-trivial to answer and would represent the more realistic case of using a tool like Meta-Manager, where the full "answer" is multi-faceted and comprised of multiple information pieces. For example, question Q1 asks both what string a regex is matching on and why – "what" refers to the implementation of the regex and is requisite knowledge in order to make a change to the code, while "why" represents the rationale behind the current design and is information that can be used to reason about how a new version should be designed in order to adhere to the original design constraints, goals, and specifications. Table 7.1 lists each question, along with the steps a participant could do in Meta-Manager to answer the question. The solution in the table represents the most efficient way to answer a question, but each question can be answered using other methods. For example, for question Q6, a participant can choose to search forwards in time on the AI generated code, which will pull out instances where that code changed, but they can also scrub through the code versions using the scrubber up to the present to visually see how the code is edited over time. Participants had 10 minutes per question and were not allowed to edit or run the code, or search for information online. When a participant felt they had come to an answer, they were instructed to state their answer and they would move on to the next question.

Questions 1 and 2 were in a file with 90 versions, Question 3 and 4 were in a file with 619 versions, Question 5 was in a file with 727 versions, and questions 6 through 8 were in a file with 1,302 versions.

Analysis

For each participant, we recorded whether or not they got the correct answer for each question and how long it took them to come to the answer. "Correctness" was determined objectively by whether or not they found the correct code or code version that contained the answer and whether the participant's summation of what they learned was accurate. If a participant got only part of a question right, such as understanding in Q1 *what* the regex is matching on but not understanding *why*, the question was still marked as incorrect. If the participant did not finish within 10 minutes, the question was marked as incorrect. We additionally reviewed the video recordings to see what features of the tool and strategies participants used when coming to an answer.

Question in Task	Info. Need (Sec. ??)	Solution	Outcome	Avg. Time Spent If Correct
Q1. In config.ts, there is a regex for search pattern matching. Can you tell me what it is matching on and why?	Rationale	Find paste from ChatGPT, read user's ChatGPT query	6 correct, 1 incorrect	3:28 *(min: 0:58, max: 6:32, std. dev.: 1:59)*
Q2. There is a bug in the commented out `Promise` code. Can you find where the bug was and what happened?	History	Find where `Promise` was initially commented out, where `Promise` came from, and look at versions before that event	4 correct, 3 out-of-time	5:04 *(min: 1:58, max: 7:02, std. dev.: 2:24)*
Q3. Prior to using `parseHTML`, the author was using a different API - what was it and why did they stop using it?	Rationale	Search to when `parseHTML` no longer exists, see what code was there before, and see Stack Overflow post	7 correct	4:35 *(min: 2:23, max: 7:38, std. dev.: 1:46)*
Q4. Recently, some code was added to `search` that came from a different file – can you find that code and explain what changed?	Provenance	Filter to see pasted code, find paste event with code copied from a different file, then search for that code in the file	7 correct	4:47 *(min: 2:00, max: 8:08, std. dev.: 2:29)*
Q5. Look at lines 68 to 70 – there is a commented out `forEach` loop. Can you find the last time it was used and explain why it was removed and what it was replaced with?	Relationships	Search on commented out code, click on "Commented Out" event, find Stack Overflow post near event with replacement code	7 correct	4:24 *(min: 1:57, max: 7:12, std. dev.: 1:44)*
Q6. What code was generated by an AI system and what ended up happening to it?	History	Filter to see ChatGPT code, then search forwards in time on that code	5 correct, 2 out-of-time	6:36 *(min: 2:02, max: 9:15, std. dev.: 1:53)*
Q7. What were all the different things that the programmer tried when setting the `match` variable?	History	Search backwards in time on `match`, look at events	5 correct, 2 incorrect	5:18 *(min: 2:15, max: 8:17, std. dev.: 2:15)*
Q8. Some code from `activate` was moved into a different file. When did this happen and what was the code that was moved?	Provenance	Filter to code copied in `activate`, then see corresponding paste locations	7 correct	3:42 *(min: 1:14, max: 8:35, std. dev.: 2:17)*

TABLE 7.1: Each question that was asked during the task, along with what information need from prior literature it corresponds to, the steps that could be taken in Meta-Manager to answer the question, and how participants performed on the question in terms of correctness and time spent (in minutes). Note that some questions represent more than one information need, such as Q5, which both asks what code is related to the commented out loop, but also why the loop was commented out, which is a rationale question.

7.3.2 Participants

We recruited 7 participants (6 men, 1 woman) using study recruitment channels at our institution, along with advertisements on our social networks. All of the participants were required to have some amount of experience using TypeScript and be familliar with Visual Studio Code. Participant occupations included 4 professional software engineers, 2 researchers, and a financial operations engineer with a computer science background. The average amount of years of professional software engineering was 3.16, self-reported competency with JavaScript was 4.5 (out of 7, where 7 is expert), and an average self-reported competency score of 3 for TypeScript. All study sessions were completed and recorded using Zoom and participants used Zoom to take remote control of the experimenter's computer in order to use the tool. Participants were compensated $25 for completing the study and the study was approved by our institution's Institutional Review Board. Participants 1 through 7 are hereafter referred to as P1 through P7.

7.3.3 Quantitative Results

Participants, on average, were able to correctly answer their questions 85.7% of the time (48 out of 56), and averaged 4 minutes and 52 seconds per question. No participant got every answer correct, and all participants got at least 6 answers correct. Of the 8 failed questions, 5 occurred because the participant ran out of time, and 3 occurred because the participant came to the wrong answer.

Table 7.1 shows question outcome and how long, on average, getting the correct answer to the question took. Questions 3, 4, 5, and 8 were answered correctly by all participants and did not take relatively long to solve. Participants also solved these questions in the most consistent manner, with all participants starting with the same first step that was outlined in Table 7.1 as the intended solution path. Notably, these questions correspond to 3 of the 4 types of hard-to-answer questions discussed in Section 7.2.1, suggesting that the tool was successful in supporting rationale, provenance, and relationship needs. Participant's success with answering provenance questions supports our hypothesis that copy-paste data can help with reasoning about where and how some code came to be. Additionally, in our post-task survey, participants rated Q5 as the most similar to frequently asked questions they have, suggesting that our tool's ability to support finding relationship and rationale information is particularly valuable.

In the post-task survey, participants reacted favorably to Meta-Manager. Participants agreed that they would find Meta-Manager useful for their daily work (avg. 6.14 out of 7, with 7 being "strongly agree") and enjoyed the features provided by Meta-Manager (avg. 6.57 out of 7). Participants particularly liked the ability to see where code from online came from within the context of the IDE as a way to see what the original developer was doing, with one participant stating that they imagined that this will be how they spend "most of their development time in the future, with more code coming from AI" (P7). This, along with participants overall success on Q1, supports our hypothesis that reasoning about rationale can be supported using information traces from AI code-generation tools and related web activity. Participants desired improvements to the user interface to make some of their interactions a bit more clear (avg., 4.3 out of 7), especially with respect to the interaction between the filter, search, and zoom operations. Participants sometimes lost track of what filters they had in place when searching, and vice versa, which caused confusion. Adopting the UI for sorting and filtering from popular shopping websites might help alleviate these problems.

We additionally asked participants to rate each question asked in the study by how often they have encountered similar questions in their own programming experiences on a 5-point scale from "never ask" to "always ask" (Figure 7.4). Participants reported asking questions similar to Q5, which asked about why some code was introduced to replace some other code, most often, with 4 participants stating they "always ask" questions like this. Notably, that is also one of the questions all participants were successfully able to answer, which suggests the tool is useful in

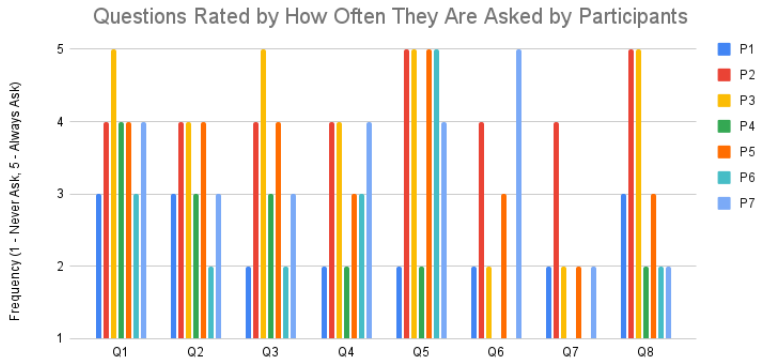

FIGURE 7.4: Each question scored by participants in terms of how often they encounter similar questions in their own programming experiences.

answering this type of common, hard-to-answer question. Only two questions had some participants state they never asked that question, which were the questions corresponding to reasoning about where some code originated from (an AI system, in this case) and what the previous developer had tried when implementing some change. All questions had at least one participant say that they sometimes ask that question, which is both inline with prior research and contributes additional evidence that tooling to answer these questions is valuable.

7.3.4 Qualitative Results

We now explore participants' qualitative experiences using Meta-Manager in terms of how they used its features to answer each question with respect to Meta-Manager's design goals.

[D1] Automatic Code History and Provenance Data

Participants, overall, enjoyed having access to the code-related history and provenance data, especially in the case of code sourced from online. 6 out of 7 participants explicitly stated in the post-task survey that they valued Meta-Manager's ability to capture what code was sourced from online sources, especially ChatGPT, and that these events were explicitly called out on the timeline and filterable. This preference also manifested in their question-answering strategies with participants commonly defaulting to clicking on any event annotation that came from online, especially if they were stumped on what to do to answer a question. P4 clearly articulated this strategy by saying, after using ChatGPT to solve Q1 and why a regex was written this way, "I'm looking at ChatGPT because that worked well last time." Other participants did not immediately understand that the web-based pastes contained additional meta-information that could help with reasoning about "why" some code is the way it is – P3, in attempting to answer Q3, did not look at the Stack Overflow code version which has a Google query explaining why the user switched API methods, and, instead, brute-force searched through the surrounding code versions

and correctly reasoned that the API methods were swapped due to an asynchronous issue given some type changes made between versions. While this strategy was successful in this case, their usage suggests that some users may not see the connection between web-activity and rationale for changes, suggesting that further highlighting the most pertinent "information cues" from these versions (e.g., Stack Overflow question titles) in the user interface, either through the timeline annotation text or within the version itself, may better serve to highlight the significance of the web activity.

[D2] Make Information Scalable

In terms of managing the sheer scale of the version space participants were operating in, the combination of the visualization, zooming, and filtering worked together well to isolate "sub-histories" of the history to explore. A common strategy in answering history and provenance-based questions, used by 4 participants 12 times, was to use the annotated timeline labels as a boundary for a search space, then "zoom" into this space to look at the intermediate versions. For example, P3, in order to answer Q2, used the visualization to identify that there was a large growth in the code base at the end of the history and there was pasted code added at that time – he then zoomed into the end part of the history at the first instance of pasted code when the lines of code grew in order to reason about how the code changed after the addition of it and prior to it being commented out. In this way, participants were able to leverage the significant editing events, not only for meta-information, but also for their ability to segment the information space. This behavior of orienteering [10] to gain an understanding of part of the information landscape is consistent with behaviors exhibited in other information foraging studies [230], suggesting Meta-Manager's feature set supports these processes when navigating a large information-space.

[D3] Support Navigation

In our design of Meta-Manager, we were particularly concerned with making the code history space navigable, given this significant challenge in prior work [117]. To this end, we adapted different techniques for moving through the history including annotated timeline labels, scrubbing, and search. However, one interesting aspect of navigation that we did not explore as much, nor has been explored in related literature to the best of our knowledge, is how navigation worked with respect to moving between the "live" version of the code within the IDE and the historical versions housed with Meta-Manager. Through supporting this relationship, we found multiple design challenges and opportunities.

Navigating Through Time. All participants began each question that had an optimal first step of searching by "searching" – the ubiquity of search made it a common strategy. However, one challenge participants faced when searching through

history was going *too far back* in the history and missing the connection of what they were seeing in the prior version versus what was in the IDE. This happened with 2 participants across 3 questions – the participants would search on the current version of the code and then began clicking through the search results starting from the earliest version. Since our algorithm works across time, it begins at the current code and works backwards by adapting its query given identified changes between versions – since participants could not readily see how the query evolved, jumping to the beginning of the search results in the history (which is the *last* match the algorithm found) was sometimes confusing. Evolving the search query is necessary in order to ensure trivial changes are not disregarded as search results (e.g., switching `const match = 'foo'` to `const match = 'Foo'` where the "f" is now capitalized), but Meta-Manager may be improved by supporting more sophisticated ways of summarizing the search over time or refining which matches should be included. This optimization would also help with another issue participants encountered, where the search would perform differently depending upon what code was selected in the IDE – given a question such as Q5 where participants would begin by searching on a commented-out `forEach` loop, some participants would select the whole loop while others would just select the first line, which would result in the search performing differently given these different yet semantically-similar initial strings.

Communicating "Now" versus "Then". Another challenge of supporting navigation between the IDE and the code history was how to represent the time between the last-logged version and the current version within the IDE. 3 participants did not immediately understand that the end of the code history timeline represented the last-logged version of the code, and not a live view of what was in the IDE. This may be a product of the lab experiment, which did not have participants editing the code which would have let them see how the timeline updates given their edits. Nonetheless, we found that the collapsing of edits by disregarding the amount of time between edits, consistent with prior work [97], did not confuse participants – however, not showing the lapsed time or un-logged edits between the last version and the "live" version within the IDE was confusing, suggesting future work in better showing how the user's current in-IDE version of the code correlates to the larger history.

Navigating Between and Across Files, Spatially and Temporally. Questions that required participants to reason not only about the history of their current file, but how that history relates to the history of *other* files caused confusion. Q2 required participants to reason about how some code in the current file changed, given its relationship to its original copy source in another file. Understanding the original code's intent was necessary in order to better reason about why the code from the question was commented out – participants, with the "See Corresponding Copy" button, can see a preview of what the copied code looked like at the time of the paste. 2 successful participants and all unsuccessful participants struggled to reason

about the connection between the "Corresponding Copy" version (which is on a different version within in a different file), the version of the code that received the paste, and how both of these information pieces related to the code in their current IDE. Future systems may investigate how to better support this reasoning across both time and space through supporting more interactive mechanisms for managing versions, which has shown success in other contexts [204].

7.4 Discussion

We now discuss how Meta-Manager is situated in the larger context of making sense of code and its history, and the role meta-information can play in that process. Prior work has investigated how developers make sense of many variants of the same code and its output [230] and the challenges in doing so – the authors note that this foraging process involves managing similar yet disconnected information patches. We showcase Meta-Manager as an improvement upon that model through extracting and utilizing *meta-information* to serve as strong informational cues to both reduce the number of candidate patches to traverse through and to connect the history into a larger, contextualized narrative. P2 noted that they were essentially "recreating the story" of the code when clicking from event label to event label to answer a question about design rationale – notably, [230] also discusses this phenomenon of information foraging being construed by users as assembling a "story", suggesting that our event labels may serve as one way of structuring these "stories".

When considering code history as a story, this is not dissimilar to the concept of "literate programming" and its philosophies, originally proposed by Donald Knuth [122]. Knuth believed that code should be more naturalistic, written as an expression of an author's reasoning behind solving a computational problem. Programming, in its current state, typically relies on documentation as a way of translating between the lower-level code and its higher-level semantic meaning, with this documentation often spread across various platforms and represented using different modalities such as inline code comments, Git commit messages, GitHub pull requests and issues, and formal design documentation. With the rise of LLMs for code generation, there is a new platform and modality for these natural language descriptions of code, which our work has shown are worth capturing as they can be used for reasoning about design rationale. The not-so-distant future of software engineering may consist primarily of this prompting for generating and modifying code – a future in which whole programs may be constructed predominantly through prompts that can be translated into code narratives not unlike the literate programs Knuth described. In this way, the code serves to describe the lower-level implementation but the higher-level goals and reasoning are communicated through the prompts. Meta-Manager begins to probe at how these forms of code-related meta-information may be captured and presented to help construct these narratives.

To the best of our knowledge, our work with Meta-Manager and its study are the first pieces of research to investigate the provenance of AI-generated code. Prior research has cited the importance of this research thread in order to answer questions such as "does AI-generated code leads to fewer (or more?) build breaks", "what prompts were used to create this code", and if AI-generated code should be under more or less scrutiny during code reviews [24, 228]. These questions, in theory, could be investigated using Meta-Manager through following the development of AI-generated code throughout its life-cycle. Meta-Manager also demonstrates that, through capturing AI code-generation provenance information, other questions and activities can be supported, such as reasoning about code design rationale. Considering more software will likely have source code written by LLMs, tooling support for maintaining and comprehending this code, along with reviewing it for its applicability and correctness, are likely to become more important.

In summarizing our findings and their implications, we find support for the claims that:

- code history data, when properly versioned, contextualized with meta-information, scaled, visualized, and prioritized to support easier navigation, can be used by developers to reason not only about what, how, and when some change happened, but also *why*;
- capturing information traces during the AI code-generation process can be used to support this reasoning; and
- more generally, some information produced as a by-product of authoring code can be mapped to later developers' information needs – thus supporting the research framework (Figure 1.1) proposed in Chapter 1.

Previous systems have captured some of this meta-information, such as Mylyn [75], and typically used this information to support code *authoring* tasks, such as localizing code patches to change, but were less concerned with questions of code comprehension by later developers. Other systems, such as the wide array of code visualization systems (e.g., [257, 266]), usually did not imbue additional information about the context in which the code was created. We show, with Meta-Manager, how these two approaches can be complementary to one another in supporting developer sensemaking tasks.

7.5 Limitations and Threats to Validity

Our study is limited by the fact that we did not have a control condition to compare our results to. While the questions that we asked may be impossible to answer without a tool like Meta-Manager, without a control condition, it is difficult to make any definitive claims about whether or not a participant's ability to answer these questions would result in some measurable difference in terms of code comprehension.

Given the amount of prior work claiming that these are important questions and no question received a "never asked" in the post-task survey, we have evidence to suggest that supporting these questions is useful.

Our study is also limited by the fact that we used an artificial code base, as opposed to a code base generated with prolonged usage of the tool. We feel that choosing to have an artificial code base such that we can simulate the real experience of having many code versions to navigate through allows us to better ensure that Meta-Manager is scalable to support development on a real code project. We attempted to mitigate the potential biases introduced through artificially creating the code base by ensuring that our code changes were produced in a way consistent with real world code editing practices, diversifying where our code came from, and asking our participants whether the code from the study was consistent with code they have seen in their own work. Future work would benefit from assessing how the code versioning works for information seeking by many developers working on the same code base over time.

An additional limitation is that the questions asked during the study were made up by the first author. While each question was derived from previously-reported information needs developers have about unfamiliar code and participants rated each question as a question they at least sometimes ask, additional studies may investigate how developers can use Meta-Manager to answer their own questions about their code in order to better understand how the system supports answering real developers' questions that were not asked in this study. Further, the study in its current form cannot answer how often Meta-Manager would be useful, as we did not capture the full breadth of questions it can be used to answer and the frequency developers ask those types of questions. Previous work, such as [156], and our own reports from our participants suggest that these questions occur semi-frequently and are challenging to answer – nonetheless, future work would improve upon our work by investigating to what extent the breadth of developers' information-seeking behaviors are supported with Meta-Manager, and what extensions to Meta-Manager might help it answer more questions. Our study, instead, focuses on to what extent Meta-Manager and its features do work for answering some of the questions we know from prior literature are difficult to answer.

