Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code

Amber Horvath
ahorvath@cs.cmu.edu
Human-Computer Interaction
Institute, Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Andrew Macvean
amacvean@google.com
Google
Seattle, Washington, USA

Brad A. Myers
bam@cs.cmu.edu
Human-Computer Interaction
Institute, Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

ABSTRACT

Modern software engineering is in a state of flux. With more development utilizing AI code generation tools and the continued reliance on online programming resources, understanding code and the original intent behind it is becoming more important than it ever has been. To this end, we have developed the "Meta-Manager", a Visual Studio Code extension, with a supplementary browser extension, that automatically collects and organizes changes made to code while keeping track of the provenance of each part of the code, including code that has been copy-pasted from popular programming resources online. These sources and subsequent changes are represented in the editor and may be explored using searching and filtering mechanisms to help developers answer historically hard-to-answer questions about code, its provenance, and its design rationale. In our evaluation of Meta-Manager, we found developers were successfully able to use it to answer otherwise unanswerable questions about an unfamiliar code base.

1 INTRODUCTION

Software engineering is a discipline about information management. When writing code, software engineers are typically managing many different tasks and questions [60]. While higher level goals, such as "implement this feature", are typically captured through Git commit messages or pull request details, lower level design details, such as why a certain variable has a specific value and whether this value was anticipated [37], are less often recorded due to the high cost incurred in externalizing these thoughts during the implementation process [28, 49]. Nearly all developers are considering such issues continually as they make implementation decisions [46]. Despite the prevalence of these decisions, these small rationale choices can become completely lost to time, which can be problematic for later developers who are trying to maintain or contribute to the code base [5, 6, 65], with one study reporting that answering such questions was considered "exhausting" by participants, yet none of the participants recorded their own rationale [52].

Research shows that programmers spend up to 50% of their time trying to understand code written by themselves or others in order to do maintenance [37, 52, 56]. In order for this process to be successful, these later developers need to not only understand the code, but must also understand the constraints, requirements, and other reasons *why* the code is the way that it is, which is called the *design rationale*. We are focusing on supporting these *later* developers so they can better understand the code written by the code *author*. Note that sometimes these are the same person, but even then, developers often forget the design rationale of their own code [52].

Prior research has found that questions about design rationale are one of the most common and significant blockers for developers when trying to understand code authored by later developers [37, 43, 52]. Today's strategies for trying to answer these design rationale questions include trying to recreate the history of the code through asking other developers on the team or foraging through version control logs and associated documents [37, 43]. This process is both time-consuming and prone to failure if the developer who could answer the question either is no longer available or has forgotten the answer, and developers can be reticent to ask teammates for fear of interrupting them [52].

One way of keeping track of rationale and historical information is through capturing more context about the code and its development. In an example taken from Ko et. al. [37], it may not be clear initially why a variable has a specific value, but, with the added context that the developer logged this particular value and then removed that log, a later developer can reason the author was aware of the seemingly-erroneous value. Research has long shown that developers have wanted these kinds of insights into the code history in order to help them understand these lower-level implementation decisions to avoid violating constraints and requirements that are pervasive in code but are rarely documented [41, 43].

A key challenge in capturing more information is scale — a developer typically makes hundreds of edits to their code during a working day. A study of Visual Studio usage found that developers spent approximately 28% of their 7-hour workday actively editing code [1] — over many developers and many days, the number of edits to some code can balloon into an unreasonable amount. Given so many edits, how does one find and present information that can help answer developers' implicit questions regarding code?

In this work, we explore automatically collecting, organizing, and utilizing code history and development information to answer developers' historically "hard-to-answer" questions about code. We focus on questions related to code history and design rationale, given that these questions are significant blockers for later developers maintaining code [41, 43, 52].

We attempt to address this challenge of reconstructing provenance information (the origin of the code) to help developers answer their questions through our tool, Meta-Manager, a Visual Studio Code [55] extension for TypeScript [54], with a supplementary Google Chrome extension, which together capture code provenance through an event-driven lens. Meta-Manager is designed to support answering questions through not only collecting provenance information, but supporting challenges of *scale* with a visualization for summarizing histories and supporting *navigation* with interactive mechanisms for traversing through code histories. Specifically, Meta-Manager addresses the following challenges to

