Washington University in St. Louis

WashU Scholarly Repository

McKelvey School of Engineering Theses & Dissertations

WashU Graduate Theses and Dissertations

Summer 8-2025

Supporting Programmers Answering the Hard-To-Answer Questions Through the Use of Large Language Models

Haixin Zhou
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Engineering Commons

Recommended Citation
Zhou, Haixin, "Supporting Programmers Answering the Hard-To-Answer Questions Through the Use of Large Language Models" (2025). *McKelvey School of Engineering Theses & Dissertations*. 1220.
The definitive version is available at https://doi.org/10.7936/8mqr-4p85

This Thesis is brought to you for free and open access by the WashU Graduate Theses and Dissertations at WashU Scholarly Repository. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of WashU Scholarly Repository. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science & Engineering

Thesis Examination Committee:
Caitlin Kelleher, Chair
Chien-Ju Ho
Alvitta Ottley
Sam Tihen

Supporting Programmers Answering the Hard-To-Answer Questions Through the Use of Large Language Models

by
Haixin Zhou

A thesis presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Master of Science

August 2025
St. Louis, Missouri

© 2025, Haixin Zhou

Table of Contents

List of Figures

List of Tables

Acknowledgments

During my studies at WashU, I received a lot of help from professors and my peers, and I am incredibly grateful for it. I first want to express my utmost gratitude to Professor Caitlin Kelleher for her guidance and patience throughout my entire Master's study here. Working with her is such a pleasure and I am so thankful for her mentorship. She helped me through the hardship of my thesis research and her guidance supported me to be where I am today.

I also want to thank my committee members, Professor Alvitta Ottley, Professor Chien-Ju Ho, and Professor Sam Tihen for their support. Their constructive feedback is invaluable and the discussion we had was thought-provoking.

I also must thank Tri Pham and John Allen for their support during my thesis study. They supported me during the entire research process and provided a helping hand during my hardship. Their knowledge of the project has made the research progress more smoothly.

Lastly, I want to thank my family and friends. I want to thank my family, my dog Jojo, and my cats Cleo and Luanne for their emotional and financial support. I want to thank my friends for encouraging me to keep going. Their support and encouragement bring a smile to my face as I am writing this.

Haixin Zhou

Washington University in St. Louis

August 2025

ABSTRACT OF THE THESIS

Supporting Programmers Answering the Hard-To-Answer Questions Through the Use of Large Language Models

by

Haixin Zhou

Master of Science in Computer Science

Washington University in St. Louis, 2025

Professor Caitlin Kelleher, Chair

Programmers often face hard-to-answer questions when working on large-scale projects. Not only during team collaborations, but also during interruptions such as meetings, having to suddenly work on another project, or even just taking a break are the major reasons these questions arise, as programmers have to regain the context, knowledge, and mental state to resume the work. Traditional text editors such as GitHub do provide a built-in history system, but they lack historical context behind the editing to answer such questions.

To address this issue, we developed an experimental extension, Code History, on Visual Studio Code (VS Code) in the hope of finding a way that enables programmers to make effective use of these subgoal-based histories to answer hard-to-answer questions. We have created this extension through a step-by-step process where one feature is built on top of another. We began with the interface design, where we created an interface that is intuitive for users. That created a solid foundation for the next mechanism of the extension in which the extension can generate subgoal summaries based on editing histories. In addition to displaying summarized user intentions, Code Histories also displays any web searches and links accessed by the programmers during the time when they were working on the subgoal. Lastly, we have integrated large language models (LLMs) such as ChatGPT and Gemini to improve the user's experience by letting the programmer search for a specific piece of history. Rather than being conversational, the LLMs are used to retrieve and filter relevant subgoal histories in response to the users' questions/commands. With the help of Code History, programmers can now access contextualized histories and answer questions about the code that would otherwise be difficult to answer through a list of history alone.

The goal of this thesis is to enable us to test the hypothesis that question-based interfaces for sub-goal histories will better support programmers in answering hard-to-answer questions than the commonly available supports today. Pilot studies have suggested that, with some minimal alteration to the test format and interface design, we will be able to move on to the user testing phase.

Chapter 1: Introduction

Asking questions is a natural part of the coding process. Whether they are facing code they are familiar with or completely new code, programmers often raise many questions as they try to understand and edit it. Some questions are easy to answer, like where an object is defined. However, there are also what we call "hard-to-answer" questions, which require a deeper understanding of the code and the reasoning behind it [1]. No matter if it is reading your own code after a long break or trying to make sense of someone else's logic, knowing why the code was written a certain way is just as important as knowing what was written and where. During this understanding process, many questions will naturally come up.

In many real-world situations, programmers return to a piece of code they wrote and have no clue what their original intention was. During this time, they might ask questions to themselves such as “Why is the logic done this way” or “Where did I get the information on how to do this difficult part” [2], [3]. It can be hard to tell whether the code was written during a sudden moment of inspiration or simply copied from the internet. Either way, to answer these questions and to understand why a certain algorithm was written in a specific way becomes a challenge. To answer these harder questions that might come up, programmers often have to digest dense information about the code on their own. The context they turn to usually involves Git commit histories, commit logs, or comments within the code. However, these sources often focus on what changed rather than explaining why the change was made [4]. In many cases, the information documented is not detailed enough to help programmers fully understand the original intentions behind the code or how the code came to be [5]. In other cases, programmers often have to dig through their commit histories or web search histories manually, trying to locate the resources or clues that explain the decisions they made [6]. This common problem happens when code editing history is documented but not effectively organized or utilized in a way that supports the needs of programmers.

Recent work done by Pham et al. has enabled the recording of a more detailed contextual history that includes the recording of not only the code editing before and after, but also any related web searches done by the user [7]. However, we do not know how to present it to the users in a comprehensive way.

This leads us to the question of how we can better utilize both code editing histories and web search histories to build toward a more detailed and useful annotated history. In this thesis, we present an extension that functions differently than traditional code change recording systems such as Git History, where we improved upon the work by Pham et al. and improved the interface usability of the extension in consideration of both code differences data and web search data. We explored how improved interface design and the integration of large language models (LLMs) can enhance programmers' ability to answer questions about their code.

