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.

ACM Reference Format:

1 INTRODUCTION
Software engineering is a discipline about information management. When writing code, software engineers are typically managing many different tasks and questions [65]. 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 [40], are less often recorded due to the high cost incurred in externalizing these thoughts during the implementation process [29, 52]. Nearly all developers are considering such issues continually as they make implementation decisions [49]. 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, 61, 70, 71], with one study reporting that answering such questions was considered “exhausting” by participants, yet none of the participants recorded their own rationale [55]. Nonetheless, developers must continually understand unfamiliar code and reckon with these questions, with maintainers of code spending upwards of 50% of their time reading code written by themselves or others [40, 55, 59].

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 [40, 46, 55]. 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 [40, 46]. 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 [55].

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. [40], 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.

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 supporting later developers in answering questions related to code history, provenance, code relationships, and design rationale, given that these questions are significant blockers for developers maintaining code [44, 46, 55].

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 [58] extension for TypeScript [57], 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 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 [52] or because they are in a flow state, where pausing to externalize their thoughts is too burdensome [54]. 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 [37, 53] – 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 [50]) 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 [77], 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.

- **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 code base 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 and automating methods for capturing meta-information that can help developers reason about code design rationale (usage of AI code-generation systems and online information traces), history, relationships, and provenance (specific code editing patterns).
- A system, Meta-Manager, that collects these forms of meta-information with a design tailored to combat issues of scale and supports navigation for developer question-answering.
- A user study that demonstrates the efficacy of Meta-Manager for answering questions of code design rationale, history, provenance, and relationships, along with qualitative insights into what challenges remain for supporting this reasoning.

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.

2.1 Code Comprehension

Researchers in HCI and Software Engineering have extensively studied the practice of understanding code [8, 9, 12, 16, 41–43, 45, 55, 72, 74]. 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 [39, 44, 65, 76], and their growing and changing knowledge of the code [46, 55, 76]. Among these studies, there has been a focus on what are commonly called “hard-to-answer” questions about code [19, 40, 47, 55, 76]. 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 from a departed developer [61, 70, 71], maintaining and collaborating on a project [40, 55], or joining a new project [6, 32]. 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 [60], directly supporting developer sensemaking activities such as code commenting and annotating [23, 27, 29, 81, 83], using documentation [28], using print statements [31], and supporting easier navigation of code [8, 10], sometimes through sharing.
traces of navigation data from other developers within the code base [12]. Our work extends this research through explicitly attempting to address some of these comprehensive problems through a code-history exploration tool that extracts significant events worthy of investigation, without requiring extra work at design time, unlike related tools [27, 29, 81, 83] which require the developer to explicitly externalize their thoughts as notes [27, 29, 66] or code comments [81, 83] 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 developers’ navigation patterns and edit behaviors to create a degree-of-knowledge model to recommend code entities given a developer’s current task [20, 33] – Meta-Manager similarly leverages developer actions for prioritizing types of information. Further, in [20], code patches extracted by Mylyn were determined by experts not to 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. Codetrait [22] utilizes a shared information channel between the text editor and web browser to support and automate tasks such as using documentation. One feature of Codetrait that is particularly relevant to Meta-Manager is Codetrait’s identification of code copied from the web – when code is copied from online, Codetrait automatically creates a “bookmark” pointing to the web page in which the code snippet was copied from. Notably, Codetrait 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. We hypothesize that Meta-Manager’s versioning of information will make this feature more useful, as participants in the Codetrait study did not find bookmarking of information sources particularly useful and the authors posited that stronger connections between visited web pages and the code warrants more investigation.

2.2 Code 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 [87]. 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 [89–91]. 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. Notably, these visualization approaches are also used when visualizing non-code documents’ editing histories, with prior systems visualizing Wikipedia page edit histories [84] and Google Doc histories [86].

Some other code history visualizations do away with the timeline-style presentation. Quickpose [69] 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 Seseoft [15] and Augur [21] 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 [26] and Hipikat [11] 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 [64]. Other systems have leveraged discussion threads about code and mapped the threads to their corresponding source code implementations as a way to preserve history and design rationale [63, 85]. 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 [25, 34, 35, 75]. Traditionally, most software engineering teams utilize version control systems (VCS), such as GitHub [56], 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 [79]. 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 [73]. 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
Working with unfamiliar code happens in many different contexts. We then discuss how each feature in Meta-Manager instantiates our design goals and addresses these significant questions developers have about code.