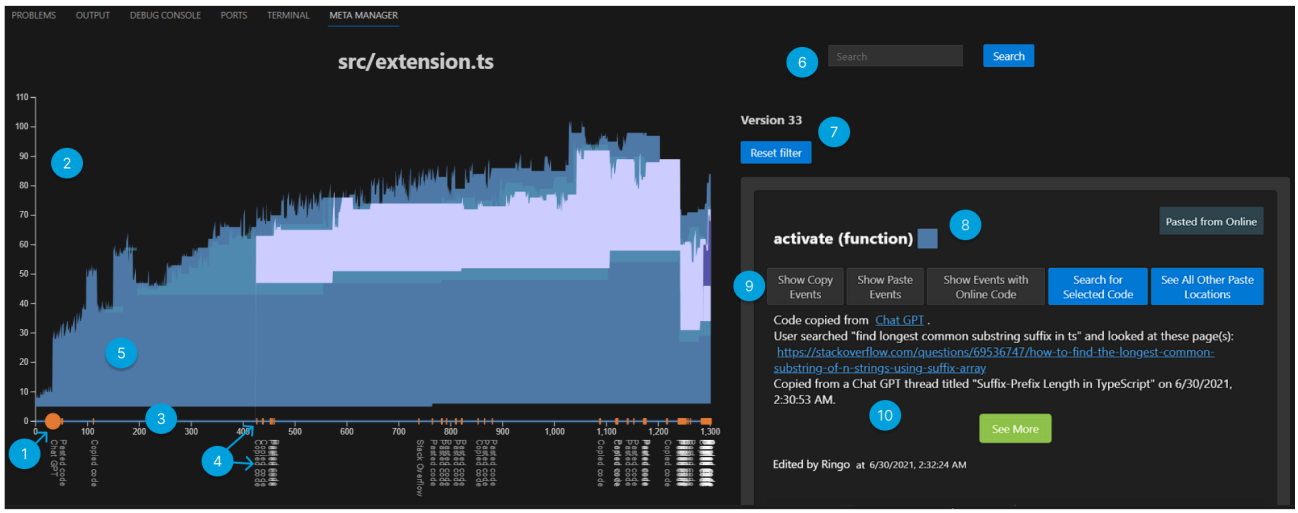

Figure 1: Meta-Manager as it appears within Visual Studio Code: the pane appears in the bottom area of the editor, with the left area displaying a visualization of the history of the code file over time, while the right area displays information about a particular code version. (1) is the scrubber, which the developer can use to move between code versions; (2) is the y-axis, which denotes the lines of code within the file; (3) is the x-axis which represents all the editing "events" on the code, with 0 being the start of the file and the right end being "now" (here there have been over 1,300 edit events); (4) is an identified event (in this case, "Copied Code") which appears along the timeline as an orange tick and label; (5) is the range of code lines as they changed over time, with the color corresponding to the particular part of the code (in this case, blue for the `activate` function); (6) is the search bar, which will search across all the code versions in the current file for the search text; (7) is the number of the version, and includes a "Reset filter" button which will set the events along the x-axis back to their default state; (8) is the code box for this particular code version – in this case, the `activate` function at version 33; (9) is the row of buttons for actions the user can perform on a code version — the 3 leftmost gray buttons act as filters for the events, while the 2 blue buttons will search for either the user's selected code within the code version ("Search for Selected Code") or paste events related to the current copied code event; (10) is the description of the code version – in this case, since the user pasted code from ChatGPT, the text describes the first search given to ChatGPT for the session from where the code was copied, and provides a button for viewing more information about the ChatGPT thread.

make question-answering about code provenance and rationale possible:

- ***Unwritten design rationale.*** Developers often do not want to write down their lower-level implementation decisions because they either do not believe the decisions to be important [49] or because they are in a flow state, where pausing to externalize their thoughts is too burdensome [51]. Meta-Manager makes this externalization unnecessary in many cases through automatically capturing information about the developers' activities such that each code version contains meta-information that a developer would not normally write down. This includes visited web pages, copy-paste sources, search queries, and copies of web pages where the pasted code came from. In other complex sensemaking domains, these information trails helped later people better understand the original user's intent and rationale behind their decision [35, 50] – we hypothesize that Meta-Manager, with its scale and navigational support, will make this reasoning possible for answering provenance and rationale questions about code. Further, we hypothesize that since more code is being generated by AI tools (developers in a recent study report around 31 percent or more of their new code comes from AI tools [47]) or pasted from online, saving the queries used to find or generate that code will be increasingly informative.
- ***Scale.*** Given the large amount of edits developers make to code [1], the varying size of code bases, and amount of potential noise in a data set comprised of such edits [71], Meta-Manager collapses and prioritizes the provenance data through multiple methods to assist later developers in their comprehension of code history.
  - *Visualization and data organization.* Meta-Manager collapses edit events in a visualized "stream" across time, with each stream corresponding to a particular code block, such that developers can, with a glance, glean when blocks of code are introduced or removed, moved, and so on. In this way, the visualization may itself answer some developer questions about the code through summarizing its

history. Our visualization, in conjunction with its highlighting of *important editing events*, is a novel interaction for reasoning about code history and design rationale.
  - *Significant editing events.* Given our hypotheses around what editing events may answer the historically-challenging questions (Section 3.1), Meta-Manager specifically tracks when and how certain editing events occur in the code's history. These edits include copy and paste events (including copy-pasted code from online), block commenting in and out code, and, given a specific code snippet, when that snippet has been edited. Meta-Manager introduces its own prioritization of code versions to reduce the search space for later users.
  - *Filter and zoom.* Meta-Manager supports further reducing the search space of code history through filtering the visualization to only show edits of specific types and zooming into parts of the visualization.
- ***Navigation.*** To *answer* a question using Meta-Manager, a developer must find the information in the code's history that is relevant to their question. This information may come in various forms, such as a code version, editing event, or web page. Meta-Manager is designed to support intuitive mechanisms for navigating through a potentially large search space of code and code edits to find these information patches[1].
  - *Annotated timeline.* Edit events of interest, along with edits involving searched-upon code, are marked as annotations on the visualization's timeline, so that a user can click on the annotation and navigate to the code version that contains, e.g., some code pasted from Stack Overflow.
  - *Scrubbing.* Meta-Manager adopts the interaction technique of moving quickly through time with brief previews of the underlying content with a scrubber, to provide a quick view about how some code changes over time – another common "hard-to-answer" question.
  - *Search by content and by time.* To support developers as they refine their query and want to limit their search space or when they find some code of interest in the current editor, Meta-Manager supports *searching* across the code history either by whether a version contains a search term or by how a line/fragment/construct/etc. changes across the version history.

We hypothesize that, through proper tooling that collects meta-information related to the code author's implementation session while combating issues of scale and supporting various navigation methods, later developers can answer their questions about code history and design rationale when understanding an unfamiliar code base.

In order to evaluate whether developers are able to answer otherwise unanswerable questions about code with Meta-Manager, we ran an exploratory user study with 7 people. In order to make the task realistic (meaning there are many edits, with only a small subset that are "useful"), we recreated a real existing code base [74] as though a developer had been using Meta-Manager during the code base's entire development. We then added in many reasonable, yet simulated, edits in order to create a much larger code history for participants to navigate through. Participants used Meta-Manager to try to answer provenance and design rationale questions about the code history. We found that developers were able to successfully use Meta-Manager to answer the questions about the history, and the participants confirmed that the code and questions were realistic.