7.6 Future Work

Our lab study provided some evidence that Meta-Manager helps developers answer hard-to-answer questions about code. However, this was in the context of a developer joining an unfamiliar code base with no real contextual knowledge of the code or its history — while this allowed us to best assess how well the system works in assisting developers in answering these questions in, arguably, the most difficult situation, future work would benefit from seeing how Meta-Manager helps developers when they are working on their *own* code. Open questions remain in this situation —

given developers' own mental models of their code base and, most likely, its history, one can imagine that usage of Meta-Manager may change, as developers' questions about the code base may become more specific since they have more information to work off of. Improvements to Meta-Manager to support more personalized information may be a richer querying system that supports project-specific terms or time constraints, such as "show me all edits to this function between bug fix #124 and now", or allowing users to define their own "events" outside of copy-paste, code commenting, etc. that the system will automatically log as an event of interest. Prior work has supported similar team and project-specific tagging of information in software projects to help with source code navigation [235, 244] – extracting project-specific tagged events as timeline events may also help developers with navigating between code versions.

Additionally, there is existing meta-information about code project history that Meta-Manager does not currently show, such as the boundaries between Git commits along the timeline, or event labels for typical historical events like merged-in pull requests. One can imagine changing the visualization to show visual boundaries between these events, which may also help to better visually convey to a user why some block of code has appeared or disappeared as code commits are merged together, pulled in, or checked out. This may also help make this information more easily navigable, considering foraging through many Git events is a known challenge [116]. We additionally want to support more event capturing, such as when documentation-type code comments are edited,[5] and other interesting code sources, such as CoPilot. CoPilot, in particular, may be particularly interesting for logging events with given that user's edits that prompt CoPilot (especially in the case of natural language prompts to CoPilot to generate some code) may serve as strong signals for what the code author was trying to do. Conveying more of this meta-information in conjunction with more structured time information may give developers more useful navigational way points for searching for code across time. Expanding the types of meta-information Meta-Manager supports may help with answering other types of programming-related questions.

However, while expanding the types of information captured may help later developers, an important concern held by developers and researchers we showed this work to is anxiety around sharing information that may be seen as "embarassing". Multiple researchers asked questions around whether or not there is a way to, perhaps, hide that one copied and pasted from ChatGPT or Stack Overflow since they would not want their coworkers to know they asked a question on one of these sites that they felt they should have known the answer to already. The current version of Meta-Manager was seen as a probe on whether exposing that information to support later developers' sensemaking would be useful and showed the value in

[5] In the current version of Meta-Manager, one can search on a code comment backwards through time in the same way that regular code can be searched on, but better categorizing the type of code comments, such as "TODO's" or "bugs", may make this information more comprehensible and searchable.

doing so, but one can imagine improvements to Meta-Manager that allow for more fine-grained sharing and presentation of information. Ideas include black-listing or white-listing certain websites or subject matters to share information from, choosing to only be listed as "anonymous" on other developers' Meta-Manager timelines, and/or curating your set of queries and copy-pasted website sources prior to pushing them to the Meta-Manager database.

Currently, Meta-Manager does not support saving or sharing specific code versions, queries, or filter settings. There may be situations in which it would be useful to keep track of that information, such as for communicating with collaborators about how and when a bug was introduced [139] or for saving a code version that a developer may be considering reverting back to [116, 117]. This meta-meta information (meta information about the use of the meta information about the code) could be useful to help others perform similar sensemaking to the current user, based on research [69] that multiple people through time often need to repeat previous people's work.

Chapter 8

MMAI: Accelerating Developer Sensemaking with Logs and Large Language Models

8.1 Overview

In this work, we have investigated methods for improving comprehensibility of code and code-related meta-information through strengthening the relationship between these two rich information sources. We have explored the benefits of bridging code and its meta-information across different sensemaking domains including understanding API documentation and comprehending unfamiliar code, and the results have been promising with developers successfully creating and using the contextualized information to perform better on development tasks and answer relevant questions about code.

Despite this success, I recognize that we are currently in a transitory state of software development with the rise of AI tools for programming assistance. These tools are most often used for code generation, with a recent report from developers claiming that around 31% or more of their new code is generated from AI tools [145]; it is expected that this number will continue to grow as AI tools continually improve and the barriers to using them are lowered. While work with LLMs for programming-assistance mostly revolves around improving code generation and understanding capabilities, I see alternative opportunities for supporting programmers.

One cognitively intensive task that is commonly done by programmers, yet has received surprisingly little attention nor support in Software Engineering research is *print debugging*. A recent report claims that, even though sophisticated IDE-provided debuggers have gotten only more powerful, programmers still prefer to program using print statements [19]. Yet, their take-away is not to directly support print debugging but, instead, to make IDE-provided debuggers even better, with the implication that print debugging is sub-optimal and can be eradicated given even more tooling support for sophisticated debuggers. Instead, I argue we should adapt to the tooling developers are already choosing to use in order to make print debugging more powerful. Further, I believe that the meta-information generated during print debugging

can be utilized by LLMs to accelerate developers' sensemaking.

As a case study of these hypotheses, I expanded on the Meta-Manager architecture to create MMAI. MMAI builds upon Meta-Manager in order to directly support print debugging through both handling previously-onerous information collection and tracking tasks by supporting the collection of log values through our own logging package, `mmlog`, and supporting reasoning and synthesizing of log values through connecting into the OpenAI API [197] to query on this even richer set of historical data. We chose to support the collection of log data in conjunction with Meta-Manager's code history and provenance support in order to support more fine-grained question-asking and answering when debugging. For example, consider the query "Why is `foo` equal to `undefined` when it used to be an array of integers?" – with Meta-Manager's node-based data model and historical knowledge, the system has a knowledge of both what `foo` is and what `foo` has previously been. This data is then complemented with the `console.log` values that are captured with our package, `mmlog`, such that the LLM has all of the contextual knowledge necessary in order to answer this otherwise vague question. Through our usage of GPT-4, we make asking this question directly actually possible using retrieval augmented generation (RAG), in which GPT-4 takes that query in a pre-formatted prompt and applies it to the set of historical data that we ingest into the model.

To further isolate a set of questions MMAI would be well-poised to answer, we completed an exploratory interview study. We asked 10 developers about their practices and questions they have when performing print debugging and the sorts of questions they would ask if given access to an oracle that was aware of their debugging practices (similar to [118]). Through an open-coding thematic analysis, we developed 7 queries we believe MMAI can answer through direct, AI-assisted history and output support. Through this ideation and exploration, we contribute MMAI as a prototype system for answering debugging and output-related questions with AI, and a set of developer queries asked in service of supporting print statement debugging that have not been previously supported via tooling.

8.2 Background and Related Work

In order to disambiguate our discussion of print debugging, which utilizes log statements and log values, from other logging contexts, we will briefly discuss the terminology and definitions we will be using from hereafter derived from prior work. We also briefly discuss some of the background and related work on print debugging in service of further motivating and situating our work.

8.2.1 Terminology and Background

- **Print debugging** [169], also called `printf` *debugging* [19, 20, 143] or *trace debugging* [143, 205], is the act of placing lines of code that print information to

a console or terminal at run time in service of determining when, where, and how some problem in the code manifests.

- A **log statement** [78] is the line(s) of code within a developer's source code that invokes a call to print some information to a developer's console, terminal, or other output location. For example, a log statement in JavaScript could look like this: `console.log('myDogRingo ', myDogRingo())`.
- A **log value** is the execution result generated at run time from a *log statement* that appears in the developer's console or other output location. Continuing from the last example, a log value could look like this in a web console (e.g., Google Chrome Developer Tools): `myDogRingo is very cute`, in which the string "`myDogRingo `" is appended to the function call `myDogRingo()`'s return value of "`is very cute`". Note that the form in which a log value appears is dependent upon the development environment and the code within the log statement itself, given that log values are an amalgamation of the contents of the log statement. This amalgamation often includes both static values such as strings and dynamic values such as variables. We further clarify that a *log value* is the direct result of a *log statement* – in contrast, we use the term *output* to refer to the actual result of either the entirety or some part of a software system, which may include a full software application, a graphical output, an algorithmic computation, or some other artifact created by the developer's source code.

Note that the log statements and log values discussed in our work are not the same as the logs discussed in other research about *logging*. Logging is the act of generating long-term output records that characterize the behavior of an often-large enterprise system for later analysis, which contrasts from print debugging, in which log values are not meant to persist indefinitely and are, instead, an act of inquiry by a developer *in situ*. There exists a wide body of Software Engineering research around logging (e.g., a comprehensive literature review that synthesizes over 200 papers [78]), including systems for cleaning messy logs [217] or analyzing and generating workflows of logs [152], and empirical studies of considerations developers have when designing their logs [141]. While some of the problems discussed in the works around dealing with messy logs [217] and reasoning about logs [152] generalize to our work, our design choices for MMAI had to account for in-real-time source code modification, including updating log statements and values, as opposed to running purely post hoc. Further, the nature of inquiry differs between post hoc, large-scale log analysis and ad hoc print debugging – a developer performing post hoc reasoning is often wondering *when* or *how* some observable difference occurred, while a developer debugging wants to know *why* some code *currently* does not work, which may result in reasoning about code changes, but not always. MMAI supports this inquiry through automatically capturing log values that are associated with code

changes, such that the system can draw inferences when comparing the developer's current code to code that previously worked.

8.2.2 Print Debugging

The act of debugging, by definition, is the act of fixing some defect in software. This process can be broken down into steps, called "TRAFFIC" by Zeller [271]: **t**rack the problem, **r**eproduce the failure, **a**utomate and simplify the test case, **f**ind possible infection origins, **f**ocus on the most likely origin, **i**solate the infection, and **c**orrect the defect. Most relevant to print debugging are the mid-to-end points of the process (i.e., find, focus, isolate, and correct), with *find* and *focus* being particularly important. Print debugging is often used for fault localization [256], in which developers place log statements at different places within the source code in order to observe whether or not some code is reached and what the log values are at different points during the execution. This foraging process, when debugging, can be time consuming [195]. Once a developer has formed a hypothesis about where a bug is originating from, log statements can similarly be used to observe the behavior of the problematic software and test potential solutions. Given this lightweight and flexible nature in terms of their implementation and the intent behind their usage, log statements remain the tool of choice when debugging [19, 192].

Despite the benefits of log statements for print debugging, there are some drawbacks to their usage. Challenges can be broken down into two predominant themes – log statement authoring and the implications therein, and log value presentation. *Authoring* log statements inherently introduces clutter to the code base with potentially erroneous information (not dissimilar to the code comment discussion from Chapter 4) – developers must be judicious in cleaning up the log statements when they are no longer needed [19, 78]. Interestingly, failing to clean up these log statements has organically lead to an emergent class of software repository mining research in which these log statements are mined to understand where bugs are more or less prevalent within code bases [13, 260, 261]. Further, some of these logs come from large enterprise systems such as Google3 [261]. This provides evidence that the analysis of log statements with historical data is useful for reasoning about code faults, even developers in large scale and professional settings are using print debugging despite the existence of more sophisticated techniques, and the act of removing these logs is not foolproof. Log value presentation suffers from multiple issues, including how log values are (not) ordered [138], the structure (or lack thereof) of log values [109], and the scale of log values to parse through, especially in enterprise contexts [19]. We attempt to mitigate some of these challenges through smart log value summarization and filtering, while supporting direct inquiry and log statement and log value grouping with our tool, MMAI.

8.3 Exploratory Interview Study

In order to more directly support the types of log statement and value reasoning and management activities we expected MMAI to be helpful with, we completed a formative interview study. The interview results were qualitatively analyzed in order to come up with a set of queries that we could provide direct tooling support for. In summary, we came up with 7 queries we aimed to design support for in MMAI.

8.3.1 Method

Interview Design

The interview was semi-structured and included questions and follow-up questions related to debugging with log statements. Participants began by describing their programming background, and then were asked to describe a recent debugging session, in order to help mentally situate them for the following debugging-related questions. For each high-level debugging-related question (e.g., "(In your debugging session), when you were running your code, how did you find the outputs you were interested in in the console?"), there was a follow-up question that asked participants to come up with queries they had when performing that task and how they would express that query to a log statement history and value oracle. In this way, we sought to tightly couple participants' queries to the tasks that they would support when performing print debugging. Once each high-and-lower level debugging question was asked, participants were asked to share any other debugging or log-statement and value related thought they had that did not organically come up during the interview. The interviews took approximately half an hour, were approved by our institution's Institutional Review Board, and were recorded with video conferencing software. During each interview, I took notes to supplement the recordings and transcripts.

Participants and Recruitment

We recruited 10 participants through our social networks and through our university's various computer science-related Slack work spaces. Participants' professions ranged from professional software engineer, to masters students with previous professional programming experience, to fellow PhD students who actively programmed as part of their research. On average, participants had 5 years of professional development experience, and 7.5 years of total experience. Participants' programming languages of choice ranged from JavaScript and TypeScript to Python, Scala and Go, with the majority (7 participants) primarily using web-based languages and frameworks.

Analysis

To analyze the data, I used an open-coding, thematic analysis while comparing answers by question and follow-up question. More specifically, I investigated what themes were consistent across each question and what queries participants wanted to ask, given the higher-level task or goal that was presented in the original question. Other emergent themes, such as in what context each task was more or less relevant or what other alternative tooling choices participants used when debugging were also captured.

8.3.2 Results and Discussion

Interview participants will be referred to as P1 through P10. Throughout the results I will discuss high-level, emergent themes, along with the queries that would help in supporting the behaviors or tasks described that are associated with each theme.

Participants discussed their log statements as *tools of inquiry*. P7 articulated this sentiment well, stating:

> *I'm working with a somewhat semi-transparent or opaque system, even if it's my own code, and I'm sort of like poking it, or like trying to interpret it. So I'm asking it to tell me things, and it tells me things that I look at in my console... it's kind of like a theory building exercise.* - P7

Thus, the way in which participants structured and placed their log statements within their code to support inquiry varied. Common log statement practices included placing them across branching code to understand control flow and code reachability (4 participants), tracing variable values through execution (4), and checking variable values or data shape (4). While participants' log statements are often semantically related, such as a set of statements checking whether the `if` or `else` block was reached, conventional editors and consoles do not support grouping related lines of log statements or values together. Queries that could support this group-level reasoning include:

- ***Group log statement A, B, and C's values***
- ***What are all the log statements and values related to variable A?***

Participants expressed that *finding* places to insert these print statements was often not a challenge, given their familiarity with the code (e.g., *"I usually have an intuition as to where the problem is, I'll figure out what the problem is from there."* - P4). Instead, a common challenge came from *locating* the relevant log value(s) in the console or log data. Noisy logs, in particular, commonly made locating log values of interest difficult, with participants bemoaning large log value sets due to code being continually re-run in an event loop (P4 and P10) or due to multi-threaded execution (P6 and P8), large enterprise code bases with many log statements (P2), and asynchronous code creating confusing log sequences (P4 and P7). Participants expressed

a desire to filter log values to only log statements that were relevant to their current task, with some participants already using ad hoc strategies to make that filtering possible. Strategies included using a "key-value" system for "labeling" their logs (P1, P2, P3, P6, and P10) in which the "key" will include the variable name and possibly the file name, and the "value" will be the variable itself, such that they can search for that "key" in the console; another more onerous strategy was to remove irrelevant log statements through manually editing the code (P1 and P5), but this strategy broke down if the logs were in files the participant could not edit. Notably, most "relevant" log statements were only the log statements authored within the current debugging session (e.g., *"If it's been more than a little bit of time, [the] logs won't be helpful anymore because I don't remember them."* - P1). Thus, we found a need to support the following queries:

- ***Remove all log values that are not related to variable A.***
- ***Remove all log values not created during this debugging session.***

Once the relevant log values (or groups of values) have been found, participants expressed the challenges they had both in interpreting the log values and in using that information to design a solution. Three participants (P5, P7, and P8) expressed essentially the same sentiment when performing this interpretation while debugging – either the code is wrong or the their understanding of the code is wrong (*"Code doesn't lie"* - P8). Returning to the concept of log statements as tools of inquiry, the lens they provide into the underlying logic and form of data is relatively narrow and fragmented, leading to a large amount of cognitive stress on the participants (*"There's, I guess, a big cognitive load that I'm putting on myself to remember what order everything's supposed to be firing in"* - P4). This synthesis process of reasoning about log statements both in relation to one another and their log values across time was a challenge. Three participants chose to sometimes save log values in other places, such as note pads (P1) or in files with semantically-meaningful names (*"[When doing something tricky] I actually save logs into different text files that have descriptive names, [with] one file called 'multi-thread-on' and one called 'multi-thread-off'."* - P6), in order to more directly track values over time or given code versions. Participants wanted support from an oracle that could track and infer correlations between code changes and output changes (P1, P3, P4, P6, P7, and P10), both in support of finding when a bug was introduced (P1 and P3) and for reasoning about why some code is problematic to help with deepening understanding (P3, P4, P6, P7, and P9). We found support for the following queries:

- ***When did this log statement last produce the correct value?***
- ***Why did this log statement produce the last incorrect value?***
- ***Summarize this log statements' values and find when something interesting happened.***

Other themes that emerged were not directly related to queries about log statements or log values, but included sentiments related to classic debuggers and the ephemeral nature of logging. Three participants expressed their distaste for debugging tools (*"I find debug tools very unnatural like 'you should use breakpoints' and 'step in' are not how I think and are confusing." - P4*), while one participant used these tools dependent upon their type of development (P1), and another used them almost exclusively (P8). Multiple participants, when discussing why they do not use these tools, explained that they lacked formal debugging training (P5) or found the tools too heavy-weight, either due to resource intensiveness or due to having to set up specific development configurations (P6). Interestingly, there was a pervasive sentiment of shame amongst some participants when discussing their preference for print debugging over using more sophisticated debuggers, with some claiming perhaps the preference stems from an inflated ego (*"I think to myself, 'I can get away with debugging this without these robust tools'."* - P4) or that they are just not efficient at debugging (P5 and P7). This sense of shame has not only been echoed but reinforced in other research, with a survey respondent in a related study about developer debugging practices [19] stating that people who use print debugging "never learned how to use a debugger". I see developer's reliance on print statements not as a developer-problem but as a tool problem – clearly, print statements are providing some affordances traditional debugging tools are not, so we should adapt the tools to match the developers' habits, not the other way around, which partially inspires this work.

With respect to the ephemeral nature of print debugging, the act of authoring log statements shares many similarities to the act of creating and making sense of the other forms of meta-information that have been discussed in this thesis. Participants discussed how their logs were often only useful in the moment they were authored (not dissimilar from annotations and other information scraps), and how they sometimes must be synthesized together in order to be useful (similar to a driving motivation for the curation work – see Section 5.2). More broadly, these usages and ad hoc practices imply that traditional tooling is not adequate for performing the complex sensemaking activities the participants want to perform with their print statements. Through this meta-information lens, I adopt a similar approach to my previous systems to assist in sensemaking of programming information, with the meta-information of choice this time being log statements and values.