There are three main developments for this study:

1. Designing an interface for the extension that improves user experience
2. Incorporating LLMs into the interface where subgoal summaries are inferred and generated through recorded code editing histories
3. Implementing a dynamic search feature that allows users to query their development history using natural language questions

These three major implementations will allow us to attempt to answer how we can better utilize recorded code editing history to improve programmers' performance. More specifically, we will address the two following overarching questions:

1. How can we make events in history more understandable to programmers?

2. And how can we incorporate a question-based interface to present historical information?

This research aims to create an interface that is able to display important historical information such as code editing history and web browsing history in a more intuitive and meaningful way to the programmers, making a potentially overwhelming amount of information usable. Ultimately, we hope this research provides evidence to the hypothesis that this unique way of recording history better supports the hard-to-answer questions developers ask than traditional history recording tools such as Git.

1.1 Rationale

In software development, programmers often encounter questions that are rather difficult to answer. These may involve understanding why a specific change was made, what effect a piece of code has on the program, or how and when a specific bug was formed. These types of questions are often hard to answer because they require access to contextual information that is no longer visible in the code editor and is not traditionally captured today. They demand not just an understanding of the code itself but also of its editing history, the reasoning behind the decisions, and even outside influences such as relevant web resources accessed by the programmers during early development.

1.1.1 Hard to Answer questions asked by programmers during the coding process

In *Hard-to-Answer Questions about Code*, LaToza et al. demonstrate just how frequently developers face questions that are difficult or impossible to answer using only the information available in the current codebase. These questions often require historical knowledge, such as why a change was made, what alternatives were considered, or what the developer's original intent was. These questions are highly important as programmers navigate through the given code base and attempt to familiarize themselves with it [2].

Before continuing our research, we must understand what constitutes a hard-to-answer question. The foundational research conducted by LaToza et al. in this area identifies that the most difficult questions programmers face often revolve around understanding the intent behind a code change or assessing its broader impact. As demonstrated in Figure 1.1, the more common questions asked by the participants in this research are related to changes, such as rationale, debugging, and history. When designing our extension, we use these questions as examples of what a programmer will ask during their coding process. The interface will attempt to address questions such as those mentioned in LaToza et al.'s paper.

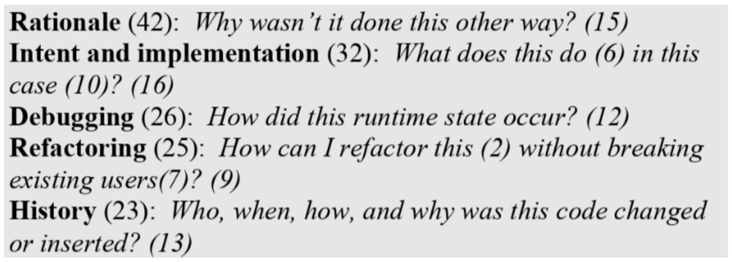

Rationale (42): *Why wasn't it done this other way? (15)*
Intent and implementation (32): *What does this do (6) in this case (10)? (16)*
Debugging (26): *How did this runtime state occur? (12)*
Refactoring (25): *How can I refactor this (2) without breaking existing users(7)? (9)*
History (23): *Who, when, how, and why was this code changed or inserted? (13)*

Figure 1.1. Five most frequently reported question categories, the number on the right shows the number of participants in LaToza et al.'s study asked that type of question

1.1.2 Why are we trying to answer hard-to-answer questions

Answering such hard-to-answer questions requires levels of understanding about the code; therefore, comprehending a given code segment and understanding how it came to be is crucial. As highlighted in *40 Years of Designing Code Comprehension Experiments* by Siegmund et al., understanding code is not just something a programmer can put aside and come back to whenever they want to [8]. It is something programmers do constantly, whether they are debugging, adding new features, onboarding to a new project, or simply trying to figure out what a teammate's code is doing. The paper reviews decades of empirical studies and shows that code comprehension is not only very time-consuming but also deeply affected by how the code is written, the mental strategies developers use, the tools available to them, and even external factors like time pressure or interruptions. The challenge in code comprehension also demonstrates supporting programmers in answering the hard-to-answer questions is no easy task.

What stands out is that despite forty years of focused work in this area, Siegmund et al. show that answering questions that require a deeper understanding of the code remains a persistent challenge. Fortunately, an increasing number of recent studies have focused on addressing this issue, in the hope of providing meaningful improvements in the future. This is especially important given that developers are regularly expected to read through unfamiliar codebases and piece together how everything fits. This process can take up a significant portion of their day, yet it is often unsupported or poorly supported by the tools they use. Siegmund et al. make it clear that helping programmers understand and be able to answer related questions about the code should be a priority because it is relevant to every part of the development process, from writing efficient and accurate code to maintaining existing programs and collaborating effectively with others.

In our research, we build on this motivation by exploring how access to historical development contexts, such as previous code states and web searches, can help developers recover from interruptions and answer difficult questions about unfamiliar code. If we know how programmers interpret code and what kinds of information help them do it better, we can design smarter tools to support that process.

1.2 Related Works

Similar studies have explored the difficulties developers face when working in an unfamiliar code base, especially when they encounter hard-to-answer questions about their code. These are the kinds of questions that go beyond surface-level understanding and require deeper comprehension of the code segment. As systems grow more complex and collaboration becomes more common, the need to quickly and accurately answer these hard questions has become even more important. It is essential for modern programmers to understand not just what the code does, but also why it was written in a specific way. Many researchers have tried to tackle the problem of code comprehension by developing tools that help programmers understand a piece of code. These tools are often designed to overcome the shortcomings of GitHub histories and to reduce the mental effort it takes to figure out the purpose behind the code, especially when that purpose is not immediately obvious from the code alone. We will introduce these tools and how the tools we have created uniquely target a different aspect of code comprehension, in particular answering the hard-to-answer questions. We will also discuss how we can improve upon these projects and really put emphasis on answering these hard-to-answer questions.