Our work contributes the following:

- Identifying that developer's usage of AI code-generation systems and the meta-information created during their usage, such as chat threads, can be used by later developers to reason about design intent of implementation choices.
- Identifying that types of code edits, including copy-pasted code and commented-out code, can help developers answer questions about code history and provenance.
- A system, Meta-Manager, that collects code editing and provenance data for answering historically "hard-to-answer" questions about code history and design rationale.
- Features integrated into Meta-Manager that organize, prioritize, and collapse information using a visualization to help combat issues of code provenance scale, and interactive mechanisms which support navigation, so that developers can answer otherwise impossible-to-answer questions about a code history of a realistic size. Meta-Manager and its features provides the following novel contributions:
  - **Combining code history visualizations with extracted significant editing events** to reduce the search space for readers of unfamiliar code in finding code versions of interest.
  - **Tracking, adding meta-information and versioning code copied and pasted from online** in order to help later readers of code reason about the provenance and rationale behind code changes.
- A user study of Meta-Manager that demonstrates:
  - Meta-Manager can be used to answer common and historically difficult questions about code, including questions about the rationale behind its design.
  - Meta-information related to usage of AI-generated code may be used to reason about the rationale behind code implementation choices.
  - Developers' information foraging for question-answering can be supported with Meta-Manager's features, with participants utilizing multiple different methods of foraging through the code histories to answer their questions – however, the design space of clarifying the relationship between code that is differentiated across both space (e.g., files) and time (e.g., versions) has room for improvement.

2 RELATED WORK

Our work builds upon human-centered software engineering research focused around how developers understand code, especially with respect to code history, along with systems that have attempted to assist in that sensemaking process.

[1]"Information patch" is a term from Information Foraging Theory [62] used to describe an information source that includes "smells" that either attract or deter users from engaging with the source, given perceived relevance.

2.1 Code Comprehension

Researchers in HCI and Software Engineering have extensively studied the practice of understanding code [7, 8, 11, 15, 38–40, 42, 52, 66, 68]. Understanding unfamiliar code is known to be cognitively demanding, as developers are attempting to keep track of many different types of information, including their current working task context [36, 41, 60, 70], and their understanding and knowledge of the code [43, 52, 70]. Among these studies, there has been a focus on what are commonly called "hard-to-answer" questions about code [18, 37, 44, 52, 70]. Typically, these questions relate to the history and design rationale of the code and can significantly block developers from progressing on a task at any point in the software development life cycle, whether that be in the context of taking over a code base [65], maintaining and collaborating on a project [37, 52], or joining a new project [6, 30]. Considering this known challenge of understanding code, prior work has explored tooling solutions to assist in this process, including generating within-IDE code descriptions using natural language processing [57], directly supporting developer sensemaking activities such as code commenting and annotating [22, 26, 28, 75, 77], using documentation [27], and supporting easier navigation of code [7, 9], sometimes through sharing traces of navigation data from other developers within the code base [11]. Our work extends this research through explicitly attempting to address some of these comprehension problems through a code-history exploration tool that extracts significant events worthy of investigation, without requiring extra work at design time, unlike related tools [26, 28, 75, 77] which require the developer to explicitly externalize their thoughts as notes [26, 28, 61] or code comments [75, 77] in order for the tools to provide value.

Of particular relevance to Meta-Manager are tools that utilize developers' natural information-seeking behaviors to assist in code comprehension. Mylar (later called "Mylyn") and its subsequent iterations utilize developer's navigation patterns, and edit behaviors to create a degree-of-knowledge model to recommend code entities given a developer's current task [19, 31] – Meta-Manager similarly leverages developer actions for prioritizing types of information. Further, in [19], code patches extracted by Mylyn were determined by experts to not be relevant for newcomers – in contrast, our system's navigation mechanisms were used by newcomers to a code base in order to reason about design rationale which suggests some of our novel editing patterns across history may be useful when used in conjunction with their system's degree-of-knowledge model in suggesting code patches. Codetrail [21] utilizes a shared information channel between the text editor and web browser to support and automate tasks such as using documentation. One feature of Codetrail that is particularly relevant to Meta-Manager is Codetrail's identification of code copied from the web – when code is copied from online, Codetrail automatically creates a "bookmark" pointing to the web page in which the code snippet was copied from. Notably, Codetrail does not version the code when this occurs, version the website in case that code snippet no longer exists, or identify within the source file where the pasted code ended up. Without this organization of information, participants in their user study said the feature was difficult to use. Meta-Manager improves upon Codetrail's model of shared information between the editor and web browser through collecting more information, including the developer's search query and visited web page(s), the content of the web page at the time of copy and the location where the copied code originated from, then versioning this information, such that the original context is retained and can be reasoned about.

2.2 Code and Document History

Some prior research tools, along with some commercial tools, are designed to support exploration of code history. Code history visualizations, specifically, have been extensively studied. While almost all code history visualizations reserve the x-axis for representing time, 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 [81]. 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. Other visualizations reserve the y-axis and data points along the visualization for edits that happened to some code at a particular time [83–85]. Our visualization differs by *combining* these two approaches through using the stream visualization approach to showcase blocks of code, while *annotating* the timeline with particular edit events.