8.4 Overview of MMAI

In order to support more sophisticated reasoning and management of log statements and values as a form of meta-information, we developed MMAI (Figure 8.1), an expansion of the Meta-Manager model. We begin our overview by discussing a debugging scenario inspired by one of our interviewee's own experiences to show how MMAI may have been used to overcome some of the barriers they discussed.

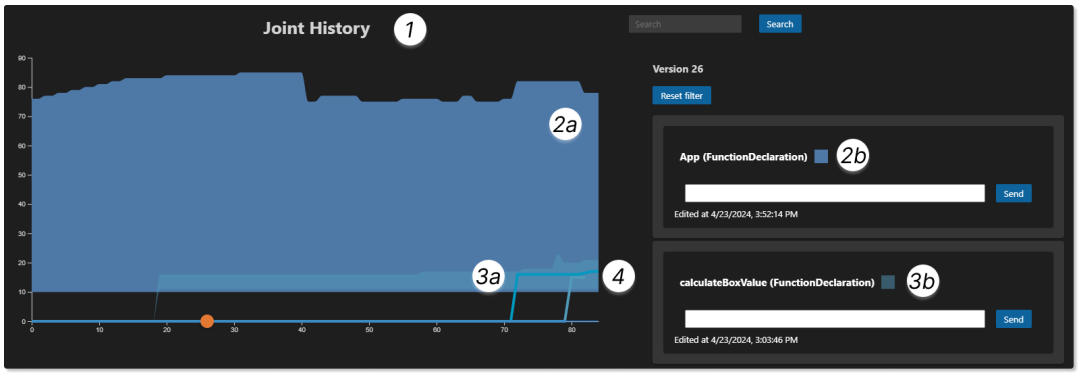

FIGURE 8.1: MMAI as it appears within Visual Studio Code: the pane appears in the bottom area of the editor, with the left area displaying a visualization of the history of user-selected nodes that the user wants to inspect, while the right area displays information about each code node, now including additional output information and an interactive chat component. The two code version boxes for `calculateBoxValue` and `App` are collapsed (see expanded version in Figure 8.3).

We then discuss the MMAI design in-depth, including how it leverages and builds upon the Meta-Manager design, and how MMAI directly supports the queries found in our formative study through both specific design choices in the user interface and through leveraging AI.

8.4.1 Scenario

Linda[1] is creating a web application that takes live-stream data from the popular video game streaming platform, Twitch, and renders the streaming video with additional information, including a drawn box around the main player character. She notices that the coordinates of the box are incorrect, thus the box is not rendering in the correct location. She uses MMAI to assist in this debugging episode. Linda has previously inserted MMAI's `mmlog` statements in this render loop as this part of the code has been problematic in the past. Further, she can choose to filter out the log values they produce as part of MMAI's log management capabilities, thus choosing to leave the `mmlog` lines in her code introduces no additional cost to her when they are not relevant.

Linda begins by finding and placing relevant `mmlog` calls given her line of inquiry. This includes `mmlog` calls in both the back-end where the computation actually occurs and in the front-end where the box is rendered, given the computed coordinates from the back-end. Normally, the structure of the code and the ordering of the log values in the output would make reasoning about these semantically-related yet disparate parts of the code difficult, but MMAI allows for users to *compose* related code histories, log statements, and log values together into a singular visualization (Figure 8.1-1) through selecting calls to `mmlog` in the editor using icons in the gutter (Figure 8.2-3). This allows Linda to not only see the histories and log statements and

[1]This scenario is inspired by a recent debugging episode shared by P4 – "Linda" is used as a pseudonym.

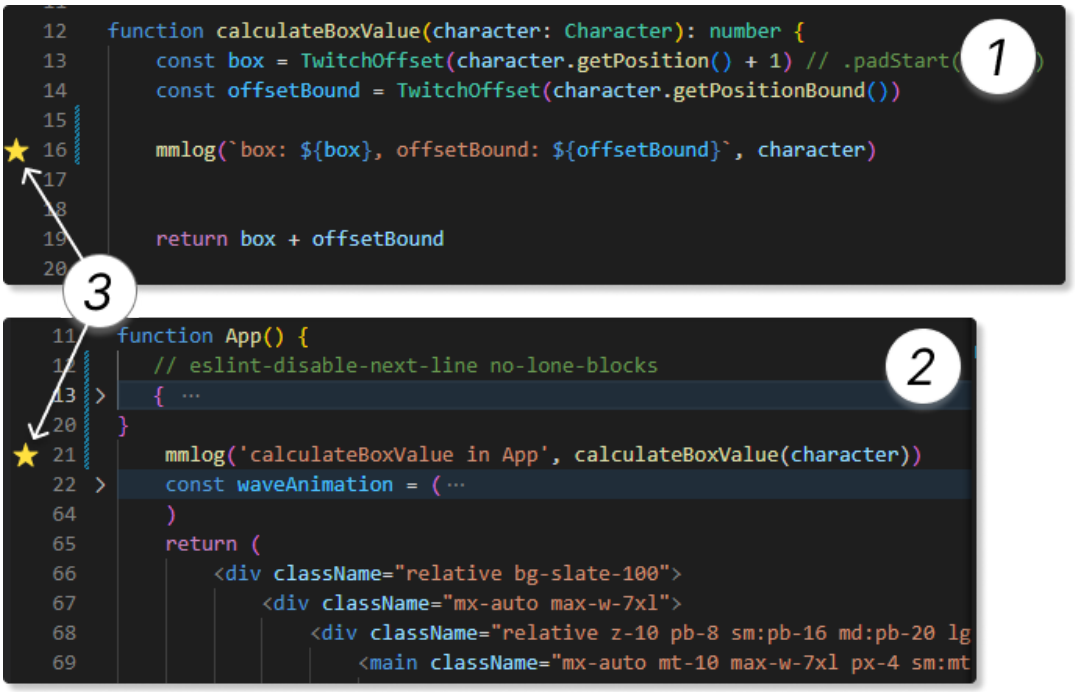

FIGURE 8.2: How `mmlog` calls appear in the editor, including star icons in the gutter to either include or exclude the calls and their corresponding nodes from the rendered "Joint History" visualization (Figure 8.1-1) and code version boxes (Figure 8.1-2b and 3b) in the MMAI pane. Filled-in star icons mean the nodes will be included. Note that these two functions, `calculateBoxValue` and `App`, are in different files.

values in a visually grouped manner, but also allows her to query on the composed history of these otherwise-separate parts of the code.

With the previously-captured log values in MMAI's memory and her set of `mmlog` statements and values of interest, Linda uses MMAI's querying system to compare her current, incorrect log value to the last-known correct value and the corresponding code. In the current code version, she explores the current code version and its run data in her function of interest. In Linda's case, *the amount of log values* is a considerable challenge, given that both the live stream data and the rendering loop run continuously. MMAI provides a high-level summary of the current run's `mmlog` data (Figure 8.3-2B) and how it compares to previous code versions' values of the same data. The LLM performs this comparison – in looking at Linda's data, the LLM finds that the data in the current run includes negative values for the x-coordinates and suggests comparing code versions (Figure 8.3-2D).

In isolating her problem, she wants to know whether the Twitch API is returning incorrect data, thereby interfering with her calculations, or whether the calculations are implemented incorrectly. Through comparing the earlier working implementation shown to her by the LLM on the timeline to the current implementation, the calculation implementation appears the same. She wants to be thorough in her investigation, so she asks MMAI to find other logs related to her calculations (Figure 8.3-2D, "Find other logs related to `box`" button), which updates the MMAI pane with other code versions that include `mmlog` calls that include the variables in her current

FIGURE 8.3: How an expanded code version appears in MMAI. (1) is similar to Meta-Manager (see Figure 7.2), in that information about the edit is shown including when it occurred and what the code looked like at that point in time, along with any additional meta-information (with no additional meta-information for this particular version). (2) shows the optional, supplementary information when `mmlog` is used, including the log statement (2A) (in case multiple `mmlog` statements exist in one version), most recent log statement and value (2B), summarized values across each run on that particular code version (2C) with counts for each output in purple, and computed queries that can be sent to the LLM, given the `mmlog` statement and value (2D).

version. Upon inspecting these versions by similarly inspecting the MMAI code version boxes, she finds the same erroneous values. Using the joint history user interface, she similarly ensures that the front-end rendering values are the same as the back-end computations, so she turns her attention towards the Twitch API.

To further dig down into her problem, she uses the MMAI chat system to directly ask the MMAI AI model about her bug. She begins with the pre-formed query of "Why am I seeing this output" (Figure 8.3-2D, "Why this output?" button) about her erroneous calculation result. The AI model uses the output values on the current code version as reference and the surrounding context including

code, the history of the `App` function given the joint history, and imported APIs in order to inform its answer (see Section 8.4.3). Using the contextualized information, the model responds that the Twitch API sometimes is laggy in its response time, which could lead to the box coordinate calculation using old data, instead of the current data. MMAI additionally generates follow-up questions given the model's response, with one of the generated responses asking how to ensure latency with the Twitch API. Linda chooses to ask that question and receives a response about how JavaScript's `async/await` constructs can resolve this bug. Linda resolves her bug using the model's suggested code.

8.4.2 Detailed Design

In designing MMAI, we sought to provide functionality that both support the needs we found in our formative study (Section 8.3.2) and allowed for varying levels of inquiry through utilizing the historical information Meta-Manager already captures, while improving upon that architecture using state-of-the-art AI. Below, we discuss how the various parts of MMAI support different stages of the print debugging process with respect to the queries asked at each point.

Locating Output Values of Interest

A recurring theme that arose in our formative study was that methods for isolating log statement(s) and values(s) of interest were ad hoc and often broke down when presented with challenges such as large-amounts of log data. To counteract this challenge, MMAI supports mechanisms for both *finding* and *grouping* log values of interest, *filtering* out irrelevant values, and *summarizing* large sets of output data.

Log values are presented in-context within each code version, such that developers can visually compare different values produced by the same log statement and values produced by multiple log statements within the same function. In this way, MMAI utilizes the Meta-Manager's block-based organization of code information, which has been successful in combating issues of scale.

The adopted Meta-Manager data organization naturally lends itself well to filtering out values that are not of interest. The user interface already organizes log statements and values by node, meaning a user can directly navigate and inspect log values in the MMAI pane by node and log statement. With this organization, the user does not need to actively filter out values, and, instead, they can focus on the values they care about.

Developers can further locate related log statements through using MMAI's query to find logs related to the currently in-focus log and its subject matter (see Figure 8.3-2D, "Find other logs..." buttons). MMAI utilizes both the capabilities of its LLM for understanding code and control flow in conjunction with MMAI's knowledge of the AST (another feature inherited from the Meta-Manager architecture) in order to isolate candidate `mmlog` calls that may be related to the identifiers included in the

original `mmlog` argument list (e.g., `box` or `offsetBound` in Figure 8.3). In this way, the user can isolate and synthesize a group of semantically-related but potentially disparate logs and present their values together.

MMAI additionally utilizes the Meta-Manager architecture to improve upon its visualization and presentation of code nodes for synthesizing historical and log data by allowing users to create their own groups of nodes. A user can do this by clicking star icons in the gutter next to `mmlog` calls (Figure 8.2-3), which will add the associated `mmlog` call and its surrounding node and history to a user-defined set, which will then be rendered in the MMAI pane. Given findings from our formative interview (i.e., query 1: "Show me just log statement A, B, and C"), we wanted to directly support this line of inquiry without requiring the user to have to format their log statement in a particular way to allow for text-based search, like the brittle "key-value" pair strategy we saw in our study. Further, since the MMAI visualization is already organized by time and code version, the traditional time-based ordering of information in a console lends itself naturally to this visualization approach.

A feature unique to MMAI is its historical knowledge of log statements and values that are contextualized to a particular code version and optional supplementary meta-information. With this knowledge, MMAI can both summarize the history through retrieval augmented generation and compare the most recent log value(s) to its understanding of the code's intended result and previously-produced values to assist in isolating anomalies. Through these multiple approaches to finding log values of interest, we expect MMAI will be an improvement over the ad hoc mechanisms developers typically employ.

Synthesizing and Understanding Log Values

MMAI allows for developers to make sense of their log values in multiple ways. An advantage of existing within the editor is MMAI can utilize more context when allowing users to prompt the LLM model. To assist in this prompt-creation, MMAI has pre-formatted prompts that the user can just press a button to ask, while also supporting free-form prompt generation. The pre-formed prompt associated with the button "Why this output?" supplies additional meta-information in the prompt including the current code version, a summarized history of the changes before, and information ingested about any external libraries the user may be using, including version numbers taken from the user's `package.json` file. In this way, MMAI attempts to provide as much contextual information it can to support the LLM in coming up with a reasonable explanation.

Despite our best efforts to improve the quality of generated LLM prompts, it is unlikely that a single LLM answer will completely address whatever issue a developer is encountering. We provide follow-up questions to each LLM-generated answer in order to further utilize the ingested context for reasoning through what the problem may be and further reducing the amount of mental processing the user needs to do. In the formative study, P7 discussed how he wanted debugging support

from a historical oracle to not only function as a repository of prior knowledge but to be proactive, not unlike a pair programming partner, in coming up with alternative lines of inquiry and suggestions. We use these follow-up questions as a first pass at attempting to provide this pair-programming type assistance when reasoning about what log value(s) may mean and how to address them.

Through directly supporting composition of logs, MMAI can provide more tailored support for reasoning about log statements and values given more semantically-meaningful relationship. When a user creates a group of `mmlog` statements that are not in the same node[2], MMAI generates a new, synthesized history that organizes and merges the constituent nodes' versions, log statements and values, and any additional meta-information into a singular history that a new LLM agent is trained upon. In this way, the user can ask more complicated questions regarding the relationship of multiple functions, log values, and versions across time that would otherwise be very difficult to ask in a de-contextualized format. For example, this synthesizing of information allows for a user to ask "Why does box go from an object to `undefined` between these two functions?" – with the synthesized history knowledge, our system and the LLM can isolate when the log value was last *not* `undefined`, what was changed since that point in time between both implementations, and come to an answer.

8.4.3 Implementation

MMAI is a Visual Studio Code extension for the JavaScript and TypeScript programming languages. Considering it leverages the Meta-Manager architecture, some features are identical to that system, including the node matching algorithm utilizing our Firestore database. Logs are now, additionally, stored on the database and associated with a parent code version.

Log Data Design and Processing

In designing MMAI, we extended the Meta-Manager architecture to allow for handling of log statements and values with our own Node.js package for logging, `mmlog`. As a brief reminder, Meta-Manager pushes new versions of each node to the database on file save, given whether or not the node has experienced a change that should be saved to the database (see Appendix A for a full discussion on Meta-Manager's event model). Capturing log values at the code version level introduced some complexities to this model that required design and implementation consideration.

One challenge was how to capture whether or not the code was run and what to count as a run. Given that MMAI is supporting web development, specifically, the event-based nature of such development means that code is nearly constantly running and many paths of execution are only triggered when a certain event, such as a

[2]If the `mmlog` statements *are* in the same node, there is no need to generate a synthesized history, given that the node has access to each `mmlog` statements' historical data, including prior versions and log values.

click, occurs. Thus, one version of the code may have hundreds of runs or none at all, depending upon what part of the code the user is testing. For simplicity's sake, we utilize the VS Code API's native debugging API event, `onDidStartDebuggingSession`, as our ground truth for whether or not the user is running their code. Each debugging session has a unique id that we associate with any emitted log values from `mmlog` as a way of grouping runs as an additional mechanism for organizing log value data.

When the user is debugging and has `mmlog` calls in their code, `mmlog` transmits information to MMAI which handles the delegation of the log statement and values to the corresponding node. The `mmlog` package acts as a wrapper around `console.log` and transmits any arguments to the `mmlog` call on a socket connection shared between MMAI and `mmlog`. In order to match `mmlog` calls between the user's source code and the actual log values received at run time, `mmlog` additionally transmits the stack trace, which includes the file name and line number. MMAI takes that stack trace and finds attempts to find a matching file name in its set of managed user source code files. If a file is found, MMAI transmits the `mmlog` statement and values, along with the line numbers to MMAI's document manager which finds the node that contains that line number and `mmlog` statement, such that the node can add the log data to its history (see Figure 8.4 for an example of how log data looks in history).

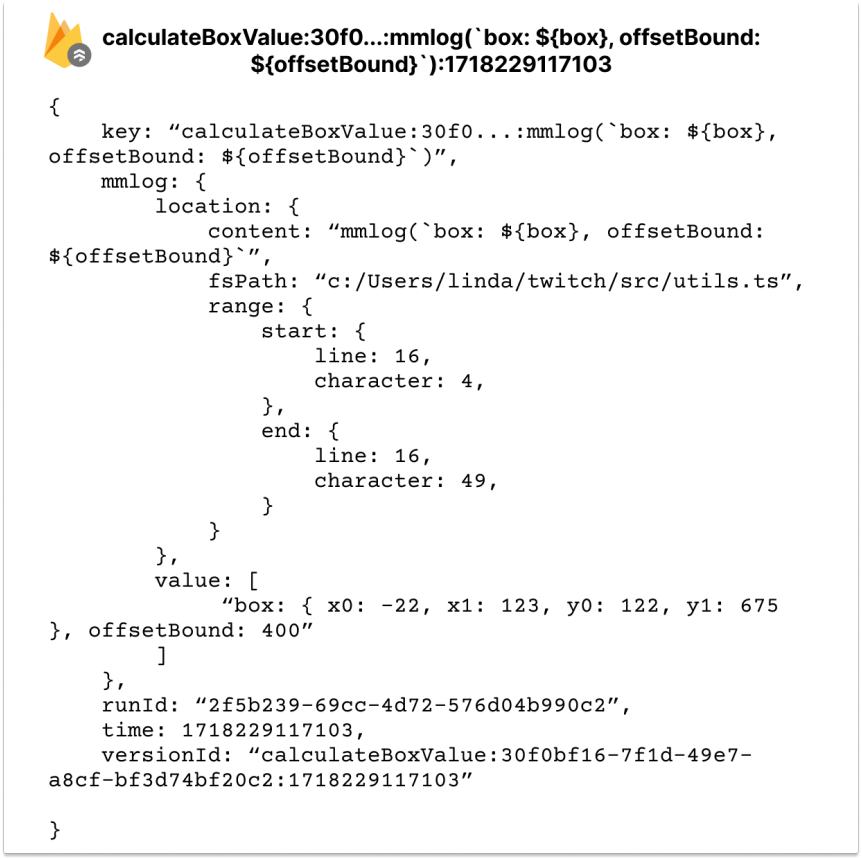

FIGURE 8.4: How an `mmlog` version appears in the MMAI database. One code version owns a collection of these `mmlog` versions dependent upon how many `mmlog` statements the MMAI node has at a particular version and how many times each `mmlog` statement is invoked.

One advantage of handling log statements and values in a VS Code extension is that we can specifically track the log statement lines in the user's source code. Similar to how nodes are handled, each `mmlog` statement is given a unique identifier derived from the parent node's ID and the contents of the `mmlog` statement. If the user edits their `mmlog` statement (e.g., `mmlog('ringo is', myDogRingo())` to `mmlog('ringo is', myDogRingo(), myCatEevee())`), MMAI detects that edit and updates that `mmlog` statement's ID, both within the extension and in the database, such that future `mmlog` values are associated with the same `mmlog` history.