1.2.1 Code Compass

As codebases grow in complexity, developers increasingly need to rely on tools to help them understand unfamiliar code and experiment with the code without breaking it. Various tools have been introduced to assist with navigating large or inherited codebases, with one example being *Code Compass*.

In *Code Compass: A Study on the Challenges of Navigating Unfamiliar Codebases*, researchers discovered that programmers often not only struggle to locate important information but also find it difficult to understand the purpose of specific code segments [9]. Their findings also suggest that traditional tools such as Git History often fail to provide sufficient context, often forcing programmers to rely on colleagues to explain their code segments or build understanding through repeated trial and error exploration. On top of their findings, they have also introduced a novel tool called Code Compass to address code comprehension issues. Code Compass works by integrating contextual documents and visualizations directly into the code editing platform. Their tool helps locate the definition of a method/function, bringing the definition directly next to the code, and preventing programmers from having to tediously look for function definitions. Code Compass also tries to match helpful references to reduce the time programmers have to switch tabs. It allows programmers to leave visual cues to remind themselves how parts of the code relate to each other. Another important functionality they have introduced is the function of sandboxing. Code Compass has a sandbox functionality where programmers can experiment with a block of code without worrying that any significant changes they make will impact or break the existing codebase.

While Code Compass enhances navigation of unfamiliar codebases by integrating function definitions, contextual documents, and visual aids, along with a sandbox environment for safe experimentation, our project focuses instead on capturing and organizing editing history and web activity to provide rich contextual grounding. Rather than supporting hands-on code exploration, our system is designed to help programmers answer hard-to-answer questions by reconstructing the intent behind past changes. We do not offer sandbox functionality, as our goal lies more in surfacing historical context than in enabling code manipulation.

1.2.2 Meta-Manager

Like Code Compass, which aims to improve code comprehension through better navigation and the ability to experiment with code, Meta-Manager also addresses the challenges of understanding unfamiliar code, with an emphasis on locating code that is copied from the internet and determining whether a block of code is AI-generated or not.

In *Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code*, Horvath et al. introduced another tool to tackle the everlasting problem of code comprehension. Meta-Manager is a Visual Studio Code extension with a complementary browser extension. Meta-Manager, though it records all the code editing histories, puts a heavy emphasis on how code that is pasted from the web ends up in the project. Having the web search histories documented allows programmers to locate code that is copied from the internet more quickly; they are able to access their original source code faster, thus improving the efficiency of code comprehension [10].

Our project shares a similar motivation: to improve programmers' code comprehension through historical context. However, rather than focusing on detecting whether code is from the internet or AI-generated, we put more emphasis on having histories in a searchable query, improving the overall lookup time, not just focusing on the copied and pasted code online. We also put a heavy emphasis on using historical information to answer hard-to-answer questions about the code, whereas Meta-Manager is a tool primarily for keeping track of any referenced code and AI-generated code.

1.2.3 Information Recovering Through Storytelling

Another relevant study explored the potential of storytelling in improving programmers' understanding of a given code base.

In *Exploring the impacts of semi-automated storytelling on programmers' comprehension of software histories*, Allen et al. investigated how they can help developers understand the rationale and intent behind a code base's design decisions [11]. They found that storytelling is especially beneficial when trying to present information that is complex and time-dependent. As part of their research, they created a tool that utilizes generative AI to help programmers understand code histories by transforming raw change logs into narrative-driven stories. This study indicates that AI-generated storytelling improves not only the recall of historical information in code but also helps programmers understand the code better, as they perform better in answering questions regarding the code.

Both this study and our study rely on the capabilities of AI to improve programmers' code comprehension and reformat history into an accessible presentation; however, our projects differ as we focus less on the narrative-driven presentation and more on the query ability of the recorded histories.

1.3 Research Overview

This thesis will attempt to address how we can incorporate large language models (LLMs) into an interface to support programmers in answering typically hard-to-answer questions that might come up during their coding process. This paper presents the development and proposed evaluation of a VS Code history recording extension with an interface that can infer and generate high-level user coding objectives while also allowing users to search within their recorded histories.

There are three main contributions described in this paper:

1. Redesigned interface that demonstrates an approach to incorporating history into the IDE.
2. An approach to generating code history overviews using LLM support.
3. A novel query mechanism that enables programmers to ask historical questions.

This research presents the final user interface design and the process of how we improved upon the original version of the extension. This lays the foundation for the incorporation of large language models, which allow us to implement the summarization and search features. This paper also presents pilot studies conducted to test whether we can move on to the user study phase. The pilot study results show the design of the proposed user testing needs a few improvements before ready to move on to the user testing process as users behave unexpectedly. Based on these findings, we propose a revised user study method that addresses the issues that were discovered during the pilot study phase.

Chapter 2: Interface Improvement

To better understand how the redesigned interface supports answering hard questions about code development, it is important to recognize the foundation this project is built upon. This work extends a pre-existing Visual Studio Code extension, Code History, originally developed by Pham et al., as introduced in their paper *Code Histories: Documenting Development by Recording Code Influences and Changes in Code* [7]. However, its original interface presented usability challenges that limited its effectiveness in helping users engage meaningfully with their own development history.

In order to support programmers through an LLM-integrated VS Code extension, it is important to first introduce an interface for the extension that can be easily navigated. This chapter details a redesign of the interface with a focus on improving usability, readability, and clarity, making this part a foundation for the later implementation of summarizing and querying functions. The new layout introduces better visual organization and naming conventions to help programmers understand the extension they can utilize.

2.1 Redesign Goal

The main design goal at this step is to create an interface that allows programmers to easily locate and understand both small and significant past code changes. In doing so, the new layout aims to avoid overwhelming users with an unfiltered chronological list of all edits, as was the case in the original version. Instead, it will visually separate minor and major changes, use clearer section titles, and make navigation more intuitive for the programmers, whether they are familiar with the recorded history interface or not.

In earlier versions, as we will introduce in the next section, users had to scroll through every recorded change with little context or structure. This made it difficult to identify meaningful development steps or understand how smaller changes built toward larger goals. To address this, the redesigned interface now highlights key development events and organizes ongoing edits in a visually structured way, promoting a clearer mental model of progress over time.