Visualizing code history is similar to visualizing other types of document histories. In a study of Wikipedia contribution patterns, the researchers created the "HistoryFlow" visualization, in which the x-axis corresponds to either edits or time depending upon user selection, while the y-axis represents the length of the article – contributions are color-coded by which contributor made the edit [78]. DocuViz [80] draws from "HistoryFlow" and similarly shows edits by time and authorship – the main difference is DocuViz is designed for asynchronous editing platforms in which multiple edits may be happening concurrently. This approach is similar to the stream approach used by [81] and our system. Our approach differs by coloring the timeline streams by code structure as opposed to by user who made the edit – we do this considering our question-answering support is focused primarily around questions of design rationale as opposed to authorship.

Some other code history visualizations do away with the timeline-style presentation. Quickpose [64] uses a node-based graph structure in which each node is a version that may be annotated, moved around, and executed, allowing for an interactive approach to version management. Another approach for visualizing code history is to present a visualization that mirrors the presentation of the source code, yet imbues meta-information about the history of the code. Both Seesoft [14] and Augur [20] visualize each line of code with a color representing how recently it has been edited, along with colors denoting who made the edit and what type of code structure the line of code is a part of (e.g., method). This visualization is particularly effective for answering "when, how, and by whom was this code most recently changed" but does not serve to answer some of the other questions related to code provenance and rationale that we are interested in. Nonetheless, this more fine-grained information is present in Meta-Manager as part of the code box for a particular version, while the part of the code structure is denoted using the stream visualization. In this way, Meta-Manager attempts to combine more forms of code history-related meta-information in a single visualization and user interface, through supporting both

macro-level insights with the visualization and annotated timeline and micro-level details with a code details view.

Other code history systems focus less on visualizing history and more on utilizing the history to support development tasks. Deep Intellisense [25] and Hipikat [10] are code project "memory" systems that serve code patch recommendations given a user's currently-assigned bug. Parnin introduces the concept of "code narratives" and instantiates it with a suite of tools that capture code versions on save and summarize the changes, with a separate tool for collecting visited programming web pages [59]. Code history information is particularly important in the context of data science, where recreating analyses that lead to specific outcomes is necessary – this need leads to specific tools for exploring code versions in computational notebooks [24, 32, 33, 69, 79]. Traditionally, most software engineering teams utilize version control systems (VCS), such as GitHub [53], for managing their code and the subsequent deployments of those versions. These systems typically operate at the file level, which makes finding fine-grained versions difficult, if not impossible, and these systems do not extract additional meta-information, such as where code originated from or what the original code author was attempting to achieve. Further, these versions are typically abstracted away from the original development context, which makes finding a version nearly impossible to locate amongst many similar versions on a website [73]. By losing the context of the original editing workspace, it becomes more difficult for the developer to formulate a useful query to locate their code events of interest [67]. Meta-Manager improves upon this model through capturing more contextual information about the editing workspace, versioning this meta-information, and supporting fine-grained explorations of code history through search.

Our system builds upon prior code-versioning works through introducing a lightweight mechanism that collects and saves significantly more information without requiring overhead from the code author, and allows for querying within the code editor at a fine-grained level to find particularly interesting events, which cuts down on the required work a later developer must do when understanding code history. Notably, nearly all of the prior tools and visualizations do not explore to what extent the history data and/or visualization can actually help developers *learn* something about the code, and few tools provide features for interacting with the history. Meta-Manager builds upon this code history work through providing mechanisms for navigating with interactive mechanisms including search and a scrubbing mechanism to help find information that is relevant.

3 OVERVIEW OF META-MANAGER

In order to capture code provenance and rationale information at scale in an investigable manner, we developed Meta-Manager. We begin our discussion by delineating what questions we believe Meta-Manager is able to answer, then show how a new developer to a software team may use Meta-Manager to answer these questions. We then discuss how each feature in Meta-Manager instantiates our design goals and addresses these significant questions developers have about code.

3.1 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 [10, 13, 18, 30, 37, 43, 52, 65, 70]. Working with unfamiliar code happens in many different contexts, including maintaining a code base that has been edited by many engineers across time [18, 37, 43, 52, 70], joining a new project [10, 30], adopting a code base from a departed coworker [65], or using a new software library [13]. In reviewing the literature, we were particularly interested in questions about the *rationale* behind code's design, given that questions of design rationale were the most common question in a study of professional software engineers [43] and there are minimal tooling options to support answering these questions, despite their ubiquity [52]. Through our literature review, we identified the following questions as related to code rationale and provenance, and as potentially answerable through supporting developer's sensemaking of code history:

- ***History*: How has this code changed over time?** [13, 43, 52, 70] 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 [72], finding when some code was last used in service of understanding how a feature changed over time [82], getting "up-to-speed" on a new code base[30], or finding a snippet of code that was edited repeatedly to understand where the original developer had issues [43, 46, 72]. 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 [32].
- ***Rationale*: Why was this code written this way?** [30, 37, 43, 52, 65, 70] 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 given developers reticence to pause and write [49] or because they believe it to be unimportant[52]), 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, which are often too difficult to forage through [73]. Developers have stated that attempting to answer these questions are "exhausting" given the lack of tooling support and reticence to ask co-workers [52], 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?** [10, 18, 43, 70] 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 (often called the "working set" [7, 9]), 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 [43].

- ***Provenance*: Where did this code come from?** [18, 70] 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 [63]. Given this ubiquity of online code and developers reliance upon it, researchers have investigated the trustworthiness of code that is sourced from online resources [4], ability to be adapted to a developer's own working context [86], and correctness of the code in terms of API usage, syntax, and so on [76]. With the rise in large language models (LLMs) for code generation, research is beginning to focus on the quality of AI-generated code as well [34, 47, 48]. 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 [3] 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 [23].

3.2 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.

3.3 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 3.1).