GPT-4 Support

For the GPT-4 support, we chose to have each node of the AST not only manage its code versions and log statements and values, but also manage an intelligent agent in the form of a unique LLM instance trained on that node's history and log data. To do so, for each node, each code version (see Figure A.2 for an example version) and `mmlog` version (Figure 8.4) it contained was concatenated into a long historical record, sorted by time, and fed into that node's GPT-4 instance (see Appendix B for prompts). We chose to do this such that the LLM could ingest more specific data as opposed to all of the history across, e.g., the whole project and suffer from too much context. Further, ingesting that much data would likely exceed the token limit for the model (as of writing, GPT-4 supports a context window of up to 128,000 tokens).

8.5 Discussion

We position MMAI as an exploration of to what extent more robust and domain-specific meta-information collection can assist with developer sensemaking when assisted with LLMs. Without a proper user study, it is hard to know for certain to what extent developer's appreciate the class of features supported by MMAI and whether it does, indeed, cut down on the amount of lower-level information scrounging and synthesizing work typically required during print statement debugging. Nonetheless, we position this work as an introductory exploration into how we can re-frame AI assistance in programming as an opportunity for cutting down on the tedious work often required when developing code. These small productivity gains can add up over time and relieve cognitive strain that can be better applied to more creatively-fulfilling tasks, such as designing a coding solution.

MMAI and its design, like the prior systems, were initially motivated by thinking through challenges I have experienced in my own development and how human behavior is often at odds with the design of more formal software development tools. As discussed previously in Section 8.3.2, more sophisticated debugging support does exist and the productivity gains one can experience through becoming well-versed in using debuggers can be significant. Nonetheless, despite having used debuggers previously, I find myself still almost always choosing to use print statement debugging over a debugger, given the much lower cost and the nature

of my path of inquiry. The insight that print statement debugging is popular is not a novel contribution of this work, yet little tooling has attempted to actively support this practice, with the most relevant and recent work focusing primarily on re-shaping log value data but not supporting other aspects of print statement debugging [109]. This work seeks to celebrate print statement debugging as a useful practice that should not be shamed or written-off as "inefficient" when compared to more powerful mechanisms.

When placing MMAI in the context of my prior work, much of the same challenges and themes arise. In returning to the concept of "information scraps" and meta-information, more broadly, log values when print debugging share the ephemeral nature of this class of information, in that it is typically only useful in the moment it is generated, despite having the potential to answer later questions. However, these logs made in the moment to support print debugging are often not written with the intention of having long-term utility, leading to terseness that makes their interpretation difficult. Indeed, multiple interview participants expressed that they systematically "clean up" these logs once they are done debugging, such that they do not interfere with the business logic of their code or take up valuable space in the console. This tension between keeping this information to support later sensemaking versus removing the information so it does not interfere with other development-related tasks is partially rectified in MMAI's adopted architecture from Meta-Manager which airs on the side of "save everything" considering database storage is relatively cheap in comparison to the human cognitive power of recall. Nonetheless, supporting other information scrap management tasks, such as curation, may make sense when considering extending this work.

Another advantage of the "save everything" design principle in this work is exploring how that data can be utilized by LLMs. A task LLMs are particularly good at is summarizing and retrieving data [264] – we take that idea and utilize it for retrieving log statements and values that can act in service of assisting with a developer's reasoning about code behavior when print debugging. Considering we are using MMAI largely as a retriever and summarizer of a large repository of information as part of the retrieval augmented generation class of LLM usage, we additionally require MMAI to include references to what information it used when coming up with its answer. In this way, we expect that developers will have an easier time ascertaining whether or not an answer is trustworthy, given that they can reference the citations the system is required to produce upon generation. While this is not necessarily a "fool-proof" solution, it is a first step towards assisting in bug isolation and reasoning in a more precise manner than context-free usage of LLMs, given the large amount of contextual and historical data we are feeding the model.

8.6 Future Work and Conclusion

The space of development tasks that may be offloaded or, at the least, accelerated with AI is vast. Yet, tools focus primarily around code generation and, to a slightly lesser extent, code understanding. In this work, we take AI capabilities and place them in a different context, namely supporting print debugging. An obvious and planned next step for this work is to perform a user study to see to what extent MMAI actually succeeds in helping users make sense of and resolve bugs when performing print debugging.

Additionally, there are some technical issues that may arise that require more significant consideration. One such issue is what to do when an output value is too large to store, such as if the user chooses to log the entire DOM on a web page. Potential solutions could be fragmenting the object across multiple database entries or otherwise minifying or truncating the representation, such that it can be stored. An alternative solution is to grant the user more control in terms of what is logged – indeed, when discussing logging objects with P7, they mentioned that they are often only interested in one or two properties, yet specifying those specific properties can be onerous. Having further insight into the intent behind *why* a user is choosing to log a particular value could lead to smarter decisions when choosing what information to push to the database.

In the current implementation of MMAI, we only capture values specifically logged using `mmlog`. This was an intentional design decision, since we wanted to mirror the development practices developers are currently doing with, e.g., `console.log` when print debugging. Nonetheless, this choice results in us not capturing other information that may otherwise be helpful for a developer and an LLM agent when interpreting `mmlog` values post hoc and further contextualizing buggy code. Additional data future versions of MMAI may consider supporting include output data and other information logged to the console (e.g., error messages). Output data, especially graphical output, could be helpful in further understanding how a bug is manifesting. When considering the scenario described earlier (Section 8.4.1), Linda first realized she had a problem in her code given the graphical output, with a box being rendered incorrectly. If herself or another collaborator wants to later understand the bug fix she applied, a capture of the graphical output may be more useful in remembering what the bug was, as opposed to the logged negative numbers. Print debugging, and the log statements and values produced during that process, represent a line of inquiry that is often motivated by both information that is gained through the log values but also information that is generated elsewhere – capturing more of that line of inquiry may result in a more thorough understanding of the developers print debugging practices and may make reasoning about and questioning of history at a later point easier.

Another effect of only capturing log values generated with `mmlog` is that this

requires developers to use an external package and editor extension in order to receive any of the benefits of this work, as opposed to this support being natively supported by current editors and logging functions, like `console.log`. Similar to Meta-Manager and how a developer needs to both have a VS Code extension and Google Chrome browser extension in order to experience the full benefit of that system, we position this work as an exploration of the potential benefits of having this more tightly-integrated working system, in which components of the system are aware of other, relevant contextual meta-information generated in other usually disparate systems. Like Meta-Manager, this approach introduces some risk to developers in terms of information privacy and having their activities potentially monitored. Future work may explore how to allow for better information protection and filtering on the developer's end.

Some future work from Meta-Manager is still relevant here given the relationship between the two projects, such as supporting annotating or sharing specific code versions with collaborators. In the context of print statement debugging, one can imagine "saving" a user's path of inquiry that led to their identification and fix of a bug such that the information can be documented on a bug report or pull request description. This practice of transforming a user's sensemaking process into a shareable format is not dissimilar to the work discussed in the curation chapter (Chapter 5) and lessons learned from that work may apply here.

Given MMAI's implementation as a VS Code extension and the system's knowledge of user's source code, some additional functionalities to support print debugging are possible. Functionalities future versions of MMAI may consider supporting include removing no longer relevant `mmlog` calls if the user wants to clean up their code, directly importing code from LLM responses into the editor, or automatically inserting `mmlog` statements into the code if the user wants to log every instance of a variable or function call. In the current version of MMAI, we chose not to modify the user's code because we did not want to potentially introduce issues given uncertainties around potentially improperly applying an edit and accidentally editing code a user does not want edited and given our lack of certainty in the correctness of our LLM's generated code. Nonetheless, considering a large issue in print debugging is the resulting clean up of log statements and their interference with the console content and business logic of code, automatically removing log statements is an obvious and powerful solution to this problem. We chose to address part of this challenge through supporting filtering of `mmlog` values and allowing users to directly manipulate the MMAI interface to make it clear what `mmlog` statements they wanted to focus on during their current print debugging session.

When looking at the future of software engineering, it is important to keep the "engineer" in engineering. The anxiety that LLM is introducing into many different practices, including software engineering, about job security and what the future of these industries will look like given the disruptive nature of these technologies is making people apprehensive about even joining the field. As an academic, I believe

it is paramount we try and use these technologies to augment and accelerate the skills developers already have as opposed to introducing technologies that replace the engineer. These technologies are not going away, so we have to learn to use them for good.

Chapter 9

Conclusion and Future Work

9.1 Summary of Contributions

The series of work discussed in this thesis explores the utility of supporting developers in both authoring and utilizing meta-information to better make sense of code and documentation. Specifically, the systems provide platforms for authoring and sharing useful and contextualized information about code (Adamite, Catseye, Sodalite), unify various forms of meta-information into one platform with a single interaction method (Catseye, Meta-Manager), utilize artificial intelligence for accelerating sensemaking with meta-information (MMAI), and demonstrate how meta-information may be used to overcome known barriers in documentation usage and answer long-standing, hard-to-answer questions about code (Adamite, Sodalite, Meta-Manager).

My work has made the following contributions:

- A review of literature around developer information needs, systems developed to support some of these needs, and how meta-information about code has or can be used to combat these challenges (chapter 2).
- Identifying that short, easy-to-author notes, when attached to programming-related documents including code and documentation, can address known challenges in information tracking and usage [101, 104] (chapters 3 and 4).
- Adamite, an annotation system designed with specific features to use meta-information as annotations on documentation to overcome known barriers in using software documentation [101] (chapter 3).
- Evidence that developers' notes can be useful for other developers when properly presented given the implicit context communicated through annotated text [101] (chapter 3).
- Catseye, a code annotation system, with features to help a developer keep track of code-related meta-information using a unified annotating interaction [104] (chapter 4).

- A literature review of information needs developers have when making sense of and curating ephemeral, short-form information to assist in sensemaking tasks (chapter 5).
- An exploration of design considerations and probes on how to manage large sets of ephemeral data (i.e., annotations) in a complex and oft-changing environment (i.e., source code) where re-attachment and curation are non-trivial (chapter 5).
- A documentation authoring and maintenance system, Sodalite, that utilizes code meta-information to support developers in assessing the validity of documentation and finding appropriate parts of the code to document [103] (chapter 6).
- A system, Meta-Manager, that can collect, index and store a larger collection of code editing and provenance meta-information that has never previously been collected at scale for answering otherwise unanswerable questions about code history [102] (chapter 7).
- Identifying a set of queries that developers have when performing print debugging that are not currently supported in modern tooling (chapter 8.3.2).
- An LLM-powered system utilizing retrieval augmented generation, MMAI, that captures code meta-information at scale, including output data, to support print debugging (chapter 9).
- A series of lab studies showcasing the usability and utility of these tools in supporting developers' varied information needs in different contexts.

9.2 Discussion and Future Work

In developing the aforementioned systems, including determining design and functionalities, we gained insights into ways in which developer meta-information may be collected, transformed, and presented to assist in developer sensemaking. Below, I briefly reflect on some of the implications and lessons learned from that exploration, along with avenues for further inquiry based upon these implications.

9.2.1 Designing with Information Ephemerality In Mind

When beginning this thesis, I was initially drawn to the idea of supporting lightweight developer sensemaking through notes and annotations due to the seemingly-untapped market of developer note-taking tools. It was only upon beginning to do more literature review that it became clear that developers were, indeed, taking notes, but there was no single tool or practice that was ubiquitous. Instead, developers naturally did what other information workers often do and utilized implicit cues such

as open files or tabs for assisting in recall and their natural workspace (i.e., source code and code comments) for jotting down small notes such as open to-do items. Code comments, in particular, seemed, as a "user interface" for note-taking, to be at odds with the paradigm of note taking – code comments are represented within the source code, meaning they will persist indefinitely, whether that be in the current source code or in a commit contained within a VCS. Further, they do not provide support for richer expression of thought or structuring of information, such as creating follow-up comments in the form of a reply.

Catseye, as opposed to code comments, explores more directly supporting information ephemerality with respect to developer note taking. The conventional design approach when creating an information system is to allow information to persist indefinitely, be accessed indefinitely, and be editable indefinitely, such as in the case of Google Drive, where any document you have access to will exist in your drive unless the permissions are changed or it is deleted (and, even if a Drive document is deleted, it is still accessible in a trash bin for a finite period of time). When starting the curation work (Chapter 5), we were initially motivated and bound to a similar design goal – make annotations *persist* as long as possible (i.e., keep them attached to source code) such that the *user* can derive value from them as long as possible and do something with the annotations at a *later point in time*. However, the more I reflect on this work, the more I wonder whether perhaps that is the wrong direction and the correct direction is "how do we get rid of erroneous or no longer useful information as quickly as possible". Indeed, prior work (e.g., [21, 148]) and my own experiences using Catseye (Section 4.6 and Section 5.3) suggest that much of the information generated is irrelevant past a certain, finite window of usefulness. Not all information is created equally, yet systems often treat the information as such.

Future work may benefit from intelligent mechanisms for automatically curating content. In the context of annotation management, one potentially fruitful avenue may be intelligently inferring relevance to a particular user and the relative importance of an annotation. Thus, curation becomes a function of two different needs that can be calculated dynamically. Meta-Manager showed the promise of keeping track of information while not requiring anything of the user and MMAI showed that this information can be used by an intelligent agent to assist in querying. Relevance to a user could similarly be calculated through a query or through knowledge about the user's current task, which may be inferred from edit patterns [115] and code locations of interest [45]. How to intelligently determine the importance of an annotation is slightly less obvious – ideas we have had include calculating how often the annotation is revisited, how thorough it is (i.e., does it have multiple anchors? Replies? Long annotation content?), and how recently it was created. A truly intelligent user interface for these annotations could have annotations hidden from the user once they have lost relevance to them, with the expectation they likely will never become relevant again, save for if we can ascertain that the annotation will accelerate a later user's sensemaking.

9.2.2 Designing for the Software Engineer of Tomorrow

Much of my work has been predicated on both understanding the information needs of software developers given the large body of Software Engineering and HCI research that discuss these needs, and considering what developer meta-information may be utilized to address those needs. For example, Catseye, Meta-Manager, and MMAI all explore mechanisms for using meta-information to assist in code authoring, comprehension, and debugging, respectively. However, this meta-information capture and transformation may look vastly different in the coming years if developers are not actively writing code and, instead, guiding AI tools to write code. Developers will most likely still need to understand the existing code to design their changes – what meta-information will be most useful for this next developer to formulate, e.g., the correct prompts for a LLM? How will harnessing this meta-information scale as more of a code base becomes sourced primarily from AI?

How I see software development moving is that understanding code at the syntax level will become less important as understanding the intent behind its design and how it was created will become more important. With code generated by AI systems becoming more common, tooling solutions for tracking the creation, evolution, and intent behind this code will become more relevant. Software development has largely followed a pattern where programming is abstracted further and further away from the actual machinery with tooling to support these higher levels of abstraction. For example, in the move from writing assembly code to writing C code, tools were needed to better help with diagnosing memory allocation bugs – in the move from writing code to guiding AI systems (e.g., ChatGPT, GitHub CoPilot) to write code, I see an expansive and rich design space for improving the usability of these tools and the comprehension of the code created by them through leveraging meta-information generated in this authoring process. My recent work with MMAI begins to probe on this thread by utilizing LLMs to support software development tasks not directly involving code generation or understanding, which is primarily what AI code-development tools have been used for. Future work, in moving towards an "IDE of the Future", may similarly explore what emergent tasks come out of software development using LLMs (e.g., prompt (re-)authoring is now a common activity amongst software developers [145]) and what tooling support is necessary to support these tasks versus what classic software development tasks are no longer relevant, if any.

A related research thread, when thinking about how classic software development challenges may change when using LLMs for development, is considering what new "hard-to-answer" questions [136], if any, emerge during this paradigm shift. The Meta-Manager, specifically, was designed to collect contextual, authoring-time meta-information at scale that could be used to answer some of the hard-to-answer questions derived from LaToza and Myers' seminal paper [136]. Some of the questions described in the paper, such as questions about how to best implement some code or how to refactor some code, can potentially be answered by an

LLM agent (whether or not its answer is correct is a different question). Other questions, such as questions of design rationale, which were the largest blockers, rely upon extensive knowledge of the code's history and sometimes information that is invisible to an LLM (e.g., web activity, discussions amongst team members, GitHub PR descriptions), meaning such questions cannot currently be answered by an LLM. Perhaps it is too early yet to do a follow-up study to LaToza's work that seeks to explore whether the hard-to-answer questions have shifted in terms of difficulty or if new questions have emerged, given that LLM-authored code has not been around long enough for developers to know how it is reshaping their question-asking and answering. Still, future work would benefit from understanding the information support needs of developers in a future filled with LLM-generated code.

9.2.3 Designing for Meta-Information as a First-Class Entity

The rise of LLMs holds implications for meta-information. Information provenance, especially, is only going to become more important as users in a variety of contexts need to ascertain *where* some information came from. For example, in the education sector, there is ongoing discussions on how to determine whether assignments and tests are actually being completed by a student versus by an LLM – *where* is the content in the assignment originating from. Given concerns around generative AI in a variety of disciplines, tracing information to its origin is becoming more necessary universally. In the context of software development, I showed that this can be possible given extensive, domain-specific tooling with the Meta-Manager and its connectivity between IDE and browser. While this approach was successful, researchers reasonably expressed some hesitance around sharing where and how some information (code, in this case) came to be. Further, Meta-Manager only operates one layer of provenance deep – in other words, if a user copy pastes code from Stack Overflow, but the code on Stack Overflow was generated by an LLM, Meta-Manager only knows that the code came from Stack Overflow. Designing solutions that can both capture meta-information, including historical and provenance data across platforms and "layers" of sharing, while also respecting a user's right to privacy will become even more necessary as more generative AI content proliferates our day-to-day lives. More extensive information provenance work can also help with answering other research questions – for example, in the context of software engineering, researchers have already begun wondering whether code produced by LLM's introduces more build breaks [24], a question which cannot be answered without code provenance meta-information since AI-generated code is typically indiscernible from developer-authored code.

More broadly, my research has shown that collecting large amounts of highly-contextualized meta-information can assist in sensemaking, both for an initial author and later user. Given how storage sizes have ballooned over time and database services have only gotten more powerful, *storing* this meta-information is not necessarily the challenge. The challenge comes in *making this information useful*. One

design technique nearly all my systems utilize that has been effective is creating strong, traceable, and visual connections between meta-information and the original information source it is derived from. My annotation work, in particular, succeeded primarily due to the in-context nature and presentation of the annotations. My later works, especially Meta-Manager and MMAI, explored how to introduce new dimensions to that information connectivity through exploring relationships such as time, and new presentations, including visualizations. Sodalite and MMAI additionally showed how utilizing that information connectivity can support new tasks, such as determining documentation accuracy, and that meta-information connectivity and the context such a relationship creates can be utilized by an LLM for completing higher-order intelligence tasks, such as reasoning. Future work would benefit from exploring other ways of making meta-information more valuable through further exploring the design space of how to present meta-information, especially when dealing with competing information dimensions, and how to transform and use meta-information and information connections for serving domain-specific tasks. Collecting more meta-information is paramount in order to serve these design and research goals, so I once again implore researchers to treat meta-information as a first class entity.