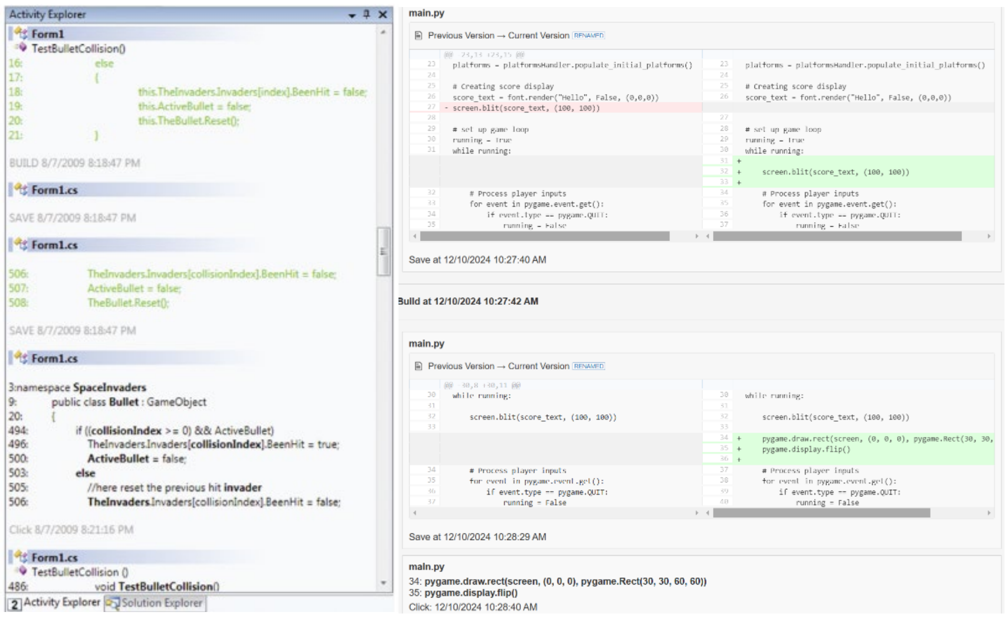
We envision this interface as a practical tool that helps developers make sense of their past work or understand unfamiliar code segments. It should clearly indicate that one section displays all significant past edits that contributed to the current state of the code, while another section is dedicated to recent, smaller changes that may not have had a significant impact yet. Users should also be able to view the interface according to their preferred code change format, whether it is line-by-line or side-by-side. They will also, compared to traditional Git commits where internet searches are not recorded, be able to access past links or search results that led them to write their code in a certain way.

In conclusion, this redesigned interface aims to support developers in answering complex, history-based questions about their code. The interface will be more accessible and user-friendly while providing vital information for developers during their coding process.

2.2 Original Interface Evaluation

The initial goal was to encourage users to actively engage with their development history. However, the earlier versions of this extension interface did not do that.

The earliest version of the interface design, based on the design described in *Evaluating Cues for Resuming Interrupted Programming Tasks* by Parnin et al., followed a more traditional layout. It is a chronological display showing all code edits and web activity, ordered from least recent to most recent, as shown on the left side of Figure 2.1 [12]. On the right side is an interface developed for this study following the exact structure of the interface mentioned by Parnin et al. While this approach efficiently documented every action taken by the programmer, including every minute change, it simply listed out every single change without summarizing or grouping them into a more readable or meaningful format, such as a natural language summary of what has been done. As a result, the interface becomes overwhelming to navigate, especially when users need to quickly locate specific changes or recall the reasoning behind them. It lacked the structure to highlight important edits or present developing edits in a way that supported deeper code understanding.