3.3.1 *[D1] Automatic Code History and Provenance Data.*

We developed Meta-Managerto support better navigation and sensemaking of code history through a scalable and visualized history view (see Figure 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 developer's mental models of code [25] 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, given 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 1-5) and, given the location of the scrubber (Figure 1-1) along the timeline, a code box version (Figure 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 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 2, this particular version of the method `getConfig` had a paste event, where the user pasted in the code `const searchQueryTerm = '//findme';` on line 6. The version adds additional information such as where that copy came from (in this case, the file "src-extension.ts") and functionality, such as seeing the original copied code (Figure 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[2]. 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 [35, 50] and our observations.

Similarly, if the user makes a programming-related Google search[3] prior to visiting these websites, their initial query and visited web pages will be included with the meta-information about the pasted code (Figure 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 [23]. By having this information versioned, we give the developer the tools to reason about this rationale across time.

3.3.2 ***[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 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 ChatGPT's significant contribution at version. In this way, the visualization itself can serve to answer some questions about the code's evolution on its own. The visualization also contextualizes the annotated timeline of events (see section 3.3.3-[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 3.3.3-[F7]). Edit events of interest include copy-paste events, both from online 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 1-10 which shows where the code came from, what the user was doing online, who performed the edit, and when

[2]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.

[3]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 [50].

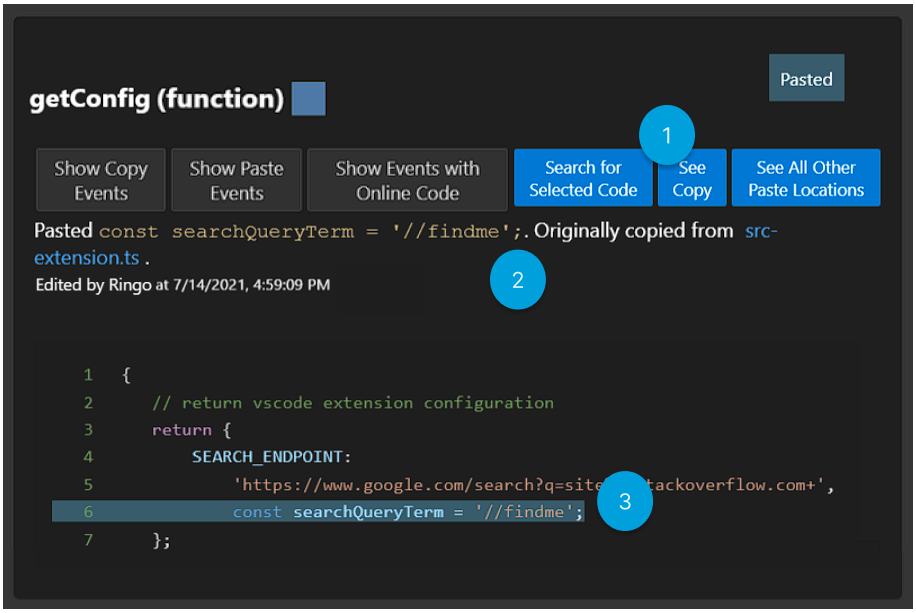

Figure 2: How the code box looks when expanded to show a code version – in this case, a "Paste" event version. (1) shows the buttons specific to a "Paste" code version, including the"See Copy" button which will navigate the user to the corresponding copy event on the timeline (if the copy happened in 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.

it occurred. This meta-information will change given the type of edit (see Figure 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 [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 amount 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 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.

3.3.3 ***[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 may be imbued with 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 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 is 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 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 3.3.2-[F4]) by annotating these events along the visualization's timeline (see Figure 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 3.3.3-[F6]). A large barrier to making sense of code history is the challenge of searching through large histories [73]. 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 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, alignig 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.

3.4 Implementation

The Meta-Manager, both the Visual Studio Code editor extension and supplementary browser extension, utilize TypeScript for the logic and React [16] (with D3.js [58] for the chart in the Visual Studio Code extension) for the front end. FireStore [12] 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 relies on the TypeScript abstract syntax tree (AST) to match parsed blocks to stored code entities in the database. Matching utilizes text-matching via a "bag of words" approach (explained in [81]), prioritizing known block relationships, Git commits, and line differences. Each node manages its version history through a "change buffer" that monitors changes to detect our edits of interest. Despite copy events not altering the code, the system identifies which node experienced the copy, establishing a connection with the pasted node.

4 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. This study design of asking questions about an unfamiliar code base, in which the code base is artificially designed to include qualities derived from prior work as significant challenges in comprehension, is consistent with related prior work [11, 28, 33].

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 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.

4.1 Method

4.1.1 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 prior to running the study. 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 would be trivial. The artificial code base is based upon a real code base [74] 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.