9.3 Concluding Remarks

Developing code is hard. Anyone who has written code can attest to this – given the sheer amount of information management work involved in balancing tasks, designing solutions, navigating through an IDE, and so on, developers are under intense cognitive strain, while often amassing a large set of highly-contextualized information that is subsequently thrown away or remains tacit within the developer's mind.

My research has explored mechanisms for both extracting developer's tacit knowledge in the form of meta-information and presenting it to accelerate a later developer's sensemaking. Through this research process, I have blended insights from the fields of Human-Computer Interaction and Software Engineering in order to serve developers' specific information needs. HCI research in other domains, such as web browsing and online shopping, has shown that capturing aspects of an initial user's sensemaking journey can accelerate a future user's sensemaking when performing a similar task. In the context of Software Engineering, other research projects primarily focused on assisting developers in completing sensemaking tasks, such as understanding unfamiliar code using in-situ code explanations, while not leveraging this virtuous cycle of sensemaking to assist in other cognitively-demanding aspects of software development. My projects, such as Adamite and Meta-Manager, show that such an approach can be adapted to an information and context-rich discipline such as software development given proper tooling and design.

As the nature of software development evolves with the proliferation of AI tools, there is a need to continue understanding the impact these tools have on developers.

Much of my work has been inspired by understanding that human sensemaking, especially when completing a cognitively intense task such as programming, is messy and tooling solutions are often inadequate in terms of supporting the messiness of such information management. There are large unknowns in terms of how AI tools will change software development – my hope is that the creative parts remain in the hands of the developers and AI tools can act as supplementary aids that adhere and adapt to a developer's sensemaking when performing creative work. Such AI support can come in the form of adopting AI tools to the virtuous sensemaking cycle, where meta-information serves as training data to inform the tool about whatever contextual knowledge is necessary to support my development work. By keeping the developer in the loop, I believe we can create a better developer experience and more reliable code for the eventual end user.

Appendix A

Meta-Manager Architecture and History Model

The basic structure of Meta-Manager's architecture and historical model is as follows:

- On VS Code launch, Meta-Manager attempts to match the project to a known entry in its database through using the GitHub ID (which follows the structure "PROJECT_OWNER/PROJECT_NAME")[1].
  - If a match is found, Meta-Manager ingests all of the information it has for that project, which follows the structure `FILE has many NODES which have many VERSIONS`.
  - If no match is found, continue to next step.
- Meta-Manager traverses through the AST for each file that is a legal JavaScript or TypeScript file.
  - If Meta-Manager has found a match for the project, Meta-Manager attempts to match the nodes[2] it is finding within the AST to known nodes from the database.
    - Matching nodes utilizes multiple heuristics including text matching between the last known version of the node and the current node, if the node is named (e.g., a non-anonymous function), whether the names match, and how close the locations within the AST match (i.e., does the node we are currently comparing have the same parent node(s) as the node from the database?).
  - For each node, either take the closest matching node's already-unique ID or generate a new unique ID – ID's include the name of the node, a colon, then a unique string created by the uuid package [247] (e.g., `recursiveSearch:ccaeaf0c-53bc-45dc-a76e-44a08a5a3a2f`).

[1] We get the GitHub information using the GitHub API.

[2] Note that the nodes we discuss are nodes that are direct parents of block nodes (i.e., a left curly brace with code contained therein followed by a right curly brace) – technically, there are many such as expression nodes and identifier nodes that Meta-Manager does not perform this comparison on.

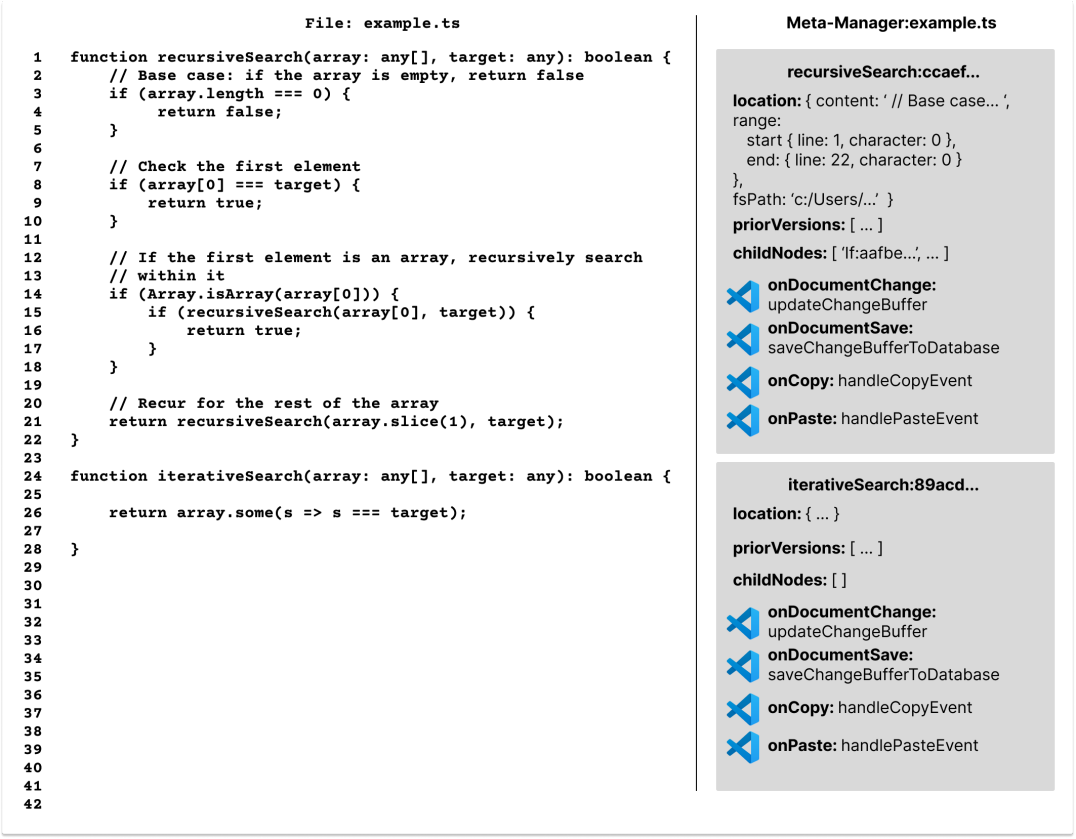

FIGURE A.1: A simplified version of how Meta-Manager's data model and architecture look on a file called "example.ts".

- For each node, instantiate the meta-information event listeners and functionality.
  - Some events are derived from native events within VS Code that each node can subscribe to. This includes:
    - `onDocumentChange`: if some content within the bounds of the node's location is edited, capture whatever change occurred and add it to the node's change buffer as a new version.
    - `onDocumentSave`: if the node has items in its change buffer, push those to the database.
    - `onCopy`[3]: if some content within the bounds of the node's location is copied, capture the code that has been copied and wait for paste event.
    - `onPaste`: if a paste occurs within the bounds of the node's location, capture the code that has been pasted and save any additional meta-information, including web-based meta-information or the copied node's information.

[3]The VS Code API does not surface the copy or paste events so we use a workaround to emit an event when a copy or paste occurs – see this Stack Overflow post for more details.

FIGURE A.2: How a version of a node appears in the Meta-Manager database on Firestore [58]. Note that the id of the version includes the node ID and the time at which the version was captured. This version does not include any additional meta-information, such as web activity.

– Other events are specific to Meta-Manager, such as transmitting information to the Meta-Manager pane when the node is selected. For full details on all Meta-Manager events, see the implementation on GitHub.

Once the matching algorithm is completed, the Meta-Manager's internal representation of the data is similar to Figure A.1.

A version of a node saved on the database would look like this:

Appendix B

MMAI GPT-4 Prompts

B.1 Pre-Processing History Prompt

The following prompt is generated for each `mmlog` version (see Figure 8.4 for an example of how such a `mmlog` version is formatted). The version is joined with the corresponding MMAI code version.

- `$codeLine` is the `mmlog` statement, e.g., Figure 8.4, `mmlog.location.content`.
- `$value` is the `mmlog` value, e.g., Figure 8.4, `mmlog.value`.
- `$codeVersionId` is the unique ID for a particular code version, e.g., Figure A.2, `id` or Figure 8.4, `versionId`.
- `$codeVersionCode` is the code content at a particular code version, e.g., Figure A.2, `location.content`.
- `$runId` is the unique run ID generated by the VS Code Debugging API (see Section 8.4.3), e.g., Figure 8.4, `runId`.

The prompt returns a summarized and indexed version of each `mmlog` statement execution, `$summary` that is kept in that node's instance of GPT-4's memory.

B.1.1 Prompt

You are a code history oracle. You should summarize the output value for the provided code. Summarize the output for the following code: `$codeLine`. `$codeLine` generated the following output value: `$value`. The output came from code version `$codeVersionId` and the following code: `$codeVersionCode`. The output was generated on run `$runId`.

B.2 User Query Prompt

The user query prompt utilizes the output of the Pre-Processing History Prompt, `$summary`, and formats and appends the user's query to it, `$query`.

B.2.1 Prompt

HISTORY DATA `$summary` END HISTORY DATA Answer the user's query. A user is asking this: `$query`. Use the above HISTORY DATA when answering the question. Reference specific version IDs and run IDs.

Bibliography

[1] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley, and Anita Sarma. "Supporting code comprehension via annotations: Right information at the right time and place". In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2020, pp. 1–10.

[2] Emad Aghajani, Gabriele Bavota, Mario Linares-Vásquez, and Michele Lanza. "Automated documentation of android apps". In: *IEEE Transactions on Software Engineering* 47.1 (2019), pp. 204–220.

[3] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. "Software Documentation: The Practitioners' Perspective". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Seoul, South Korea: Association for Computing Machinery, 2020, 590–601. ISBN: 9781450371216. DOI: 10.1145/3377811.3380405. URL: https://doi.org/10.1145/3377811.3380405.

[4] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. "Software Documentation Issues Unveiled". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, 2019, pp. 1199–1210. DOI: 10.1109/ICSE.2019.00122.

[5] Maristella Agosti, Giorgetta Bonfiglio-Dosio, and Nicola Ferro. "A historical and contemporary study on annotations to derive key features for systems design". In: *International Journal on Digital Libraries* 8.1 (2007), pp. 1–19.

[6] Mohammad Allahbakhsh, Boualem Benatallah, Aleksandar Ignjatovic, Hamid Reza Motahari-Nezhad, Elisa Bertino, and Schahram Dustdar. "Quality Control in Crowdsourcing Systems: Issues and Directions". In: *IEEE Internet Computing* 17 (2 2013), pp. 76–81. DOI: 10.1109/MIC.2013.20. URL: https://doi.org/10.1109/MIC.2013.20.

[7] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. "A Study of Visual Studio Usage in Practice". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. 2016, pp. 124–134. DOI: 10.1109/SANER.2016.39.

[8] Muhammad Asaduzzaman, Ahmed Shah Mashiyat, Chanchal K. Roy, and Kevin A. Schneider. "Answering Questions about Unanswered Questions of Stack Overflow". In: *MSR 2013*. San Francisco, CA, USA: IEEE, 2013, pp. 97–100. DOI: 10.1109/MSR.2013.6624015. URL: https://doi.org/10.1109/MSR.2013.6624015.

[9] Alberto Bacchelli and Christian Bird. "Expectations, outcomes, and challenges of modern code review". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 712–721. DOI: 10.1109/ICSE.2013.6606617.

[10] Jeff Baker, Donald Jones, and Jim Burkman. "Using visual representations of data to enhance sensemaking in data exploration tasks". In: *Journal of the Association for Information Systems* 10.7 (2009), p. 2.

[11] Sebastian Baltes, Richard Kiefer, and Stephan Diehl. "Attribution Required: Stack Overflow Code Snippets in GitHub Projects". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 161–163. DOI: 10.1109/ICSE-C.2017.99.

[12] Sebastian Baltes, Christoph Treude, and Stephan Diehl. "SOTorrent: Studying the Origin, Evolution, and Usage of Stack Overflow Code Snippets". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, pp. 191–194. DOI: 10.1109/MSR.2019.00038.

[13] Liang Bao, Qian Li, Peiyao Lu, Jie Lu, Tongxiao Ruan, and Ke Zhang. "Execution anomaly detection in large-scale systems through console log analysis". In: *Journal of Systems and Software* 143 (2018), pp. 172–186. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.05.016. URL: https://www.sciencedirect.com/science/article/pii/S0164121218301031.

[14] Deborah K Barreau. "Context as a factor in personal information management systems". In: *Journal of the American society for information science* 46.5 (1995), pp. 327–339.

[15] David Bawden, Clive Holtham, and Nigel Courtney. "Perspectives on information overload". In: *Aslib proceedings*. Vol. 51. 8. MCB UP Ltd. 1999, pp. 249–255.

[16] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. "Tinkering and gender in end-user programmers' debugging". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '06. Montréal, Québec, Canada: Association for Computing Machinery, 2006, 231–240. ISBN: 1595933727. DOI: 10.1145/1124772.1124808. URL: https://doi.org/10.1145/1124772.1124808.

[17] Andrew Begel, Yit Phang Khoo, and Thomas Zimmermann. "Codebook: Discovering and Exploiting Relationships in Software Repositories". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, 125–134. ISBN: 9781605587196. DOI: 10.1145/1806799.1806821. URL: https://doi.org/10.1145/1806799.1806821.

[18] Andrew Begel and Beth Simon. "Novice Software Developers, All over Again". In: *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER '08. Sydney, Australia: Association for Computing Machinery, 2008, 3–14. ISBN: 9781605582160. DOI: 10.1145/1404520.1404522. URL: https://doi.org/10.1145/1404520.1404522.

[19] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. "On the dichotomy of debugging behavior among programmers". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 572–583. ISBN: 9781450356381. DOI: 10.1145/3180155.3180175. URL: https://doi.org/10.1145/3180155.3180175.

[20] Moritz Beller, Niels Spruit, and Andy Zaidman. "How developers debug". In: *PeerJ Preprints* 5 (2017), e2743v1.

[21] Michael Bernstein, Max Van Kleek, David Karger, and MC Schraefel. "Information scraps: How and why information eludes our personal information management tools". In: *ACM Transactions on Information Systems (TOIS)* 26.4 (2008), pp. 1–46.

[22] Michael Scott Bernstein. "Information Scraps: Understanding and Design". PhD thesis. Citeseer, 2008.

[23] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. "What makes a good bug report?" In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 2008, pp. 308–318.

[24] 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". In: *Queue* 20.6 (2023), 35–57. ISSN: 1542-7730. DOI: 10.1145/3582083. URL: https://doi.org/10.1145/3582083.

[25] Tristan Blanc-Brude and Dominique L. Scapin. "What do people recall about their documents? implications for desktop search tools". In: *Proceedings of the 12th International Conference on Intelligent User Interfaces*. IUI '07. Honolulu, Hawaii, USA: Association for Computing Machinery, 2007, 102–111. ISBN: 1595934812. DOI: 10.1145/1216295.1216319. URL: https://doi.org/10.1145/1216295.1216319.

[26] Jürgen Börstler and Barbara Paech. "The role of method chains and comments in software readability and comprehension—An experiment". In: *IEEE Transactions on Software Engineering* 42.9 (2016), pp. 886–898.

[27] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. "Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, 2503–2512. ISBN: 9781605589299. DOI: 10.1145/1753326.1753706. URL: https://doi.org/10.1145/1753326.1753706.

[28] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. "Example-Centric Programming: Integrating Web Search into the Development Environment". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, 513–522. ISBN: 9781605589299. DOI: 10.1145/1753326.1753402. URL: https://doi.org/10.1145/1753326.1753402.

[29] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code". In: *CHI '09*. CHI '09. Boston, MA, USA: Association for Computing Machinery, 2009, 1589–1598. ISBN: 9781605582467. DOI: 10.1145/1518701.1518944. URL: https://doi.org/10.1145/1518701.1518944.

[30] Margaret Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters, and William Jernigan. "GenderMag: A Method for Evaluating Software's Gender Inclusiveness". In: *Interacting with Computers* 28.6 (Nov. 2016). DOI: 10.1145/3134737. URL: https://doi.org/10.1145/3134737.

[31] CodeSandbox BV. *CodeSandbox: Online Code Editor and IDE for Rapid Web Development*. CodeSandbox BV. 2021. URL: https://codesandbox.io/.

[32] Paul Chandler and John Sweller. "Cognitive load theory and the format of instruction". In: *Cognition and instruction* 8.4 (1991), pp. 293–332.

[33] Joseph Chee Chang, Nathan Hahn, Yongsung Kim, Julina Coupland, Bradley Breneisen, Hannah S Kim, John Hwong, and Aniket Kittur. "When the Tab Comes Due:Challenges in the Cost Structure of Browser Tab Usage". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445585. URL: https://doi.org/10.1145/3411764.3445585.

[34] Joseph Chee Chang, Nathan Hahn, and Aniket Kittur. "Supporting Mobile Sensemaking Through Intentionally Uncertain Highlighting". In: *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. UIST '16. Tokyo, Japan: Association for Computing Machinery, 2016, 61–68. ISBN: 9781450341899. DOI: 10.1145/2984511.2984538. URL: https://doi.org/10.1145/2984511.2984538.

[35] Preetha Chatterjee, Manziba Akanda Nishi, Kostadin Damevski, Vinay Augustine, Lori Pollock, and Nicholas A. Kraft. "What information about code snippets is available in different software-related documents? An exploratory study". In: *SANER 2017*. New York City, NY, USA: IEEE, 2017, pp. 382–386.

[36] Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. "GPTutor: A ChatGPT-Powered Programming Tool for Code Explanation". In: *Artificial Intelligence in Education. Posters and Late Breaking Results, Workshops and Tutorials, Industry and Innovation Tracks, Practitioners, Doctoral Consortium and Blue Sky*. Ed. by Ning Wang, Genaro Rebolledo-Mendez, Vania Dimitrova, Noboru Matsuda, and Olga C. Santos. Cham: Springer Nature Switzerland, 2023, pp. 321–327. ISBN: 978-3-031-36336-8.

[37] J. C. Chen and S. J. Huang. "An empirical analysis of the impact of software development problem factors on software maintainability". In: *Journal of Systems and Software* 82.6 (2009).

[38] Kai Chen, Stephen R. Schach, Liguo Yu, Jeff Offutt, and Gillian Z. Heller. "Open-Source Change Logs". In: *Empirical Software Engineering* 9.3 (Sept. 2004), pp. 197–210. ISSN: 1573-7616. DOI: 10.1023/B:EMSE.0000027779.70556.d0. URL: https://doi.org/10.1023/B:EMSE.0000027779.70556.d0.

[39] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. "Let's Go to the Whiteboard: How and Why Software Developers Use Drawings". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. San Jose, California, USA: Association for Computing Machinery, 2007, 557–566. ISBN: 9781595935939. DOI: 10.1145/1240624.1240714. URL: https://doi.org/10.1145/1240624.1240714.

[40] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. "Self-explanations: How students study and use examples in learning to solve problems". In: *Cognitive science* 13.2 (1989), pp. 145–182.

[41] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. "Self-explanations: How students study and use examples in learning to solve problems". In: *Cognitive science* 13.2 (1989), pp. 145–182.