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 [11, 14, 19, 32, 40, 46, 55, 61, 70, 71, 76]. Working with unfamiliar code happens in many different contexts, including maintaining a code base that has been edited by many engineers across time [19, 40, 46, 55, 76], joining a new project [11, 32], adopting a code base from a departed coworker [61, 70, 71], or using a new software library [14]. 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 [46] and there are minimal tooling options to support answering these questions, despite their ubiquity [55]. 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?** [14, 46, 55, 76] 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 [78], finding when some code was last used in service of understanding how a feature changed over time [88], getting “up-to-speed” on a new code base [32], or finding a snippet of code that was edited repeatedly to understand where the original developer had issues [46, 49, 78]. 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 [34].

- **Rationale: Why was this code written this way?** [32, 40, 46, 55, 71, 76] 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 [52] or because they believe it to be unimportant [55]), 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 [79]. Developers have stated that attempting to answer these questions are “exhausting” given the lack of tooling support and reticence to ask co-workers [55], 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?** [11, 19, 46, 76] 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” [8, 10]), 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 [46].

- **Provenance: Where did this code come from?** [19, 76] 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 [68]. 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 [92], and correctness of the code in terms of API usage, syntax, and so on [82]. 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 [36, 50, 51]. 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 [24].

### 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 a code example using the older version of the API. 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-Manager to 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 developers’ mental models of code [26] and given that our supported questions are often asked at the block or snippet level. This approach also complements our design goals of combating scale, considering each code block is in charge of its own history, meaning code versions are only captured when a block has changed. By deconstructing the versioning space to each code block and allowing each code block to manage its own history, we can support more fine-grained answering of questions related to history and provenance.