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

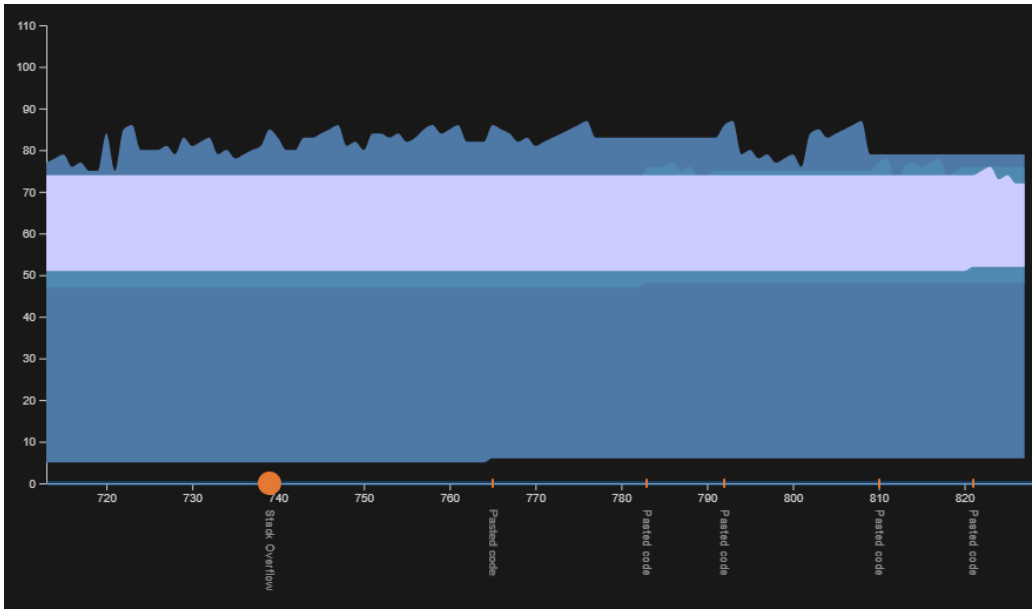

Figure 3: A zoomed-in portion of the timeline shown in Figure 1. This zoomed-in portion shows around 120 edits between Version 710 and Version 830, with the scrubber set around Version 740, when a user pasted code from Stack Overflow.

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 code block. The first author intentionally did not write "perfect" code that matched what was in each commit, to account for the 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 [29]. We also added realistic copy-pastes from Stack Overflow and a few from ChatGPT since these will be increasingly important, with these events occurring less frequently than within-editor copy-pastes. We followed an approach consistent with prior work that was similarly exploring code history by intentionally including code from a variety of online sources so as to not bias the code base by only using code sourced from one individual representing one implementation style [33]. To further validate the realism of the code, we followed the same approach as [28] 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]. We generated a code base consisting of 5,661 edits in 1,328 lines of code across 10 files and 28 different code blocks.

4.1.2 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 understand the context of the code base, but none of the code history task questions related to anything in that particular file.

4.1.3 Task. For the main task, 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 3.1) given prior literature 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 1 lists each question, along with the steps a participant could do in Meta-Manager to answer the question. The complete set of study materials, and a video showing how to answer the questions, is included in the supplemental material[5]. The solution in the table represents the most efficient way to answer a question, but each

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

[5]The complete code for Meta-Manager will be released with the non-anonymous version of the paper.

question can be answered using other methods. 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, questions 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.

4.1.4 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.

4.2 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 self-reported competency with TypeScript was 3. All study sessions were completed and recorded using Zoom and participants used Zoom to take 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 referred to as P1 through P7.

4.3 Quantitative Results

Participants, on average, were able to correctly answer their questions 85.7% of the time, 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 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 1 as the intended solution path. Notably, these questions correspond to 3 of the 4 types of information needs discussed in Section 3.1, 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 relationship and rationale information needs 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 supports our hypothesis, along with participants overall success on Q1, that reasoning about rationale can be done by using information traces from AI code-generation tools and related web activity.

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 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 supporting this information need. 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 adds more evidence that supporting answering these questions is useful.

4.4 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.

4.4.1 [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

Question in Task	Info. Need (Sec. 3.1)	Solution	Outcome	Avg. Time Spent
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 1: Each question that was asked during the task, along with what information need from prior literature it corresponds to, the steps 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 "hard-to-answer" question, 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.

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.

4.4.2 [D2] Make Information Scalable. In terms of managing the sheer scale of the version space participants were operating in, the

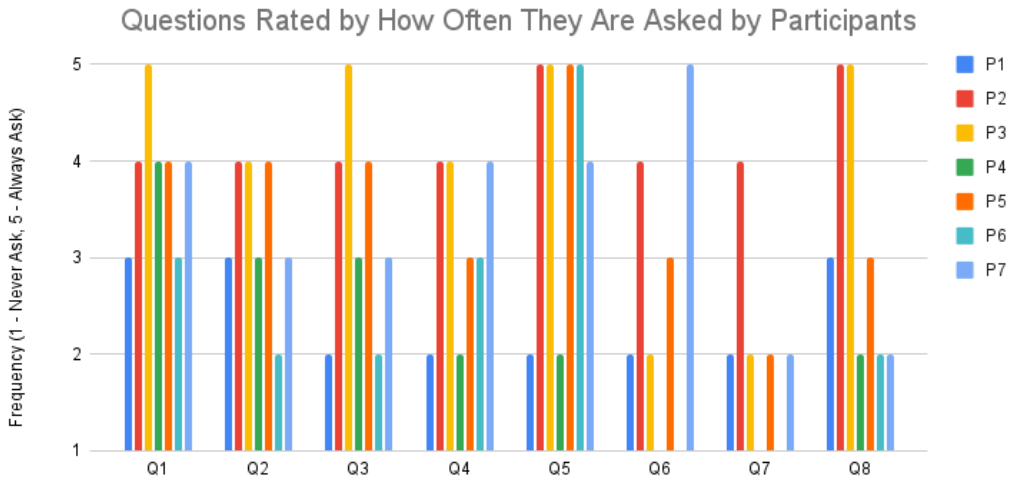

Figure 4: Each question scored by participants in terms of how often they encounter similar questions in their own programming experiences.

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 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 [2] to gain an understanding of part of the information landscape is consistent with behaviors exhibited in other information foraging studies [73], suggesting Meta-Manager's feature set supports these processes of handling a large information-space.

4.4.3 [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 [33]. 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, in which 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.

Navigating Between and Across Files, Spatially and Temporally. Some participants faced confusion when attempting to not only reason about history and how it related to their current file, but when also reasoning about code versions *across* files. 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 [64].

5 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 [73] 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, [73] 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".