[42] Parmit K. Chilana, Amy Ko, and James O. Wobbrock. "LemonAid: selection-based crowdsourced contextual help for web applications". In: *CHI 2012*. New York City, NY, USA: ACM, 2012, pp. 1549–1558. DOI: 10.1145/2207676.2208620. URL: https://doi.org/10.1145/2207676.2208620.

[43] Bhavya Chopra, Yasharth Bajpai, Param Biyani, Gustavo Soares, Arjun Radhakrishna, Chris Parnin, and Sumit Gulwani. *Exploring Interaction Patterns for Debugging: Enhancing Conversational Capabilities of AI-assistants*. 2024. arXiv: 2402.06229 [cs.HC]. URL: https://arxiv.org/abs/2402.06229.

[44] Matteo Ciniselli, Niccolò Puccinelli, Ketai Qiu, and Luca Di Grazia. *From Today's Code to Tomorrow's Symphony: The AI Transformation of Developer's Routine by 2030*. 2024. arXiv: 2405.12731 [cs.SE]. URL: https://arxiv.org/abs/2405.12731.

[45] Michael J. Coblenz, Amy J. Ko, and Brad A. Myers. "JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks". In: *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology EXchange*. eclipse '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, 65–69. ISBN: 1595936211. DOI: 10.1145/1188835.1188849. URL: https://doi.org/10.1145/1188835.1188849.

[46] C.R. Cook, J.C. Scholtz, and J.C. Spohrer. *Empirical Studies of Programmers: Fifth Workshop : Papers Presented at the Fifth Workshop on Empirical Studies of Programmers, December 3-5, 1993, Palo Alto, CA*. Human/computer interaction. Norwood, NJ, USA: Ablex Publishing Corporation, 1993. ISBN: 9781567500899. URL: https://books.google.com/books?id=rMmxq8q0CGYC.

[47] Joel Cordeiro, Bruno Antunes, and Paulo Gomes. "Context-based search to overcome learning barriers in software development". In: *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*. 2012, pp. 47–51. DOI: 10.1109/RAISE.2012.6227970.

[48] D. Cordes and M. Brown. "The literate-programming paradigm". In: *Computer* 24.6 (1991), pp. 52–61. DOI: 10.1109/2.86838.

[49] Carlos J Costa, Manuela Aparicio, and Robert Pierce. "Evaluating information sources for computer programming learning and problem solving". In: *Proceedings of the 9th WSEAS International Conference on APPLIED COMPUTER SCIENCE*. 2009, pp. 218–223.

[50] csillag. *Fuzzy Anchoring*. Hypothes.is. 2013. URL: https://web.hypothes.is/blog/fuzzy-anchoring/.

[51] D. Cubranic, G.C. Murphy, J. Singer, and K.S. Booth. "Hipikat: a project memory for software development". In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 446–465. DOI: 10.1109/TSE.2005.71.

[52] Alex Cummaudo, Rajesh Vasa, John Grundy, and Mohamed Abdelrazek. "Requirements of API Documentation: A Case Study Into Computer Vision Services". In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1. DOI: 10.1109/TSE.2020.3047088. URL: https://doi.org/10.1109/TSE.2020.3047088.

[53] Barthélémy Dagenais and Martin P. Robillard. "Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors". In: *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE '10. Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, 127–136. ISBN: 9781605587912. DOI: 10.1145/1882291.1882312. URL: https://doi.org/10.1145/1882291.1882312.

[54] Liwei Dai, Wayne G. Lutters, and Carlie Bower. "Why use memo for all? restructuring mobile applications to support informal note taking". In: *CHI '05 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '05. Portland, OR, USA: Association for Computing Machinery, 2005, 1320–1323. ISBN: 1595930027. DOI: 10.1145/1056808.1056906. URL: https://doi.org/10.1145/1056808.1056906.

[55] Arghavan Moradi Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Hironori Washizaki. "Generative AI for Software Development: A Family of Studies on Code Generation". In: *Generative AI for Effective Software Development*. Ed. by Anh Nguyen-Duc, Pekka Abrahamsson, and Foutse Khomh. Cham: Springer Nature Switzerland, 2024, pp. 151–172. ISBN: 978-3-031-55642-5. DOI: 10.1007/978-3-031-55642-5_7. URL: https://doi.org/10.1007/978-3-031-55642-5_7.

[56] Uri Dekel and James D. Herbsleb. "Reading the documentation of invoked API functions in program comprehension". In: *2009 IEEE 17th International Conference on Program Comprehension*. New York City, NY, USA: IEEE, 2009, pp. 168–177. DOI: 10.1109/ICPC.2009.5090040. URL: https://doi.org/10.1109/10.1109/ICPC.2009.5090040.

[57] Robert Deline, Mary Czerwinski, and George Robertson. "Easing program comprehension by sharing navigation data". In: *VLHCC 2005*. New York City, NY, USA: IEEE, 2005, pp. 241–248. DOI: 10.1109/VLHCC.2005.32. URL: https://doi.org/10.1109/VLHCC.2005.32.

[58] Google Developers. *Cloud Firestore: Store and sync app data at global scale*. Google LLC. 2022. URL: https://firebase.google.com/products/firestore.

[59] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. "Log2: A Cost-Aware Logging Mechanism for Performance Diagnosis". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 139–150. ISBN: 978-1-931971-225. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/ding.

[60] Ekwa Duala-Ekoko and Martin P. Robillard. "Asking and answering questions about unfamiliar APIs: An exploratory study". In: *ICSE 2012*. New York City, NY, USA: IEEE, 2012, pp. 266–276.

[61] Ralph H. Earle, Mark A. Rosso, and Kathryn E. Alexander. "User preferences of software documentation genres". In: *Proceedings of the 33rd Annual International Conference on the Design of Communication*. SIGDOC '15. Limerick, Ireland: Association for Computing Machinery, 2015. ISBN: 9781450336482. DOI: 10.1145/2775441.2775457. URL: https://doi.org/10.1145/2775441.2775457.

[62] S.C. Eick, J.L. Steffen, and E.E. Sumner. "Seesoft-a tool for visualizing line oriented software statistics". In: *IEEE Transactions on Software Engineering* 18.11 (1992), pp. 957–968. DOI: 10.1109/32.177365.

[63] Daniel S. Eisenberg, Jeffrey Stylos, and Brad A. Myers. "Apatite: A New Interface for Exploring APIs". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '10. Atlanta, Georgia, USA: Association for Computing Machinery, 2010, 1331–1334. ISBN: 9781605589299. DOI: 10.1145/1753326.1753525. URL: https://doi.org/10.1145/1753326.1753525.

[64] Elastic. *Free and Open Search: Elasticsearch*. Elastic. 2021. URL: https://www.elastic.co/.

[65] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. "How do API documentation and static typing affect API usability?" In: *ICSE 2014*. New York City, NY, USA: ACM, 2014, pp. 632–642. DOI: 10.1145/2568225.2568299. URL: https://doi.org/10.1145/2568225.2568299.

[66] Facebook. *React - A JavaScript library for building user interfaces*. 2023. URL: https://reactjs.org/.

[67] Jingchao Fang, Yanhao Wang, Chi-Lan Yang, and Hao-Chuan Wang. "NoteCoStruct: Powering Online Learners with Socially Scaffolded Note Taking and Sharing". In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2021. Chap. Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems, pp. 1–5. ISBN: 9781450380959. URL: https://doi.org/10.1145/3411763.3451694.

[68] Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. "Validating AI-Generated Code with Live Programming". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. CHI '24. Honolulu, HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642495. URL: https://doi.org/10.1145/3613904.3642495.

[69] Kristie Fisher, Scott Counts, and Aniket Kittur. "Distributed sensemaking: improving sensemaking by leveraging the efforts of previous users". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: Association for Computing Machinery, 2012, 247–256.

ISBN: 9781450310154. DOI: 10.1145/2207676.2207711. URL: https://doi.org/10.1145/2207676.2207711.

[70] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks". In: *ACM Trans. Softw. Eng. Methodol.* 22.2 (2013). ISSN: 1049-331X. DOI: 10.1145/2430545.2430551. URL: https://doi.org/10.1145/2430545.2430551.

[71] Beat Fluri, Michael Wursch, and Harald C Gall. "Do code and comments co-evolve? on the relation between source code and comment changes". In: *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE. 2007, pp. 70–79.

[72] Daniela Fogli, Giuseppe Fresta, and Piero Mussio. "On electronic annotation and its implementation". In: *AVI 2004*. New York, NY, USA: ACM, 2004, pp. 98–102. DOI: 10.1145/989863.989877. URL: https://doi.org/10.1145/989863.989877.

[73] Andrew Forward and Timothy C. Lethbridge. "The Relevance of Software Documentation, Tools and Technologies: A Survey". In: *Proceedings of the 2002 ACM Symposium on Document Engineering*. DocEng '02. McLean, Virginia, USA: Association for Computing Machinery, 2002, 26–33. ISBN: 1581135947. DOI: 10.1145/585058.585065. URL: https://doi.org/10.1145/585058.585065.

[74] Adam Fourney and Meredith Ringel Morris. "Enhancing Technical Q&A Forums with CiteHistory". In: *ICWSM 2013*. Palo Alto, CA, USA: AAAI, 2013, pp. 1–10.

[75] Thomas Fritz and Gail C Murphy. "Using information fragments to answer the questions developers ask". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 175–184.

[76] Thomas Fritz, Jingwen Ou, Gail C. Murphy, and Emerson Murphy-Hill. "A Degree-of-Knowledge Model to Capture Source Code Familiarity". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, 385–394. ISBN: 9781605587196. DOI: 10.1145/1806799.1806856. URL: https://doi.org/10.1145/1806799.1806856.

[77] J. Froehlich and P. Dourish. "Unifying artifacts and activities in a visual tool for distributed software development teams". In: *Proceedings. 26th International Conference on Software Engineering*. 2004, pp. 387–396. DOI: 10.1109/ICSE.2004.1317461.

[78] Sina Gholamian and Paul A. S. Ward. *A Comprehensive Survey of Logging in Software: From Logging Statements Automation to Log Mining and Analysis*. 2022. arXiv: 2110.12489 [cs.SE]. URL: https://arxiv.org/abs/2110.12489.

[79] GitHub. *GitHub Copilot*. Microsoft. 2024. URL: https://github.com/features/copilot.

[80] Max Goldman and Robert C. Miller. "Codetrail: Connecting source code and web resources". In: *Journal of Visual Languages and Computing* 20.4 (2009). Special Issue on Best Papers from VL/HCC2008, pp. 223–235. ISSN: 1045-926X. DOI: https://doi.org/10.1016/j.jvlc.2009.04.003. URL: https://www.sciencedirect.com/science/article/pii/S1045926X09000263.

[81] Daniel Gonçalves and Joaquim A. Jorge. "Describing documents: what can users tell us?" In: *Proceedings of the 9th International Conference on Intelligent User Interfaces*. IUI '04. Funchal, Madeira, Portugal: Association for Computing Machinery, 2004, 247–249. ISBN: 1581138156. DOI: 10.1145/964442.964494. URL: https://doi.org/10.1145/964442.964494.

[82] Wayne D Gray and Deborah A Boehm-Davis. "Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior." In: *Journal of experimental psychology: applied* 6.4 (2000), p. 322.

[83] Wayne D. Gray and Wai-Tat Fu. "Ignoring perfect knowledge in-the-world for imperfect knowledge in-the-head". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '01. Seattle, Washington, USA: Association for Computing Machinery, 2001, 112–119. ISBN: 1581133278. DOI: 10.1145/365024.365061. URL: https://doi.org/10.1145/365024.365061.

[84] Sanuri Dananja Gunawardena, Peter Devine, Isabelle Beaumont, Lola Piper Garden, Emerson Murphy-Hill, and Kelly Blincoe. "Destructive Criticism in Software Code Review Impacts Inclusion". In: *Proc. ACM Hum.-Comput. Interact.* 6.CSCW2 (2022). DOI: 10.1145/3555183. URL: https://doi.org/10.1145/3555183.

[85] Mark Guzdial and Jennifer Turns. "Effective Discussion Through a Computer Mediated Anchored Forum". In: *The Journal of the Learning Sciences* 9 (4 2000), pp. 437–469. DOI: 10.1207/S15327809JLS0904_3. URL: https://doi.org/10.1207/S15327809JLS0904_3.

[86] Anja Guzzi, Lile Hattori, Michele Lanza, Martin Pinzger, and Arie van Deursen. "Collective Code Bookmarks for Program Comprehension". In: *2011 IEEE 19th International Conference on Program Comprehension*. 2011, pp. 101–110. DOI: 10.1109/ICPC.2011.19.

[87] Anja Guzzi, Martin Pinzger, and Arie van Deursen. "Combining micro-blogging and IDE interactions to support developers in their quests". In: *2010 IEEE International Conference on Software Maintenance*. 2010, pp. 1–5. DOI: 10.1109/ICSM.2010.5609683.

[88] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. "On the use of automated text summarization techniques for summarizing source code". In: *2010 17th Working Conference on Reverse Engineering*. IEEE. 2010, pp. 35–44.

[89] Björn Hartmann, Mark Dhillon, and Matthew K. Chan. "HyperSource: Bridging the Gap between Source and Code-Related Web Sites". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, 2207–2210. ISBN: 9781450302289. DOI: 10.1145/1978942.1979263. URL: https://doi.org/10.1145/1978942.1979263.

[90] Hideaki Hata, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio. "9.6 Million Links in Source Code Comments: Purpose, Evolution, and Decay". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 1211–1221. DOI: 10.1109/ICSE.2019.00123.

[91] Andrew Head, Elena L. Glassman, Björn Hartmann, and Marti A. Hearst. "Interactive Extraction of Examples from Existing Code". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018, 1–12. ISBN: 9781450356206. DOI: 10.1145/3173574.3173659. URL: https://doi.org/10.1145/3173574.3173659.

[92] 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*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, 1–12. ISBN: 9781450359702. DOI: 10.1145/3290605.3300500. URL: https://doi.org/10.1145/3290605.3300500.

[93] Andrew Head, Jason Jiang, James Smith, Marti A. Hearst, and Björn Hartmann. "Composing Flexibly-Organized Step-by-Step Tutorials from Linked Source Code, Snippets, and Outputs". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, 1–12. ISBN: 9781450367080. DOI: 10.1145/3313831.3376798. URL: https://doi.org/10.1145/3313831.3376798.

[94] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. "When Not to Comment: Questions and Tradeoffs with API Documentation for C++ Projects". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 643–653. ISBN: 9781450356381. DOI: 10.1145/3180155.3180176. URL: https://doi.org/10.1145/3180155.3180176.

[95] Marti A. Hearst. "What's Missing from Collaborative Search?" In: *Computer* 47.3 (2014), pp. 58–61. DOI: 10.1109/MC.2014.77.

[96] Ken Hinckley, Xiaojun Bi, Michel Pahud, and Bill Buxton. "Informal Information Gathering Techniques for Active Reading". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: Association for Computing Machinery, 2012, 1893–1896. ISBN: 9781450310154. DOI: 10.1145/2207676.2208327. URL: https://doi.org/10.1145/2207676.2208327.

[97] Reid Holmes and Andrew Begel. "Deep intellisense: a tool for rehydrating evaporated information". In: *Proceedings of the 2008 international working conference on Mining software repositories*. 2008, pp. 23–26.

[98] Reid Holmes and Andrew Begel. "Deep Intellisense: A Tool for Rehydrating Evaporated Information". In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR '08. Leipzig, Germany: Association for Computing Machinery, 2008, 23–26. ISBN: 9781605580241. DOI: 10.1145/1370750.1370755. URL: https://doi.org/10.1145/1370750.1370755.

[99] Lichan Hong and Ed H Chi. "Annotate once, appear anywhere: collective foraging for snippets of interest using paragraph fingerprinting". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, 2009, pp. 1791–1794.

[100] Amber Horvath, Sachin Grover, Sihan Dong, Emily Zhou, Finn Voichick, Mary Beth Kery, Shwetha Shinju, Daye Nam, Mariann Nagy, and Brad Myers. "The Long Tail: Understanding the Discoverability of API Functionality". In: *VLHCC 2019*. New York, NY, USA: IEEE, 2019, pp. 157–161. DOI: 10.1109/VLHCC.2019.8818681. URL: https://doi.org/10.1109/VLHCC.2019.8818681.

[101] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A. Myers. "Understanding How Programmers Can Use Annotations on Documentation". In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3491102.3502095. URL: https://doi.org/10.1145/3491102.3502095.

[102] Amber Horvath, Andrew Macvean, and Brad A. Myers. "Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24), May 11–16, 2024, Honolulu, HI, USA*. CHI '24. Honolulu, HI, USA: Association for Computing Machinery, 2024. DOI: 10.1145/3613904.3642676. URL: https://doi.org/10.1145/3613904.3642676.

[103] Amber Horvath, Andrew Macvean, and Brad A. Myers. "Support for Long-Form Documentation Authoring and Maintenance". In: *VL/HCC 2023*. 2023.

[104] 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*. UIST '22. Bend, OR, USA: Association for Computing Machinery, 2022. ISBN: 9781450393201. DOI: 10.1145/3526113.3545667. URL: https://doi.org/10.1145/3526113.3545667.

[105] Amber Horvath, Mariann Nagy, Finn Voichick, Mary Beth Kery, and Brad A Myers. "Methods for investigating mental models for learners of APIs". In: *CHI LBW '19*. New York, NY, USA: ACM, 2019, pp. 1–6.

[106] Daqing Hou, Patricia Jablonski, and Ferosh Jacob. "CnP: Towards an environment for the proactive management of copy-and-paste programming". In: *2009 IEEE 17th International Conference on Program Comprehension*. 2009, pp. 238–242. DOI: 10.1109/ICPC.2009.5090049.

[107] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. "API Method Recommendation without Worrying about the Task-API Knowledge Gap". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. Montpellier, France: Association for Computing Machinery, 2018, 293–304. ISBN: 9781450359375. DOI: 10.1145/3238147.3238191. URL: https://doi.org/10.1145/3238147.3238191.

[108] Hypothes.is. *Hypothes.is: Annotate the web, with anyone, anywhere.* Hypothes.is. 2012. URL: https://web.hypothes.is/.

[109] Peiling Jiang, Fuling Sun, and Haijun Xia. "Log-It: Supporting Programming with Interactive, Contextual, Structured, and Visual Logs". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3581403. URL: https://doi.org/10.1145/3544548.3581403.

[110] An Ju, Hitesh Sajnani, Scot Kelly, and Kim Herzig. "A case study of onboarding in software teams: Tasks and strategies". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 613–623.

[111] Ján Juhár. "Supporting Source Code Annotations with Metadata-Aware Development Environment". In: *2019 Federated Conference on Computer Science and Information Systems (FedCSIS)*. 2019, pp. 411–420. DOI: 10.15439/2019F161.

[112] Md Mahir Asef Kabir, Sk Adnan Hassan, Xiaoyin Wang, Ying Wang, Hai Yu, and Na Meng. *An empirical study of ChatGPT-3.5 on question answering and code maintenance*. 2023. arXiv: 2310.02104 [cs.SE]. URL: https://arxiv.org/abs/2310.02104.

[113] Vaiva Kalnikaité and Steve Whittaker. "Software or wetware? discovering when and why people use digital prosthetic memory". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '07. San Jose, California, USA: Association for Computing Machinery, 2007, 71–80. ISBN: 9781595935939. DOI: 10.1145/1240624.1240635. URL: https://doi.org/10.1145/1240624.1240635.