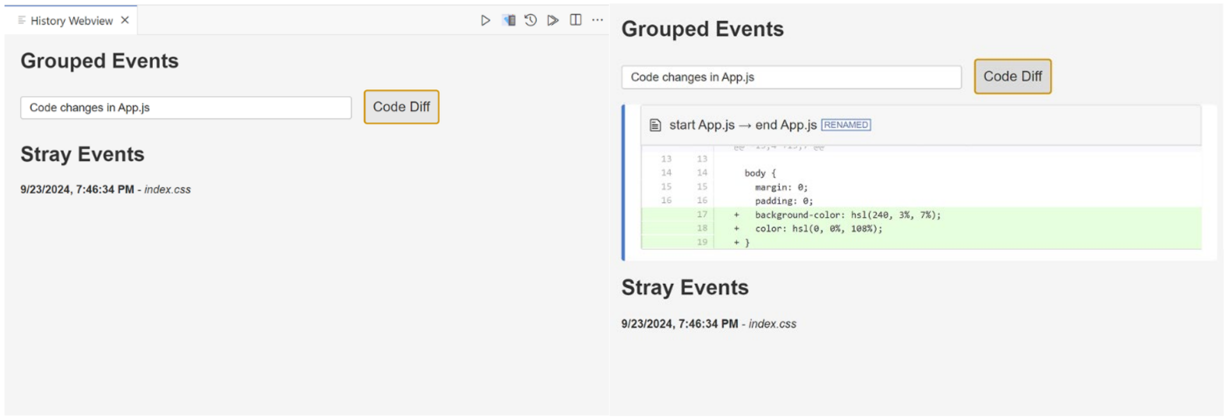


```
Form1
  TestBulletCollision()
16:            else
17:            {
18:                    this.TheInvaders.Invaders[index].BeenHit = false;
19:                    this.ActiveBullet = false;
20:                    this.TheBullet.Reset();
21:            }

BUILD 8/7/2009 8:18:47 PM

Form1.cs

SAVE 8/7/2009 8:18:47 PM

Form1.cs

506:           TheInvaders.Invaders[collisionIndex].BeenHit = false;
507:           ActiveBullet = false;
508:           TheBullet.Reset();

SAVE 8/7/2009 8:18:47 PM

Form1.cs

3:namespace SpaceInvaders
9:      public class Bullet : GameObject
20:     {
494:        if ((collisionIndex >= 0) && ActiveBullet)
496:           TheInvaders.Invaders[collisionIndex].BeenHit = true;
500:           ActiveBullet = false;
503:        else
505:           //here reset the previous hit invader
506:           TheInvaders.Invaders[collisionIndex].BeenHit = false;

Click 8/7/2009 8:21:16 PM

Form1.cs
  TestBulletCollision ()
486:          void TestBulletCollision()
```

```
main.py
Previous Version → Current Version RENAMED
@@ -23,14 +23,15 @@
  platforms = platformsHandler.populate_initial_platforms()    |   platforms = platformsHandler.populate_initial_platforms()
                                                               |
  # Creating score display                                     |   # Creating score display
  score_text = font.render("Hello", False, (0,0,0))            |   score_text = font.render("Hello", False, (0,0,0))
- screen.blit(score_text, (100, 100))                          |
                                                               |
  # set up game loop                                           |   # set up game loop
  running = True                                               |   running = True
  while running:                                               |   while running:
                                                               | +
                                                               | +     screen.blit(score_text, (100, 100))
                                                               | +
      # Process player inputs                                  |       # Process player inputs
      for event in pygame.event.get():                         |       for event in pygame.event.get():
          if event.type == pygame.QUIT:                        |           if event.type == pygame.QUIT:
              running = False                                  |               running = False
Save at 12/10/2024 10:27:40 AM

Build at 12/10/2024 10:27:42 AM

main.py
Previous Version → Current Version RENAMED
@@ -30,8 +30,11 @@
  while running:                                               |   while running:
                                                               |
      screen.blit(score_text, (100, 100))                      |       screen.blit(score_text, (100, 100))
                                                               |
                                                               | +     pygame.draw.rect(screen, (0, 0, 0), pygame.Rect(30, 30,
                                                               | +     pygame.display.flip()
                                                               | +
      # Process player inputs                                  |       # Process player inputs
      for event in pygame.event.get():                         |       for event in pygame.event.get():
          if event.type == pygame.QUIT:                        |           if event.type == pygame.QUIT:
              running = False                                  |               running = False
Save at 12/10/2024 10:28:29 AM

main.py
34: pygame.draw.rect(screen, (0, 0, 0), pygame.Rect(30, 30, 60, 60))
35: pygame.display.flip()
Click: 12/10/2024 10:28:40 AM
```

Figure 2.1. Diagram on the left is the history documentation cue structured by Parnin et al. and the Diagram on the right is a recreation of the one mentioned on the left, with the same structure

Pham et al. improved the original naive interface by separating the history into two categories: Stray Events and Grouped Events. Stray Events are smaller edits with ambiguous purposes that, without the context of related searches or other changes, may appear unclear in intent. As development continues, they may eventually form part of a Grouped Event. The Grouped Events section displays significant code changes made by the programmer, potentially occurring at different points in time. Pham et al. developed a heuristic where, when the extension notices changes done by the user that can be considered significant, the interface will store the before and after of these related changes together as significant code editing in the project programming history. The code changes then get grouped together and move from Stray Events to Grouped Events. At this point, the interface will call each grouped event as "Code changes in *file_edited_by_user.js*." The actual code changes can be viewed after clicking on the Code Diff button, as seen in Figure 2.2. This allows programmers to quickly know where the editing takes place before checking the details of the code segment [7].



Grouped Events

Code changes in App.js

Stray Events

9/23/2024, 7:46:34 PM - *index.css*

Grouped Events

Code changes in App.js

start App.js → end App.js RENAMED

```

  body {
    margin: 0;
    padding: 0;
+   background-color: hsl(240, 3%, 7%);
+   color: hsl(0, 0%, 100%);
+ }
```

Stray Events

9/23/2024, 7:46:34 PM - *index.css*

Figure 2.2. Revised interface after the recreated interface of the one Parnin et al. developed

However, this interface still needs rounds of revision for it to be more user-friendly and easy to navigate. First of all, it oversimplifies the two sections with the use of vague wording that can also lead to confusion. Grouped Events and Stray Events might seem clear to the developers writing it, but it can confuse the user as to what "grouped" means and what "stray" means. Having a singular naming of the subgoal summary also defeats the purpose of grouping the larger edits together, as the subgoal summary only explains the location of those changes, but there are no further details of what the code changes are about. The users will be forced to click on the Code Diff button to reveal the details, and the comprehension of that segment is not necessarily guaranteed. At the bottom, there are no actual code change comparisons for smaller changes, which can be inconvenient if the users are just seeing numerous files listed without showing any code changes.

2.3 Final Design

To address the mentioned problem and reach the goal of encouraging programmers to interact with the interface, multiple iterations of thumbnail sketches were done in the hope of coming up with a more usable design. Many layouts were considered in this case, including split sections vertically, different naming conventions, etc., as shown in Figure 2.3. The renaming of each section was also done in multiple iterations. However, we chose to have the completed design continue to employ a horizontal split with the top panel displaying grouped events and the bottom panel displaying stray events.