In summarizing our findings and their implications, we find support for the claims that (a) 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 *why*; (b) capturing information traces during the AI code-generation process ca be used to support this reasoning about why; and (c), more generally, some information produced as a by-product of authoring code can be mapped to later developers information needs. Previous systems have captured some of this meta-information, such as Mylyn [18], typically captured this information to support code *authoring* tasks, such as localizing code patches to change, and were less concerned with questions of code comprehension by later developers. Other systems, such as the wide array of code visualization systems [81, 83], 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.

6 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 all questions receiving at least a "rarely asked" or higher 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 and search through better allows us to 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 the types of questions it is designed to answer. Previous work, such as [52], 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 through investigating to what extent the breadth of developers' information-seeking behaviors are supported with Meta-Manager. Our study, instead, focuses on to what extent Meta-Manager and its features do work for answering the questions we know from prior literature are difficult to answer.

7 FUTURE WORK

Our lab study provided some evidence that Meta-Manager helps developers answer what have historically been 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 allowing users to define their own "events" 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 [75, 77] – extracting project-specific tagged events as timeline events may also help developers with navigating between code versions.

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 [45], or for saving a code version that a developer may be considering reverting back to [32, 33]. 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 [17] that multiple people through time often need to repeat previous people's work.

8 CONCLUSION

Understanding code and developing tools for assisting in that process is becoming more important than ever. We present Meta-Manager as a tool designed to help with answering historically challenging questions related to code design and history that are unanswerable without the provenance information that our tool automatically collects, including AI code-generation meta-information, the first of its type to show the utility of that information for reasoning about design. The success of Meta-Manager in allowing developers in our study to answer 85.7% of the otherwise un-answerable questions suggests that such approaches should be further investigated to support future developers. As AI permeates more creative work beyond programming, including text and picture generation, Meta-Manager points to a way to keep track of more context about what happened, which can make future systems more maintainable and understandable.

REFERENCES

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

[2] Jeff Baker, Donald Jones, and Jim Burkman. 2009. Using visual representations of data to enhance sensemaking in data exploration tasks. *Journal of the Association for Information Systems* 10, 7 (2009), 2.

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

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

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

[6] Andrew Begel and Beth Simon. 2008. Novice Software Developers, All over Again. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1404520.1404522

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

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

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

[10] D. Cubranic, G.C. Murphy, J. Singer, and K.S. Booth. 2005. Hipikat: a project memory for software development. *IEEE Transactions on Software Engineering* 31, 6 (2005), 446–465. https://doi.org/10.1109/TSE.2005.71

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

[12] Google Developers. 2022. *Cloud Firestore: Store and sync app data at global scale*. Google LLC. Retrieved March 27, 2022 from https://firebase.google.com/products/firestore

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

[14] S.C. Eick, J.L. Steffen, and E.E. Sumner. 1992. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering* 18, 11 (1992), 957–968. https://doi.org/10.1109/32.177365

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

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

[17] Kristie Fisher, Scott Counts, and Aniket Kittur. 2012. Distributed Sensemaking: Improving Sensemaking by Leveraging the Efforts of Previous Users. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 247–256. https://doi.org/10.1145/2207676.2207711

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

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

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

[21] Max Goldman and Robert C. Miller. 2009. Codetrail: Connecting source code and web resources. *Journal of Visual Languages Computing* 20, 4 (2009), 223–235. https://doi.org/10.1016/j.jvlc.2009.04.003 Special Issue on Best Papers from VL/HCC2008.

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

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

[24] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3290605.3300500

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

[26] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A. Myers. 2022. Understanding

How Programmers Can Use Annotations on Documentation. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA) *(CHI '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3491102.3502095

[27] 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)*. 109–114. https://doi.org/10.1109/VL-HCC57772.2023.00020

[28] Amber Horvath, Brad Myers, Andrew Macvean, and Imtiaz Rahman. 2022. Using Annotations for Sensemaking About Code. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3526113.3545667

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

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

[31] Mik Kersten and Gail C. Murphy. 2005. Mylar: A Degree-of-Interest Model for IDEs. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development* (Chicago, Illinois) *(AOSD '05)*. Association for Computing Machinery, New York, NY, USA, 159–168. https://doi.org/10.1145/1052898.1052912

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

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

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

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

[36] Amy J. Ko, Htet Aung, and Brad A. Myers. 2005. 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* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 126–135. https://doi.org/10.1145/1062455.1062492

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

[38] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/985692.985712

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

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

[41] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971–987. https://doi.org/10.1109/TSE.2006.116

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

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

[44] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-Answer Questions about Code. In *Evaluation and Usability of Programming Languages and Tools* (Reno, Nevada) *(PLATEAU '10)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/1937117.1937125

[45] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. 2013. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering* 39, 2 (2013), 197–215. https://doi.org/10.1109/TSE.2010.111

[46] Jenny T. Liang, Maryam Arab, Minhyuk Ko, Amy J. Ko, and Thomas D. LaToza. 2023. A Qualitative Study on the Implementation Design Decisions of Developers. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 435–447. https://doi.org/10.1109/ICSE48619.2023.00047

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

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

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

[50] Michael Xieyang Liu, Aniket Kittur, and Brad A. Myers. 2021. To Reuse or Not To Reuse? A Framework and System for Evaluating Summarized Knowledge. *Proc. ACM Hum.-Comput. Interact.* 5, CSCW1, Article 166 (apr 2021), 35 pages. https://doi.org/10.1145/3449240

[51] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering* 39, 9 (2013), 1264–1282. https://doi.org/10.1109/TSE.2013.12

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

[53] Microsoft. 2023. *GitHub*. Microsoft. Retrieved September 11, 2023 from https://github.com

[54] Microsoft. 2023. *TypeScript: JavaScript with Syntax for Types*. Microsoft. Retrieved September 10, 2023 from https://www.typescriptlang.org/

[55] Microsoft. 2023. *Visual Studio Code*. Microsoft. Retrieved September 10, 2023 from https://code.visualstudio.com/

[56] G.C. Murphy, M. Kersten, and L. Findlater. 2006. How are Java software developers using the Eclipse IDE? *IEEE Software* 23, 4 (2006), 76–83. https://doi.org/10.1109/MS.2006.105

[57] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2023. In-IDE Generation-based Information Support with a Large Language Model. *arXiv preprint arXiv:2307.08177* (2023).