In order for Meta-Manager to begin logging code versions, the user does not need to take any actions beyond installing the extensions. On each file save, Meta-Manager will log a new version and perform an audit of the file to see if there are any new blocks of code to track. To investigate the code history, the developer can navigate to the “Meta Manager” tab in the bottom area of the editor — doing so will render the edit history of the user’s currently-open file. Whenever the user opens a file, the Meta-Manager will render that particular file’s history. Each code block’s history appears both within the visualization as a colored stream (Figure 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...
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.

```
= '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. 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 [37, 53] and our observations.

Similarly, if the user makes a programming-related Google search 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 [24]. By having this information versioned, we give the developer the tools to reason about this rationale across time.

3.3.2 Scalability. Given the sheer amount of information we are tracking with Meta-Manager, Meta-Manager is designed

---

3We 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.

3We 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 [53].
to support managing large amounts of information. We do this in
multiple ways – both collapsing information into a visual represen-
tation ([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
represents 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 that ChatGPT provided a significant contribution at this
time. In this way, the visualization itself can serve to answer some
questions about the code’s history on its own. The visualization
also contextualizes the annotated timeline of events (see Section
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
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
it occurred. This meta-information will change given the type of
event (see Figure 2 for an example of an in-IDE paste event).

We hypothesize that these edit events will be useful to later devel-
opers due to the diverse meta-information they generate, aligning
with our earlier discussions on developer information needs. As
discussed in Section 3.3.1-[F2], web activities of developers can elu-
cidate 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 visual-
ization 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 ques-
tions. 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
events 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 an-
notating 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 ver-
sions 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 [79]. 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 mul-
tiple purposes: enabling movement between un-annotated versions
along the timeline and providing a quick overview of code changes
over time. When the code box is expanded and the user is scrubbing,
the code will update for each version. This view complements the
visualization’s high-level representation of history with its lower-level code history representation and supports varied speeds of historical sensemaking, akin to how a user can scrub through, e.g., a YouTube video and speed up or slow down for targeted viewing. Users can comprehend the code history at different levels, aligning with where they are in their sensemaking journey.

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

3.4 Implementation
The Meta-Manager, both the Visual Studio Code editor extension and supplementary browser extension, utilize TypeScript for the logic and React [17] (with D3.js [62] for the chart in the Visual Studio Code extension) for the front end. FireStore [13] 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 [87]), 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.

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 [80] for a Visual Studio Code extension created by an external group unaffiliated with Meta-Manager, which functions similarly to CoPilot. This repository was chosen due to the fact that much of the code centers around the Visual Studio Code API, which few developers are familiar with, thus lowering the likelihood of a participant performing well purely due to having more background knowledge in the domain.

Our methodology draws from prior approaches that similarly explored developer sensemaking of code history through using a variety of online sources along with the first author’s rewriting of the code base to comprise the synthetic code base [35]. Code sourced from different online sources ensures the code base is unbiased by not only using code sourced from one individual representing one implementation style. To create the other artificial edits, the first author independently rewrote the code base, following along with the Git commit history in order to capture “real” versions of the code. While writing the code present in each commit, the tool was logging these real versions, but was also recording individual edits (e.g., add 1 line that says const searchResults = match(searchResults); in file search.ts on commit 4ac09121 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 [30]. 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. To further validate the realism of the code, we followed the same approach as [29] 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. 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
4.1.3 Task. Our main task draws from similar related work [12, 35] in that each participant was required to use Meta-Manager to answer 8 questions. Each question was designed such that it would represent at least one information need we are interested in (see Section 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. The complete code for Meta-Manager can be found at https://github.com/horvathaa/meta-manager.

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 familiar 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
### 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 information need, such as Q5, which both asks what code is related to the commented out loop, but also why the loop was commented out, which is a rationale question.

<table>
<thead>
<tr>
<th>Question in Task</th>
<th>Info. Need (Sec. 3.1)</th>
<th>Solution</th>
<th>Outcome</th>
<th>Avg. Time Spent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1. In config.ts, there is a regex for search pattern matching. Can you tell me what it is matching on and why?</td>
<td>Rationale</td>
<td>Find paste from ChatGPT, read user’s ChatGPT query</td>
<td>6 correct, 1 incorrect</td>
<td>3:28 (min: 0:58, max: 6:32, std. dev.: 1:59)</td>
</tr>
<tr>
<td>Q2. There is a bug in the commented out Promise code. Can you find where the bug was and what happened?</td>
<td>History</td>
<td>Find where Promise was initially commented out, where Promise came from, and look at versions before that event</td>
<td>4 correct, 3 out-of-time</td>
<td>5:04 (min: 1:58, max: 7:02, std. dev.: 2:24)</td>
</tr>
<tr>
<td>Q3. Prior to using parseHTML, the author was using a different API - what was it and why did they stop using it?</td>
<td>Rationale</td>
<td>Search to when parseHTML no longer exists, see what code was there before, and see Stack Overflow post</td>
<td>7 correct</td>
<td>4:35 (min: 2:23, max: 7:38, std. dev.: 1:46)</td>
</tr>
<tr>