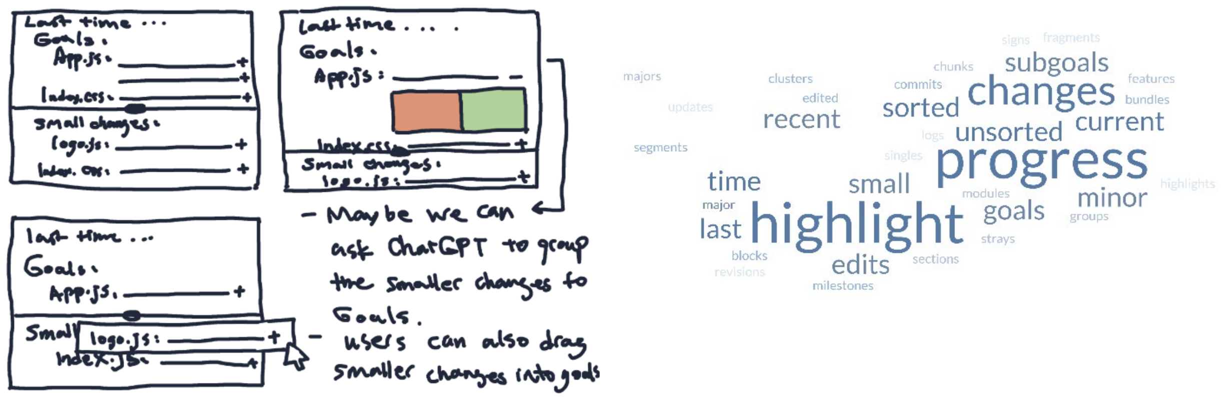
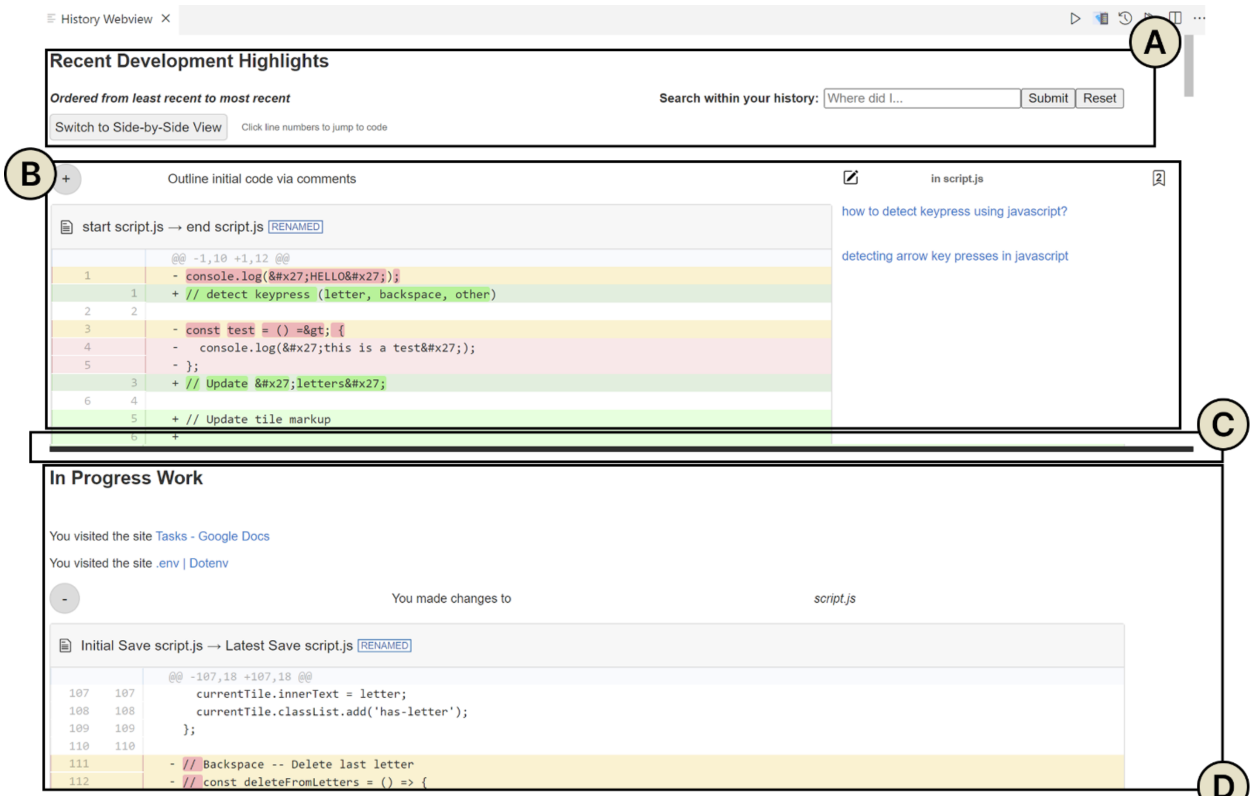


Figure 2.3. Thumbnail sketch and word cloud for potential section naming



Recent Development Highlights

Ordered from least recent to most recent **Search within your history:** Where did I... Submit Reset

Switch to Side-by-Side View Click line numbers to jump to code

Outline initial code via comments in script.js

how to detect keypress using javascript?

detecting arrow key presses in javascript

start script.js → end script.js RENAMED

```
@@ -1,10 +1,12 @@
- console.log(&#x27;HELLO&#x27;);
+ // detect keypress (letter, backspace, other)

- const test = () =&gt; {
-   console.log(&#x27;this is a test&#x27;);
- };
+ // Update &#x27;letters&#x27;

+ // Update tile markup
+
```