[58] Observable. 2023. D3 by Observable | The JavaScript library for bespoke data visualization. d3js.org

[59] Chris Parnin. 2013. *Programmer Interrupted*. ninlabs research. Retrieved November 21, 2023 from https://blog.ninlabs.com/2013/01/programmer-interrupted/

[60] Chris Parnin and Robert DeLine. 2010. *Evaluating Cues for Resuming Interrupted Programming Tasks*. Association for Computing Machinery, New York, NY, USA, 93–102. https://doi.org/10.1145/1753326.1753342

[61] Chris Parnin, Carsten Görg, and Spencer Rugaber. 2010. CodePad: Interactive Spaces for Maintaining Concentration in Programming Environments. In *Proceedings of the 5th International Symposium on Software Visualization* (Salt Lake City, Utah, USA) *(SOFTVIS '10)*. Association for Computing Machinery, New York, NY, USA, 15–24. https://doi.org/10.1145/1879211.1879217

[62] Peter Pirolli and Stuart Card. 1995. Information Foraging in Information Access Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '95)*. ACM Press/Addison-Wesley Publishing Co., USA, 51–58. https://doi.org/10.1145/223904.223911

[63] Ben Popper and David Gibson. [n. d.]. *How often do people actually copy and paste from Stack Overflow? Now we know.* Retrieved September 13, 2023 from https://stackoverflow.blog/2021/12/30/how-often-do-people-actually-copy-and-paste-from-stack-overflow-now-we-know/

[64] Eric Rawn, Jingyi Li, Eric Paulos, and Sarah E. Chasins. 2023. Understanding Version Control as Material Interaction with Quickpose. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 126, 18 pages. https://doi.org/10.1145/3544548.3581394

[65] Martin P. Robillard. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1292–1302. https://doi.org/10.1145/3468264.3473923

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

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

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

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

[70] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering* 34, 4 (2008), 434–451. https://doi.org/10.1109/TSE.2008.26

[71] Zéphyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2018. Noise in Mylyn interaction traces and its impact on developers and recommendation systems. *Empirical Software Engineering* 23, 2 (April 2018), 645–692. https://doi.org/10.1007/s10664-017-9529-x

[72] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3092703.3092717

[73] Sruti Srinivasa Ragavan, Sandeep Kaur Kuttal, Charles Hill, Anita Sarma, David Piorkowski, and Margaret Burnett. 2016. Foraging among an overabundance of similar variants. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 3509–3521.

[74] Captain Stack. [n. d.]. *Captain Stack - Code suggestion for VSCode.* Retrieved September 13, 2023 from https://github.com/hieunc229/copilot-clone

[75] Margaret-Anne Storey, Jody Ryall, Janice Singer, Del Myers, Li-Te Cheng, and Michael Muller. 2009. How Software Developers Use Tagging to Support Reminding and Refinding. *IEEE Transactions on Software Engineering* 35, 4 (2009), 470–483. https://doi.org/10.1109/TSE.2009.15

[76] Siddharth Subramanian and Reid Holmes. 2013. Making sense of online code snippets. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 85–88. https://doi.org/10.1109/MSR.2013.6624012

[77] Christoph Treude and Margaret-Anne Storey. 2012. Work Item Tagging: Communicating Concerns in Collaborative Software Development. *IEEE Transactions on Software Engineering* 38, 1 (2012), 19–34. https://doi.org/10.1109/TSE.2010.91

[78] Fernanda B. Viégas, Martin Wattenberg, and Kushal Dave. 2004. Studying Cooperation and Conflict between Authors with History Flow Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 575–582. https://doi.org/10.1145/985692.985765

[79] April Yi Wang, Zihan Wu, Christopher Brooks, and Steve Oney. 2020. Callisto: Capturing the "Why" by Connecting Conversations with Computational Narratives. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376740

[80] Dakuo Wang, Judith S. Olson, Jingwen Zhang, Trung Nguyen, and Gary M. Olson. 2015. DocuViz: Visualizing Collaborative Writing. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 1865–1874. https://doi.org/10.1145/2702123.2702517

[81] Moritz Wittenhagen, Christian Cherek, and Jan Borchers. 2016. Chronicler: Interactive Exploration of Source Code History. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '16)*. Association for Computing Machinery, New York, NY, USA, 3522–3532. https://doi.org/10.1145/2858036.2858442

[82] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2010. Understanding Feature Evolution in a Family of Product Variants. In *2010 17th Working Conference on Reverse Engineering*. 109–118. https://doi.org/10.1109/WCRE.2010.20

[83] YoungSeok Yoon and Brad A. Myers. 2015. Semantic zooming of code change history. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 95–99. https://doi.org/10.1109/VLHCC.2015.7357203

[84] YoungSeok Yoon and Brad A. Myers. 2015. Supporting Selective Undo in a Code Editor. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 223–233. https://doi.org/10.1109/ICSE.2015.43

[85] Young Seok Yoon and Brad A. Myers. 2014. A longitudinal study of programmers' backtracking. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 101–108. https://doi.org/10.1109/VLHCC.2014.6883030

[86] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and Supporting Adaptation of Online Code Examples. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 316–327. https://doi.org/10.1109/ICSE.2019.00046