<td>Q4. Recently, some code was added to search that came from a different file – can you find that code and explain what changed?</td>
<td>Provenance</td>
<td>Filter to see pasted code, find paste event with code copied from a different file, then search for that code in the file</td>
<td>7 correct</td>
<td>4:47 (min: 2:00, max: 8:08, std. dev.: 2:29)</td>
</tr>
<tr>
<td>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?</td>
<td>Relationships</td>
<td>Search on commented out code, click on ”Commented Out” event, find Stack Overflow post near event with replacement code</td>
<td>7 correct</td>
<td>4:24 (min: 1:57, max: 7:12, std. dev.: 1:44)</td>
</tr>
<tr>
<td>Q6. What code was generated by an AI system and what ended up happening to it?</td>
<td>History</td>
<td>Filter to see ChatGPT code, then search forwards in time on that code</td>
<td>5 correct, 2 out-of-time</td>
<td>6:36 (min: 2:02, max: 9:15, std. dev.: 1:53)</td>
</tr>
<tr>
<td>Q7. What were all the different things that the programmer tried when setting the match variable?</td>
<td>History</td>
<td>Search backwards in time on match, look at events</td>
<td>5 correct, 2 incorrect</td>
<td>5:18 (min: 2:15, max: 8:17, std. dev.: 2:15)</td>
</tr>
<tr>
<td>Q8. Some code from activate was moved into a different file. When did this happen and what was the code that was moved?</td>
<td>Provenance</td>
<td>Filter to code copied in activate, then see corresponding paste locations</td>
<td>7 correct</td>
<td>3:42 (min: 1:14, max: 8:35, std. dev.: 2:17)</td>
</tr>
</tbody>
</table>

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 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 combination of the visualization, zooming, and filtering worked together well to isolate “sub-histories” of the history to explore. A common strategy in answering history and provenance-based questions, used by 4 participants 12 times, was to use the annotated timeline labels as a boundary for a search space, then “zoom” into this space to look at the intermediate versions. For example, P3, in order to answer Q2, used the visualization to identify that there was a large growth in the code base at the end of the history and there was pasted code added at that time – he then zoomed into the end part of the history at the first instance of pasted code when the lines of code grew in order to reason about how the code changed after the addition of it and prior to it being commented out. In this way, participants were able to leverage the significant editing events, not only for meta-information, but also for their ability to segment the information space. This behavior of orienteering [2] to gain an understanding of part of the information landscape is consistent with behaviors exhibited in other information foraging studies [79], 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 [35]. 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.
Figure 4: Each question scored by participants in terms of how often they encounter similar questions in their own programming experiences.

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. Questions that required participants to reason not only about the history of their current file, but how that history relates to the history of other files caused confusion. Q2 required participants to reason about how some code in the current file changed, given its relationship to its original copy source in another file. Understanding the original code’s intent was necessary in order to better reason about why the code from the question was commented out – participants, with the “See Corresponding Copy” button, can see a preview of what the copied code looked like at the time of the paste. 2 successful participants and all unsuccessful participants struggled to reason about the connection between the “Corresponding Copy” version (which is on a different version within in a different file), the version of the code that received the paste, and how both of these information pieces related to the code in their current IDE. Future systems may investigate how to better support this reasoning across both time and space through supporting more interactive mechanisms for managing versions, which has shown success in other contexts [69].

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 [79] 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, [79] also discusses this phenomenon of information foraging being construed by users as assembling a "story", suggesting that our event labels may serve as one way of structuring these "stories".
When considering code history as a story, this framework is not dissimilar to the concept of "literate programming" and its philosophies, originally proposed by Donald Knuth [38]. Knuth believed that code should be more naturalistic, written as an expression of an author’s reasoning behind solving a computational problem. Programming, in its current state, typically relies on documentation as a way of translating between the lower-level code and its higher-level semantic meaning, with this documentation often spread across various platforms and represented using different modalities such as inline code comments, Git commit messages, GitHub pull requests and issues, and formal design documentation.

With the rise of LLMs for code generation, there is a new platform and modality for these natural language descriptions of code, which our work has shown are worth capturing as they can be used for reasoning about design rationale. The not-so-distant future of software engineering may consist primarily of this prompting for generating and modifying code – a future in which whole programs may be constructed predominantly through prompts that can be translated into code narratives not unlike the literate programs Knuth described. In this way, the code serves to describe the lower-level implementation but the higher-level goals and reasoning are communicated through the prompts. Meta-Manager begins to probe at how these forms of code-related meta-information may be captured and presented to help construct these narratives.

To the best of our knowledge, our work with Meta-Manager and its study are the first pieces of research to investigate AI-generated code provenance. Prior research has cited the importance of this research thread in order to answer questions such as whether or not "AI-generated code leads to fewer (or more?) build breaks" and if AI-generated code should be under more or less scrutiny during code reviews [7]. These questions, in theory, could be investigated using Meta-Manager through following the development of AI-generated code throughout its life-cycle. Meta-Manager also demonstrates that, through capturing AI code-generation provenance information, other questions and activities can be supported, such as reasoning about code design rationale.

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 can 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 [19], and typically used 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 [87, 89], 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 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 in different development 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 [55], 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 [81, 83] — 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 [48], or for saving a code version that a developer may be considering reverting back to [34, 35]. This 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 [18] 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