In Progress Work

You visited the site Tasks - Google Docs

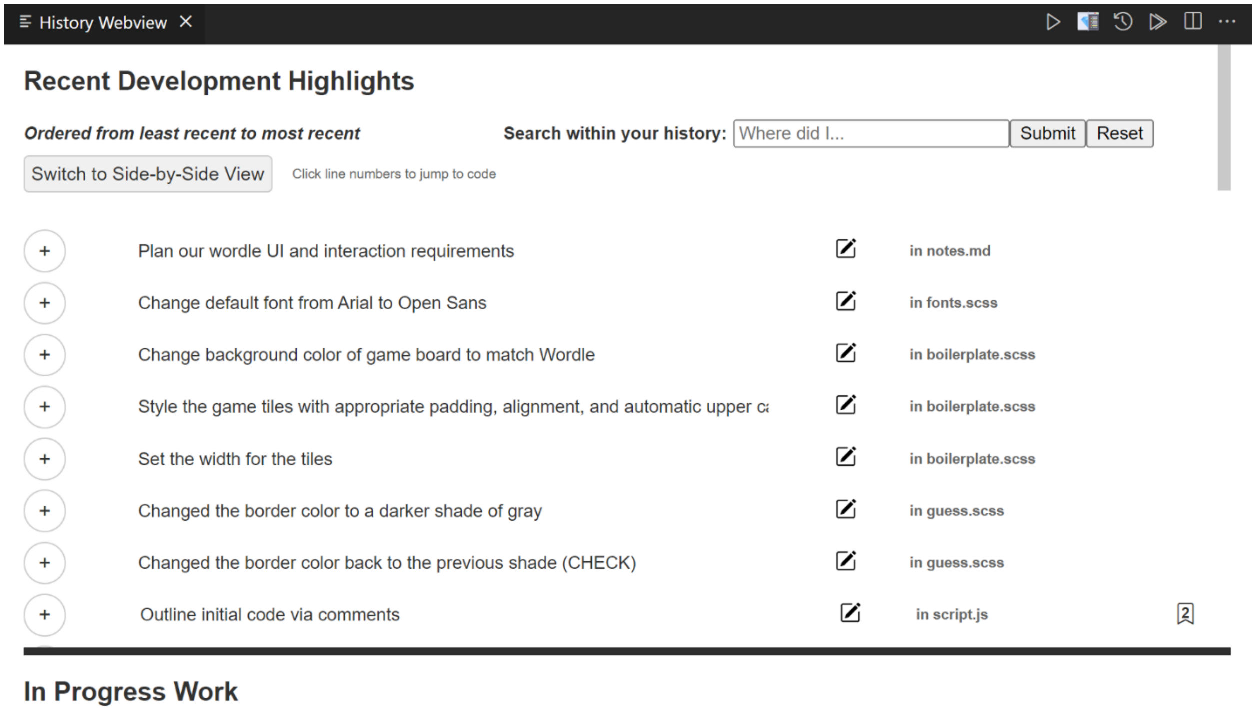
You visited the site .env | Dotenv

- You made changes to *script.js*

Initial Save script.js → Latest Save script.js RENAMED

```
@@ -107,18 +107,18 @@
      currentTile.innerText = letter;
      currentTile.classList.add('has-letter');
    };

- // Backspace -- Delete last letter
- // const deleteFromLetters = () => {
```

Figure 2.4. Final design of the interface with subgoal expanded and In Progress Works with sample site visits



Recent Development Highlights

Ordered from least recent to most recent **Search within your history:** Where did I... Submit Reset

Switch to Side-by-Side View Click line numbers to jump to code

+	Plan our wordle UI and interaction requirements	in notes.md
+	Change default font from Arial to Open Sans	in fonts.scss
+	Change background color of game board to match Wordle	in boilerplate.scss
+	Style the game tiles with appropriate padding, alignment, and automatic upper ca	in boilerplate.scss
+	Set the width for the tiles	in boilerplate.scss
+	Changed the border color to a darker shade of gray	in guess.scss
+	Changed the border color back to the previous shade (CHECK)	in guess.scss
+	Outline initial code via comments	in script.js

In Progress Work

Figure 2.5. Final interface design with sample list of subgoal tasks that was also used in pilot studies

For the final design, the code edits will still be categorized into two types: grouped events, now shown as Recent Development Highlights, and stray events, now shown as In Progress Work. We have decided to change from Grouped Events to Recent Development Highlights. The intention behind the change of wording is to address the confusing wording issue of the previous interface. "Highlight" indicates the list below is the milestones that the programmers have done. Similarly, we changed the lower panel from Stray Events to In Progress Work for the same reason. "Stray" on its own is too vague to really convey that the changes shown below are in the process of achieving a specific goal that the programmer wants to achieve.

The final design of the interface, shown in Figure 2.4, can be separated into five sections. We will introduce the changes as marked sections in the Figure.

Starting with section A, we have decided to change from Group Events to Recent Development Highlights. The intention behind the change of wording is to address the confusing wording issue of the previous interface. "Highlight" indicates the list below is the milestones that the programmers have done. To indicate how the highlight list is organized, we added a small text to remind users that they are "Ordered from least recent to most recent." Below is a button which allows users to switch whether they want to view the code changes side-by-side or line-by-line. On the right is a search function that allows users to search within their own history to find useful information, which we will discuss in more detail in Chapter 4.

Section B consists of the list of Development Highlights. Each item in the Recent Development Highlights panel corresponds to a set of related code changes grouped under a shared goal, as shown more clearly in Figure 2.5. These subgoals are generated by ChatGPT based on the user's code edits, as we will discuss in more detail in Chapter 3. The AI-generated subgoal summaries can be manually edited by the user if the AI-generated description does not fully capture their intent. To the left of the title, a circular button reveals the full code changes, as shown in Figure 2.4. If the user expands the subgoal view using the plus icon, they can click on specific line numbers to be taken directly to those parts of the code. To the right, the interface displays the location of the changes in the codebase, maintaining the original interface design where it quickly reminds users where the change took place. To the right of the location text, there is a small bookmark with a small number inside it that indicates the number of web resources accessed by the user. If a subgoal does not have bookmarks, it indicates that no web resources were accessed during the time where the programmer was working on that subgoal. When expanded by clicking on the expand button, the titles of the accessed websites, displayed as clickable hyperlinks, allow the programmer to revisit those resources if needed. Not just the websites visited by the users, the search phrase users have used to get the web sources will also be displayed here.

Section C is a draggable handlebar shown between the panels so that users can adjust the ratio of each panel, allowing them to choose which panel they want to focus on. This effectively gives the programmers control over the information density, allowing them to focus on one panel without completely losing the other.

Section D is the In Progress Work section, where smaller or unstructured changes are here. These changes haven't yet formed a subgoal, but they are still visible and will be automatically grouped into a subgoal once a pattern or intent is detected. It follows a very similar structure to Section B, where the only difference is that it does not have an AI-generated summary as it has yet to present a clear goal. This section also displays visited sites without grouping them with a subgoal. This also gives programmers freedom to revisit the links without having to complete a specific goal.

2.4 Formative Studies

In total, seven formative studies were conducted to test the participants' interactivity with the interface. Though the study here focuses on how the interface can help programmers recover after an interruption, the results are still valuable for improving the interface further.