[114] David R. Karger and Dennis Quan. "Haystack: a user interface for creating, browsing, and organizing arbitrary semistructured information". In: *CHI '04 Extended Abstracts on Human Factors in Computing Systems*. CHI EA '04. Vienna, Austria: Association for Computing Machinery, 2004, 777–778. ISBN: 1581137036. DOI: 10.1145/985921.985931. URL: https://doi.org/10.1145/985921.985931.

[115] Mik Kersten and Gail C. Murphy. "Mylar: A Degree-of-Interest Model for IDEs". In: *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*. AOSD '05. Chicago, Illinois: Association for Computing Machinery, 2005, 159–168. ISBN: 1595930426. DOI: 10.1145/1052898.1052912. URL: https://doi.org/10.1145/1052898.1052912.

[116] Mary Beth Kery, Amber Horvath, and Brad Myers. "Variolite: Supporting Exploratory Programming by Data Scientists". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. CHI '17. Denver, Colorado, USA: Association for Computing Machinery, 2017, 1265–1276. ISBN: 9781450346559. DOI: 10.1145/3025453.3025626. URL: https://doi.org/10.1145/3025453.3025626.

[117] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. "Towards Effective Foraging by Data Scientists to Find Past Analysis Choices". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, 1–13. ISBN: 9781450359702. DOI: 10.1145/3290605.3300322. URL: https://doi.org/10.1145/3290605.3300322.

[118] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. "The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, 2018, 1–11. ISBN: 9781450356206. DOI: 10.1145/3173574.3173748. URL: https://doi.org/10.1145/3173574.3173748.

[119] Ranim Khojah, Mazen Mohamad, Philipp Leitner, and Francisco Gomes de Oliveira Neto. *Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice*. 2024. arXiv: 2404.14901 [cs.SE]. URL: https://arxiv.org/abs/2404.14901.

[120] Raphaël Khoury, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. "How Secure is Code Generated by ChatGPT?" In: *arXiv preprint arXiv:2304.09655* (2023).

[121] Aniket Kittur, Andrew M. Peters, Abdigani Diriye, Trupti Telang, and Michael R. Bove. "Costs and benefits of structured information foraging". In: *CHI 2013*. New York, NY, USA: ACM, 2013, pp. 2989–2998.

[122] D. E. Knuth. "Literate Programming". In: *The Computer Journal* 27.2 (Jan. 1984), pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. eprint: https://academic.oup.com/comjnl/article-pdf/27/2/97/981657/270097.pdf. URL: https://doi.org/10.1093/comjnl/27.2.97.

[123] Amy J. Ko, Htet Aung, and Brad A. Myers. "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks". In: *Proceedings of the 27th International Conference on Software Engineering*. ICSE '05. St. Louis, MO, USA: Association for Computing Machinery, 2005, 126–135. ISBN: 1581139632. DOI: 10.1145/1062455.1062492. URL: https://doi.org/10.1145/1062455.1062492.

[124] Amy J Ko, Robert DeLine, and Gina Venolia. "Information needs in collocated software development teams". In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 344–353.

[125] Amy J Ko, Thomas D LaToza, and Margaret M Burnett. "A practical guide to controlled experiments of software engineering tools with human participants". In: *Empirical Software Engineering* 20.1 (2015), pp. 110–141.

[126] Amy J. Ko and Brad A. Myers. "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: Association for Computing Machinery, 2008, 301–310. ISBN: 9781605580791. DOI: 10.1145/1368088.1368130. URL: https://doi.org/10.1145/1368088.1368130.

[127] 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*. CHI '04. Vienna, Austria: Association for Computing Machinery, 2004, 151–158. ISBN: 1581137028. DOI: 10.1145/985692.985712. URL: https://doi.org/10.1145/985692.985712.

[128] Amy J. Ko and Brad A. Myers. "Finding Causes of Program Output with the Java Whyline". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09. Boston, MA, USA: Association for Computing Machinery, 2009, 1569–1578. ISBN: 9781605582467. DOI: 10.1145/1518701.1518942. URL: https://doi.org/10.1145/1518701.1518942.

[129] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks". In: *IEEE Transactions on Software Engineering* 32.12 (2006), pp. 971–987. DOI: 10.1109/TSE.2006.116.

[130] Amy J. Ko and Yann Riche. "The role of conceptual knowledge in API usability". In: *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2011, pp. 173–176. DOI: 10.1109/VLHCC.2011.6070395.

[131] Amy J. Ko and Bob Uttl. "Individual differences in program comprehension strategies in unfamiliar programming systems". In: *11th Annual Workshop on Program Comprehension*. New York, NY, USA: IEEE, 2003, pp. 175–184. DOI: 10.1109/WPC.2003.1199201. URL: https://doi.org/10.1109/WPC.2003.1199201.

[132] Oleksii Kononenko, Olga Baysal, and Michael W. Godfrey. "Code Review Quality: How Developers See It". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, 1028–1038. ISBN: 9781450339001. DOI: 10.1145/2884781.2884840. URL: https://doi.org/10.1145/2884781.2884840.

[133] Mik Lamming, Peter Brown, Kathleen Carter, Margery Eldridge, Mike Flynn, Gifford Louie, Peter Robinson, and Abigail Sellen. "The design of a human memory prosthesis". In: *The Computer Journal* 37.3 (1994), pp. 153–163.

[134] Mark W Lansdale. "The psychology of personal information management". In: *Applied ergonomics* 19.1 (1988), pp. 55–66.

[135] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. "Program comprehension as fact finding". In: *ESEC-FSE 2007*. New York, NY, USA: ACM, 2007, pp. 361–270.

[136] Thomas D. LaToza and Brad A. Myers. "Hard-to-Answer Questions about Code". In: *Evaluation and Usability of Programming Languages and Tools*. PLATEAU '10. Reno, Nevada: Association for Computing Machinery, 2010. ISBN: 9781450305471. DOI: 10.1145/1937117.1937125. URL: https://doi.org/10.1145/1937117.1937125.

[137] 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*. ICSE '06. Shanghai, China: Association for Computing Machinery, 2006, 492–501. ISBN: 1595933751. DOI: 10.1145/1134285.1134355. URL: https://doi.org/10.1145/1134285.1134355.

[138] Tom Lauwaerts, Carlos Rojas Castillo, Robbert Gurdeep Singh, Matteo Marra, Christophe Scholliers, and Elisa Gonzalez Boix. "Event-Based Out-of-Place Debugging". In: *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*. MPLR '22. Brussels, Belgium: Association

for Computing Machinery, 2022, 85–97. ISBN: 9781450396967. DOI: 10.1145/3546918.3546920. URL: https://doi.org/10.1145/3546918.3546920.

[139] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. "How Programmers Debug, Revisited: An Information Foraging Theory Perspective". In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 197–215. DOI: 10.1109/TSE.2010.111.

[140] Timothy C. Lethbirdge, Janice Singer, and Andrew Forward. "How software engineers use documentation: the state of the practice". In: *IEEE Software* 20 (6 Nov. 2003), pp. 35–39. DOI: 10.1109/MS.2003.1241364. URL: https://doi.org/10.1109/MS.2003.1241364.

[141] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E. Hassan. "A Qualitative Study of the Benefits and Costs of Logging From Developers' Perspectives". In: *IEEE Transactions on Software Engineering* 47.12 (2021), pp. 2858–2873. DOI: 10.1109/TSE.2020.2970422.

[142] Hongwei Li, Zhenchang Xing, Xin Peng, and Wenyun Zhao. "What help do developers seek, when and how?" In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 142–151. DOI: 10.1109/WCRE.2013.6671289.

[143] Xiangqi Li and Matthew Flatt. "Medic: metaprogramming and trace-oriented debugging". In: *Proceedings of the Workshop on Future Programming*. FPW 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, 7–14. ISBN: 9781450339056. DOI: 10.1145/2846656.2846658. URL: https://doi.org/10.1145/2846656.2846658.

[144] 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: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, 435–447. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00047. URL: https://doi.org/10.1109/ICSE48619.2023.00047.

[145] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges". In: *Proceedings of the 46th International Conference on Software Engineering*. ICSE '24. To appear. Lisbon, Portugal, 2024. URL: arXiv:2303.17125.

[146] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. "ChangeScribe: A Tool for Automatically Generating Commit Messages". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. 2015, pp. 709–712. DOI: 10.1109/ICSE.2015.229.

[147] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation". In: *arXiv preprint arXiv:2305.01210* (2023).

[148] Michael Xieyang Liu, Jane Hsieh, Nathan Hahn, Angelina Zhou, Emily Deng, Shaun Burley, Cynthia Taylor, Aniket Kittur, and Brad A. Myers. "Unakite: Scaffolding Developers' Decision-Making Using the Web". In: *UIST 2019*. New York, NY, USA: ACM, 2019, pp. 67–80.

[149] Michael Xieyang Liu, Aniket Kittur, and Brad A. Myers. "Crystalline: Lowering the Cost for Developers to Collect and Organize Information for Decision Making". In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. DOI: 10.1145/3491102.3501968. URL: https://doi.org/10.1145/3491102.3501968.

[150] Michael Xieyang Liu, Aniket Kittur, and Brad A. Myers. "To Reuse or Not To Reuse? A Framework and System for Evaluating Summarized Knowledge". In: *Proc. ACM Hum.-Comput. Interact.* 5.CSCW1 (2021). DOI: 10.1145/3449240. URL: https://doi.org/10.1145/3449240.

[151] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. "Automatic generation of pull request descriptions". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 176–188.

[152] Steven Locke, Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Wei Liu. "LogAssist: Assisting Log Analysis Through Log Summarization". In: *IEEE Transactions on Software Engineering* 48.9 (2022), pp. 3227–3241. DOI: 10.1109/TSE.2021.3083715.

[153] Yimeng Ma, Yu Huang, and Kevin Leach. "Breaking the Flow: A Study of Interruptions During Software Engineering Activities". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639079. URL: https://doi.org/10.1145/3597503.3639079.

[154] Walid Maalej and Hans-Jorg Happel. "From work to word: How do software developers describe their work?" In: *2009 6th IEEE International Working Conference on Mining Software Repositories*. 2009, pp. 121–130. DOI: 10.1109/MSR.2009.5069490.

[155] Walid Maalej and Martin P. Robillard. "Patterns of Knowledge in API Reference Documentation". In: *IEEE Transactions on Software Engineering* 39.9 (2013), pp. 1264–1282. DOI: 10.1109/TSE.2013.12.

[156] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. "On the Comprehension of Program Comprehension". In: *Transactions on Software Engineering* 23 (4 2014), pp. 1–37. DOI: 10.1145/2622669. URL: https://doi.org/10.1145/2622669.

[157] Catherine C. Marshall and Sara Bly. "Saving and Using Encountered Information: Implications for Electronic Periodicals". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '05. Portland, Oregon, USA: Association for Computing Machinery, 2005, 111–120. ISBN: 1581139985. DOI: 10.1145/1054972.1054989. URL: https://doi.org/10.1145/1054972.1054989.

[158] Anneliese von Mayrhauser and A Marie Vans. "Hypothesis-driven understanding processes during corrective maintenance of large scale software". In: *1997 Proceedings International Conference on Software Maintenance*. IEEE. 1997, pp. 12–20.

[159] Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. "Recommending Source Code Examples via API Call Usages and Documentation". In: *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. RSSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, 21–25. ISBN: 9781605589749. DOI: 10.1145/1808920.1808925. URL: https://doi.org/10.1145/1808920.1808925.

[160] Michael Meng, Stephanie M Steinhard, and Andreas Schubert. "How developers use API documentation: an observation study". In: *Communication Design Quarterly* 7 (2 2019), pp. 40–49. DOI: 10.1145/3358931.3358937. URL: https://doi.org/10.1145/3358931.3358937.

[161] Michael Meng, Stephanie Steinhardt, and Andreas Schubert. "How developers use API documentation: an observation study". In: *Commun. Des. Q. Rev* 7.2 (2019), 40–49. DOI: 10.1145/3358931.3358937. URL: https://doi.org/10.1145/3358931.3358937.

[162] Microsoft. *GitHub*. Microsoft. 2023. URL: https://github.com.

[163] Microsoft. *TypeScript: JavaScript with Syntax for Types.* Microsoft. 2023. URL: https://www.typescriptlang.org/.

[164] Microsoft. *Visual Studio Code*. Microsoft. 2023. URL: https://code.visualstudio.com/.

[165] Microsoft. *Webview API | Visual Studio Code Extension API*. Microsoft. 2023. URL: https://code.visualstudio.com/api/extension-guides/webview.

[166] Edward Misback, Zachary Tatlock, and Steven L Tanimoto. "Magic Markup: Maintaining Document-External Markup with an LLM". In: *arXiv preprint arXiv:2403.03481* (2024).

[167] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. "Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. CHI '24. Honolulu, HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3641936. URL: https://doi.org/10.1145/3613904.3641936.

[168] Gail C Murphy, Mik Kersten, Martin P Robillard, and Davor Čubranić. "The emergent structure of development tasks". In: *European Conference on Object-Oriented Programming*. Springer. 2005, pp. 33–48.

[169] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. "Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies". In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '08. Portland, OR, USA: Association for Computing Machinery, 2008, 163–167. ISBN: 9781595937995. DOI: 10.1145/1352135.1352191. URL: https://doi.org/10.1145/1352135.1352191.

[170] Emerson Murphy-Hill, Jillian Dicker, Margaret Morrow Hodges, Carolyn D. Egelman, Ciera Jaspan, Lan Cheng, Elizabeth Kammer, Ben Holtz, Matthew A. Jorde, Andrea Knight Dolan, and Collin Green. "Engineering Impacts of Anonymous Author Code Review: A Field Experiment". In: *IEEE Transactions on Software Engineering* 48.7 (2022), pp. 2495–2509. DOI: 10.1109/TSE.2021.3061527.

[171] Emerson Murphy-Hill, Ciera Jaspan, Carolyn Egelman, and Lan Cheng. "The Pushback Effects of Race, Ethnicity, Gender, and Age in Code Review". In: *Commun. ACM* 65.3 (2022), 52–57. ISSN: 0001-0782. DOI: 10.1145/3474097. URL: https://doi.org/10.1145/3474097.

[172] Brad A. Myers and Jeffrey Stylos. "Improving API Usability". In: *Communications of the ACM* 59.6 (2016), pp. 62–69. DOI: 10.1145/2896587. URL: https://doi.org/10.1145/2896587.

[173] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. "Using an LLM to Help With Code Understanding". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639187. URL: https://doi.org/10.1145/3597503.3639187.

[174] Daye Nam, Brad Myers, Bogdan Vasilescu, and Vincent Hellendoorn. "Improving API Knowledge Discovery with ML: A Case Study of Comparable API Methods". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 1890–1906. DOI: 10.1109/ICSE48619.2023.00161.

[175] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. "What makes a good code example?: A study of programming Q&A in StackOverflow". In: *ICSM 2012*. New York, NY, USA: IEEE, 2012, pp. 25–34.

[176] Mathieu Nassif and Martin P. Robillard. "Revisiting Turnover-Induced Knowledge Loss in Software Projects". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 261–272. DOI: 10.1109/ICSME.2017.64.

[177] Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q Feldman. "How Beginning Programmers and Code LLMs (Mis)read Each Other". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. CHI '24. Honolulu, HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642706. URL: https://doi.org/10.1145/3613904.3642706.

[178] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. "What programmers really want: results of a needs assessment for SDK documentation". In: *SIGDOC 2002*. New York, NY, USA: ACM, 2002, pp. 133–141. DOI: 10.1145/584955.584976. URL: https://doi.org/10.1145/584955.584976.

[179] Observable. *D3 by Observable | The JavaScript library for bespoke data visualization*. 2023. URL: d3js.org.

[180] Stephen Oney and Joel Brandt. "Codelets: Linking Interactive Documentation and Example Code in the Editor". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: Association for Computing Machinery, 2012, 2697–2706. ISBN: 9781450310154. DOI: 10.1145/2207676.2208664. URL: https://doi.org/10.1145/2207676.2208664.

[181] OpenAI. *ChatGPT*. OpenAI. 2024. URL: https://chat.openai.com/.

[182] Dennis Pagano and Walid Maalej. "How Do Developers Blog? An Exploratory Study". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, 123–132. ISBN: 9781450305747. DOI: 10.1145/1985441.1985461. URL: https://doi.org/10.1145/1985441.1985461.

[183] Dennis Pagano and Walid Maalej. "How do open source communities blog?" In: *Empirical Software Engineering* 18.6 (Dec. 2013), pp. 1090–1124. ISSN: 1573-7616. DOI: 10.1007/s10664-012-9211-2. URL: https://doi.org/10.1007/s10664-012-9211-2.

[184] Soya Park, Amy X. Zhang, and David R. Karger. "Post-literate Programming: Linking Discussion and Code in Software Development Teams". In: *Adjunct Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. UIST '18 Adjunct. Berlin, Germany: Association for Computing Machinery, 2018, 51–53. ISBN: 9781450359498. DOI: 10.1145/3266037.3266098. URL: https://doi.org/10.1145/3266037.3266098.

[185] C. Parnin and C. Gorg. "Building Usage Contexts During Program Comprehension". In: *14th IEEE International Conference on Program Comprehension (ICPC'06)*. 2006, pp. 13–22. DOI: 10.1109/ICPC.2006.14.

[186] Chris Parnin. *Programmer Interrupted*. ninlabs research. 2013. URL: https://blog.ninlabs.com/2013/01/programmer-interrupted/.

[187] Chris Parnin and Robert DeLine. "Evaluating Cues for Resuming Interrupted Programming Tasks". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: Association for Computing Machinery, 2010, 93–102. ISBN: 9781605589299. URL: https://doi.org/10.1145/1753326.1753342.

[188] Chris Parnin, Carsten Görg, and Spencer Rugaber. "CodePad: interactive spaces for maintaining concentration in programming environments". In: *Proceedings of the 5th International Symposium on Software Visualization*. SOFTVIS '10. Salt Lake City, Utah, USA: Association for Computing Machinery, 2010, 15–24. ISBN: 9781450300285. DOI: 10.1145/1879211.1879217. URL: https://doi.org/10.1145/1879211.1879217.

[189] Chris Parnin and Spencer Rugaber. "Resumption strategies for interrupted programming tasks". In: *Software Quality Journal* 19.1 (2011), pp. 5–34.

[190] Chris Parnin, Christoph Treude, and Margaret-Anne Storey. "Blogging developer knowledge: Motivations, challenges, and future directions". In: *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, pp. 211–214. DOI: 10.1109/ICPC.2013.6613850.

[191] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. "Information Needs in Contemporary Code Review". In: *Proc. ACM Hum.-Comput. Interact.* 2.CSCW (2018). DOI: 10.1145/3274404. URL: https://doi.org/10.1145/3274404.

[192] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. "Studying the advancement in debugging practice of professional software developers". In: *Software Quality Journal* 25.1 (Mar. 2017), pp. 83–110. ISSN: 1573-1367. DOI: 10.1007/s11219-015-9294-2. URL: https://doi.org/10.1007/s11219-015-9294-2.

[193] Piling.js. *The Piling.js Docs*. Piling.js. 2021. URL: https://piling.js.org/docs/.

[194] David Piorkowski, Scott Fleming, Christopher Scaffidi, Christopher Bogart, Margaret Burnett, Bonnie John, Rachel Bellamy, and Calvin Swart. "Reactive Information Foraging: An Empirical Investigation of Theory-Based Recommender Systems for Programmers". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '12. Austin, Texas, USA: Association for Computing Machinery, 2012, 1471–1480. ISBN: 9781450310154. DOI:

10.1145/2207676.2208608. URL: https://doi.org/10.1145/2207676.2208608.

[195] David Piorkowski, Scott D. Fleming, Christopher Scaffidi, Margaret Burnett, Irwin Kwan, Austin Z. Henley, Jamie Macbeth, Charles Hill, and Amber Horvath. "To fix or to learn? How production bias affects developers' information foraging during debugging". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 11–20. DOI: 10.1109/ICSM.2015.7332447.

[196] David Piorkowski, Soya Park, April Yi Wang, Dakuo Wang, Michael Muller, and Felix Portnoy. "How AI Developers Overcome Communication Challenges in a Multidisciplinary Team: A Case Study". In: *Proc. ACM Hum.-Comput. Interact.* 5.CSCW1 (2021). DOI: 10.1145/3449205. URL: https://doi.org/10.1145/3449205.

[197] OpenAI Platform. *Overview - OpenAI API*. URL: https://platform.openai.com/.

[198] Ben Popper and David Gibson. *How often do people actually copy and paste from Stack Overflow? Now we know.* URL: https://stackoverflow.blog/2021/12/30/how-often-do-people-actually-copy-and-paste-from-stack-overflow-now-we-know/.

[199] Aniket Potdar and Emad Shihab. "An Exploratory Study on Self-Admitted Technical Debt". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 91–100. DOI: 10.1109/ICSME.2014.31.

[200] James Prather, Brent N. Reeves, Paul Denny, Brett A. Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. ""It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers". In: *ACM Trans. Comput.-Hum. Interact.* 31.1 (2023). ISSN: 1073-0516. DOI: 10.1145/3617367. URL: https://doi.org/10.1145/3617367.

[201] prettier.io. *prettier - npm*. NPM. 2024. URL: https://www.npmjs.com/package/prettier.

[202] Pooja Rani, Mathias Birrer, Sebastiano Panichella, Mohammad Ghafari, and Oscar Nierstrasz. "What do developers discuss about code comments?" In: *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2021, pp. 153–164.

[203] Sarah Rastkar, Gail C Murphy, and Gabriel Murray. "Summarizing software artifacts: a case study of bug reports". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 2010, pp. 505–514.

[204] Eric Rawn, Jingyi Li, Eric Paulos, and Sarah E. Chasins. "Understanding Version Control as Material Interaction with Quickpose". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3581394. URL: https://doi.org/10.1145/3544548.3581394.

[205] Steven P Reiss. "Trace-based debugging". In: *International Workshop on Automated and Algorithmic Debugging*. Springer. 1993, pp. 305–314.

[206] Steven P. Reiss. "Tracking Source Locations". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: Association for Computing Machinery, 2008, 11–20. ISBN: 9781605580791. DOI: 10.1145/1368088.1368091. URL: https://doi.org/10.1145/1368088.1368091.

[207] Peter C. Rigby, Yue Cai Zhu, Samuel M. Donadelli, and Audris Mockus. "Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, 1006–1016. ISBN: 9781450339001. DOI: 10.1145/2884781.2884851. URL: https://doi.org/10.1145/2884781.2884851.

[208] Martin P. Robillard. "Turnover-Induced Knowledge Loss in Practice". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, 1292–1302. ISBN: 9781450385626. DOI: 10.1145/3468264.3473923. URL: https://doi.org/10.1145/3468264.3473923.

[209] Martin P. Robillard. "What Makes APIs Hard to Learn? Answers from Developers". In: *IEEE Software* 26 (6 Oct. 2009), pp. 27–34. DOI: 10.1109/MS.2009.193. URL: https://doi.org/10.1109/MS.2009.193.

[210] Martin P. Robillard and Robert DeLine. "A field study of API learning obstacles". In: *Empirical Software Engineering* 16 (6 2011), pp. 703–732.

[211] M.P. Robillard, W. Coelho, and G.C. Murphy. "How effective developers investigate source code: an exploratory study". In: *IEEE Transactions on Software Engineering* 30.12 (2004), pp. 889–903. DOI: 10.1109/TSE.2004.101.

[212] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. "How do professional developers comprehend software?" In: *ICSE 2012*. New York, NY, USA: ACM, 2012, pp. 632–542. DOI: 10.1109/ICSE.2012.6227188. URL: https://doi.org/10.1109/ICSE.2012.6227188.

[213] Daniel M Russell, Mark J Stefik, Peter Pirolli, and Stuart K Card. "The cost structure of sensemaking". In: *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*. 1993, pp. 269–276.

[214] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. "Modern Code Review: A Case Study at Google". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 181–190. ISBN: 9781450356596. DOI: 10.1145/3183519.3183525. URL: https://doi.org/10.1145/3183519.3183525.

[215] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. "How developers search for code: a case study". In: *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 2015, pp. 191–201.

[216] Bill N Schilit, Gene Golovchinsky, and Morgan N Price. "Beyond paper: supporting active reading with free form digital ink annotations". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 1998, pp. 249–256.

[217] Donghwan Shin, Domenico Bianculli, and Lionel Briand. "Effective Removal of Operational Log Messages: an Application to Model Inference". In: *arXiv preprint arXiv:2004.07194* (2020).

[218] Yusuke Shinyama, Yoshitaka Arahori, and Katsuhiko Gondow. "Analyzing Code Comments to Boost Program Comprehension". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 325–334. DOI: 10.1109/APSEC.2018.00047.

[219] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. "To document or not to document? An exploratory study on developers' motivation to document code". In: *Advanced Information Systems Engineering Workshops: CAiSE 2015 International Workshops, Stockholm, Sweden, June 8-9, 2015, Proceedings 27*. Springer. 2015, pp. 100–106.

[220] Ben Shneiderman. *Software psychology: Human factors in computer and information systems (Winthrop computer systems series)*. Winthrop Publishers, 1980.

[221] Nischal Shrestha, Titus Barik, and Chris Parnin. "Unravel: A Fluent Code Explorer for Data Wrangling". In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. 2021, pp. 198–207.

[222] Stephen Shum and Curtis Cook. "Using Literate Programming to Teach Good Programming Practices". In: *Proceedings of the Twenty-Fifth SIGCSE Symposium on Computer Science Education*. SIGCSE '94. Phoenix, Arizona, USA: Association for Computing Machinery, 1994, 66–70. ISBN: 0897916468. DOI: 10.1145/191029.191059. URL: https://doi.org/10.1145/191029.191059.

[223] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. "Asking and Answering Questions during a Programming Change Task". In: *IEEE Transactions on Software Engineering* 34.4 (2008), pp. 434–451. DOI: 10.1109/TSE.2008.26.

[224] Ananya Singha, Bhavya Chopra, Anirudh Khatry, Sumit Gulwani, Austin Z. Henley, Vu Le, Chris Parnin, Mukul Singh, and Gust Verbruggen. *Semantically Aligned Question and Code Generation for Automated Insight Generation*. 2024. arXiv: 2405.01556 [cs.SE]. URL: https://arxiv.org/abs/2405.01556.

[225] Zéphyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. "Noise in Mylyn interaction traces and its impact on developers and recommendation systems". In: *Empirical Software Engineering* 23.2 (Apr. 2018), pp. 645–692. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9529-x. URL: https://doi.org/10.1007/s10664-017-9529-x.

[226] Jeongju Sohn and Shin Yoo. "FLUCCS: Using Code and Change Metrics to Improve Fault Localization". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: Association for Computing Machinery, 2017, 273–283. ISBN: 9781450350761. DOI: 10.1145/3092703.3092717. URL: https://doi.org/10.1145/3092703.3092717.

[227] Illia Solohubov, Artur Moroz, Mariia Yu Tiahunova, Halyna H Kyrychek, and Stepan Skrupsky. "Accelerating software development with AI: exploring the impact of ChatGPT and GitHub Copilot." In: *CTE*. 2023, pp. 76–86.

[228] Cleidson R. B. de Souza, Gema Rodríguez-Pérez, Manaal Basha, Dongwook Yoon, and Ivan Beschastnikh. "The Fine Balance Between Helping With Your Job And Taking It: AI Code Assistants Come To The Fore". In: *IEEE Software* (2024), pp. 1–6. DOI: 10.1109/MS.2024.3357787.

[229] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. "A Study of the Documentation Essential to Software Maintenance". In: *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting and Designing for Pervasive Information*. SIGDOC '05. Coventry, United Kingdom: Association for Computing Machinery, 2005, 68–75. ISBN: 1595931759. DOI: 10.1145/1085313.1085331. URL: https://doi.org/10.1145/1085313.1085331.

[230] Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. "Foraging among an overabundance of similar variants". In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 2016, pp. 3509–3521.

[231] Captain Stack. *Captain Stack - Code suggestion for VSCode*. URL: https://github.com/hieunc229/copilot-clone.

[232] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David F Redmiles. "A systematic literature review on the barriers faced by newcomers to open source software projects". In: *Information and Software Technology* 59 (2015), pp. 67–85.

[233] Christoph Johann Stettina and Werner Heijstek. "Necessary and Neglected? An Empirical Study of Internal Documentation in Agile Software Development Teams". In: *Proceedings of the 29th ACM International Conference on Design of Communication*. SIGDOC '11. Pisa, Italy: Association for Computing Machinery, 2011, 159–166. ISBN: 9781450309363. DOI: 10.1145/2038476.2038509. URL: https://doi.org/10.1145/2038476.2038509.

[234] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. "TODO or to bug". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008, pp. 251–260. DOI: 10.1145/1368088.1368123.

[235] 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". In: *IEEE Transactions on Software Engineering* 35.4 (2009), pp. 470–483. DOI: 10.1109/TSE.2009.15.

[236] Siddharth Subramanian and Reid Holmes. "Making sense of online code snippets". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. 2013, pp. 85–88. DOI: 10.1109/MSR.2013.6624012.

[237] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. "/* iComment: Bugs or bad comments?*". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007, pp. 145–158.

[238] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies". In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 260–269.

[239] Minaoar Hossain Tanzil, Junaed Younus Khan, and Gias Uddin. "ChatGPT Incorrectness Detection in Software Reviews". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639194. URL: https://doi.org/10.1145/3597503.3639194.

[240] Craig S. Tashman and W. Keith Edwards. "Active Reading and Its Discontents: The Situations, Problems and Ideas of Readers". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, 2927–2936. ISBN: 9781450302289. DOI: 10.1145/1978942.1979376. URL: https://doi.org/10.1145/1978942.1979376.

[241] Grace Taylor and Steven Clarke. "A Tour Through Code: Helping Developers Become Familiar with Unfamiliar Code". In: *Psychology of Programming Interest Group 33rd Annual Workshop*. PPIG 2022, pp. 114–126.

[242] Suresh Thummalapenta and Tao Xie. "Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web". In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, 204–213. ISBN: 9781595938824. DOI: 10.1145/1321631.1321663. URL: https://doi.org/10.1145/1321631.1321663.

[243] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. "What Makes a Good Commit Message?" In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 2389–2401. ISBN: 9781450392211. DOI: 10.1145/3510003.3510205. URL: https://doi.org/10.1145/3510003.3510205.

[244] Christoph Treude and Margaret-Anne Storey. "Work Item Tagging: Communicating Concerns in Collaborative Software Development". In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 19–34. DOI: 10.1109/TSE.2010.91.

[245] Jason Tsay, Laura Dabbish, and James Herbsleb. "Influence of Social and Technical Factors for Evaluating Contribution in GitHub". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, 356–366. ISBN: 9781450327565. DOI: 10.1145/2568225.2568315. URL: https://doi.org/10.1145/2568225.2568315.

[246] Gias Uddin and Martin P. Robillard. "How API Documentation Fails". In: *IEEE Software* 32 (4 Aug. 2015), pp. 68–75. DOI: 10.1109/MS.2014.80. URL: https://doi.org/10.1109/MS.2014.80.

[247] uuid. *uuid - npm*. NPM. 2024. URL: https://www.npmjs.com/package/uuid.

[248] Max G. Van Kleek, Wolfe Styke, m.c. schraefel, and David Karger. "Finders/keepers: a longitudinal study of people managing information scraps in a micro-note tool". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '11. Vancouver, BC, Canada: Association for Computing Machinery, 2011, 2907–2916. ISBN: 9781450302289. DOI: 10.1145/1978942.1979374. URL: https://doi.org/10.1145/1978942.1979374.

[249] Fernanda B. Viégas, Martin Wattenberg, and Kushal Dave. "Studying Cooperation and Conflict between Authors with History Flow Visualizations". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '04. Vienna, Austria: Association for Computing Machinery, 2004, 575–582. ISBN: 1581137028. DOI: 10.1145/985692.985765. URL: https://doi.org/10.1145/985692.985765.

[250] April Yi Wang, Andrew Head, Ashley Zhang, Steve Oney, and Christopher Brooks. "Colaroid: A Literate Programming Approach for Authoring Explorable Multi-Stage Tutorials". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. DOI: 10.1145/3544548.3581525. URL: https://doi.org/10.1145/3544548.3581525.

[251] April Yi Wang, Anant Mittal, Christopher Brooks, and Steve Oney. "How Data Scientists Use Computational Notebooks for Real-Time Collaboration". In: *Proc. ACM Hum.-Comput. Interact.* 3.CSCW (2019). DOI: 10.1145/3359141. URL: https://doi.org/10.1145/3359141.

[252] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. "Callisto: Capturing the "Why" by Connecting Conversations with Computational Narratives". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. CHI '20. Honolulu, HI, USA: Association for Computing Machinery, 2020, 1–13. ISBN: 9781450367080. DOI: 10.1145/3313831.3376740. URL: https://doi.org/10.1145/3313831.3376740.

[253] Dakuo Wang, Judith S. Olson, Jingwen Zhang, Trung Nguyen, and Gary M. Olson. "DocuViz: Visualizing Collaborative Writing". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. Seoul, Republic of Korea: Association for Computing Machinery, 2015, 1865–1874. ISBN: 9781450331456. DOI: 10.1145/2702123.2702517. URL: https://doi.org/10.1145/2702123.2702517.

[254] Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. "Significant Productivity Gains through Programming with Large Language Models". In: *Proc. ACM Hum.-Comput. Interact.* 8.EICS (2024). DOI: 10.1145/3661145. URL: https://doi.org/10.1145/3661145.

[255] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. "A large-scale empirical study on code-comment inconsistencies". In: *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE. 2019, pp. 53–64.

[256] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. "Novice Reflections on Debugging". In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, 73–79. ISBN: 9781450380621. DOI: 10.1145/3408877.3432374. URL: https://doi.org/10.1145/3408877.3432374.

[257] 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*. CHI '16. San Jose, California, USA: Association for Computing Machinery, 2016, 3522–3532. ISBN: 9781450333627. DOI: 10.1145/2858036.2858442. URL: https://doi.org/10.1145/2858036.2858442.

[258] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. "How do developers utilize source code from stack overflow?" In: *Empirical Software Engineering* 24 (2019), pp. 637–673.

[259] Xin Xia, Lingfeng Bao, David Lo, Pavneet Singh Kochhar, Ahmed E. Hassan, and Zhenchang Xing. "What do developers search for on the web?" In: *Empirical Software Engineering* 22.6 (Dec. 2017), pp. 3149–3185. ISSN: 1573-7616. DOI: 10.1007/s10664-017-9514-4. URL: https://doi.org/10.1007/s10664-017-9514-4.

[260] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. "Detecting large-scale system problems by mining console logs". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, 117–132. ISBN: 9781605587523. DOI: 10.1145/1629575.1629587. URL: https://doi.org/10.1145/1629575.1629587.

[261] Wei Xu, Ling Huang, and Michael Jordan. "Experience mining Google's production console logs". In: *Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML 10)*. 2010.

[262] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. "Understanding Feature Evolution in a Family of Product Variants". In: *2010 17th Working Conference on Reverse Engineering*. 2010, pp. 109–118. DOI: 10.1109/WCRE.2010.20.

[263] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. 2023. arXiv: 2304.10778 [cs.SE]. URL: https://arxiv.org/abs/2304.10778.

[264] Nur Yildirim, Changhoon Oh, Deniz Sayar, Kayla Brand, Supritha Challa, Violet Turri, Nina Crosby Walton, Anna Elise Wong, Jodi Forlizzi, James McCann, and John Zimmerman. "Creating Design Resources to Scaffold the Ideation of AI Concepts". In: *Proceedings of the 2023 ACM Designing Interactive Systems Conference*. DIS '23. Pittsburgh, PA, USA: Association for Computing Machinery, 2023, 2326–2346. ISBN: 9781450398930. DOI: 10.1145/3563657.3596058. URL: https://doi.org/10.1145/3563657.3596058.

[265] Young Seok Yoon and Brad A. Myers. "A longitudinal study of programmers' backtracking". In: *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2014, pp. 101–108. DOI: 10.1109/VLHCC.2014.6883030.

[266] YoungSeok Yoon and Brad A. Myers. "Semantic zooming of code change history". In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2015, pp. 95–99. DOI: 10.1109/VLHCC.2015.7357203.

[267] YoungSeok Yoon and Brad A. Myers. "Supporting Selective Undo in a Code Editor". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 223–233. DOI: 10.1109/ICSE.2015.43.

[268] Xiao Yu, Lei Liu, Xing Hu, Jacky Wai Keung, Jin Liu, and Xin Xia. *Fight Fire with Fire: How Much Can We Trust ChatGPT on Source Code-Related Tasks?* 2024. arXiv: 2405.12641 [cs.SE]. URL: https://arxiv.org/abs/2405.12641.

[269] Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. "Example Overflow: Using social media for code recommendation". In: *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. 2012, pp. 38–42. DOI: 10.1109/RSSE.2012.6233407.

[270] Iyad Zayour and Timothy C Lethbridge. "A cognitive and user centric based approach for reverse engineering tool design". In: *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*. 2000, p. 16.

[271] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2009.

[272] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, Ying Zou, and Ahmed E. Hassan. "An Empirical Study of Obsolete Answers on Stack Overflow". In: *IEEE Transactions on Software Engineering* 47.4 (2021), pp. 850–862. DOI: 10.1109/TSE.2019.2906315.

[273] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. "Analyzing and Supporting Adaptation of Online Code Examples". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 316–327. DOI: 10.1109/ICSE.2019.00046.

[274] J. Zhi, V. Garousi-Yusifoglu, B. Sun, G. Garousi, S. Shahnewaz, and G. Ruhe. "Cost benefits and quality of software development documentation: a systematic mapping". In: *Journal of Systems and Software* 99 (2015).

[275] Sacha Zyto, David Karger, Mark Ackerman, and Sanjoy Mahajan. "Successful classroom deployment of a social document annotation system". In: *CHI 2012*. New York, NY, USA: ACM, 2012, pp. 1883–1892. DOI: 10.1145/2207676.2208326. URL: https://doi.org/10.1145/2207676.2208326.