The process of the pilot studies goes as follows:

1. Introduction: Participants are introduced to the study where they are told that they will interact with two different codebases, one with is a tile matching game written in JavaScript with several coding tasks related to the game, and the second one is a doodle jump game written in Python with several coding assignments.
2. Code base 1: Tile Matching Game, the participants are expected to complete the following task.
    a. Familiarizing task:
        i. Search pygame documentation online
        ii. Create the display_message Function and save
        iii. Call display_message
        iv. Run "codehistories python main.py" in terminal and see the how the interface updates
    b. Tasks:
        i. Add a Scoring System Based on Matches Made
        ii. Implement a Countdown Timer with Game Over State

        iii. Display Session High Score and Current Game Score
        iv. Bonus Task: Add a Hint System for Possible Matches
3. Code base 2: Doodle Jump Game
    a. Familiarizing task:
        i. Modify score display
        ii. Save and execute “codehistories python main.py” in terminal
    b. Tasks:
        i. Add a Scoring System Based on Player’s Highest Level
        ii. Create an End Game Screen
        iii. Keep Track of the Highest Score and Display It
        iv. Bonus Task: Add Moving Platforms
4. Concluding interview: Participants will be interviewed on what was their thought process and whether the interface given was helpful.

The participants will have access to different interfaces for each codebase. One of the interfaces is a replica of the interface developed and recreated by Pham et al., and the other interface is the newly developed interface created by this research. One additional important detail about the pilot studies is that while the participants are working on the tasks portion of the study, they are interrupted six times in total, each time with completely random coding questions to disrupt their thought process. After the interruption, we expect users to navigate back to where they were working using the interface.

2.5 Formative Studies Evaluation

Through pilot studies, we found that many participants did not immediately start to use the new extension features. Rather than utilizing the features offered by the interface, they preferred to spend a significant amount of time manually searching for specific segments of code or re-entering the same search phrases on the internet. This behavior persisted even when the interface clearly offered tools that could have helped them locate the necessary information more efficiently.

When we asked participants why they did not use the more advanced features provided by the interface, the majority explained that they were simply not accustomed to having such a tool available. Many admitted that they often forgot the interface was there to assist them during the development process. This feedback reveals a crucial insight: even if a tool is functionally powerful and well-designed, it may not be fully adopted unless it becomes an integrated part of the user's workflow.

Despite this tendency to revert to habitual coding practices, most participants agreed that the redesigned interface was more helpful and easier to navigate compared to earlier versions. They appreciated the clarity of the layout, the structured presentation of code history, and the visibility of past searches. The pilot study results suggest that the design direction is promising, but adoption may require more intentional task structures and user engagement strategies.

Based on these findings, we recognize the need to revise our study design. Future iterations of the pilot will include tasks that require participants to interact with the interface directly. This change will help ensure that users experience the full benefit of the tool's features, and it will also allow us to evaluate how effectively the interface supports task completion when it is actively used. By guiding users through tasks that depend on the interface, we aim to promote greater familiarity with the tool and encourage its adoption in real-world coding workflows.

However, on the bright side, the redesign of the interface did prove to be beneficial. One of the formative study participants stated that they can see themselves using this interface for their own projects. Another participant stated that “The history interface [redesigned interface in this case] was helpful because just by looking at it, they can immediately tell what they were doing.” This positive response indicates a positive impact that the redesign had, making the interface more navigable. This portion provides a solid foundation for the functions we will introduce in the next two chapters.

Chapter 3: Summary Generation

The summary generation feature is a key component of the redesigned interface that aims to allow developers to locate a change they have in mind with more ease, as the summarized subtitle will be more readable. Not only does summary generation allow for code changes to be more readable, but there are also benefits to code comprehension. Users are no longer using just raw code to build understanding; they are using the summary as a quick starting point to identify the most relevant sections. This feature is also a vital foundation for the searching function that will be introduced in the next chapter, as the summarized subgoal will be the key for the search function to perform correctly. The summary generation then opens the possibilities of helping users to answer hard-to-answer questions.

In the original interface, a simple heuristic was used to detect major code changes and group them into a single subgoal. However, as discussed in Chapter 2.2, the resulting subgoal summaries used a generic placeholder format, usually labeled as "Code Changes in [file name]," which provided little insight into the programmer's actual intent.

As briefly introduced earlier in Chapter 2.3, the redesigned interface integrates a summary generation feature that utilizes a large language model to infer and display the programmer's intention behind code edits. When a set of changes is detected, the system sends the before-and-after versions of the code to ChatGPT through an API connection. Based on the differences between the two versions, ChatGPT generates a summary representing the underlying intent of the edit, referred to as a "subgoal." This subgoal is then displayed within the Recent Development Highlights portion and can be edited by the user if desired. The interface supports viewing the associated changes in either a line-by-line or side-by-side format, depending on user preference.

In this chapter, we will explain the process of how the summary feature came to be.

3.1 Function Goal

The goal of the summary generation function is to help users understand the purpose behind individual edits or clusters of edits without having to manually review every change in detail. By converting raw code changes into summarized intent statements, the interface should be able to support users' understanding of how the code came to be and, therefore provide support in answering those hard-to-answer questions that may come up during the coding process. This feature will be especially useful when users return to a project after a break, continue improving one section of the code after working on another completely different portion of the code, or just need to check development history across multiple edits. The formed subgoal summary should be accurate and easy to understand. Ideally, the user should be able to understand a group of changes at a glance without having to read into the code or having to read the summaries multiple times to understand.

3.2 Final Implementation of Summary Feature

The final implementation of the summary function can dynamically generate user subgoal summaries as they work on the code and make any sort of edits. The summary generation function is implemented as an asynchronous call to a large language model API; in this case, we used the ChatGPT API.

The process of generating a subgoal summary is shown in Figure 3.1. The extension developed by Pham et al. decides when a change in a file is significant enough for it to be considered a subgoals; then it will dynamically store the before and after. The code change after, in this case, consists of everything that the programmer edited. It constructs a structured prompt containing these two versions of the code stringified, which is then sent to the language model for processing. As shown in Figure 3.2, the prompt provides information about which file the given code segment is from and the stringified before and after editing version of the code. We made sure the commands such as “Make sure it sounds like a natural conversation,” are included are included so that the summary generated is easily readable.

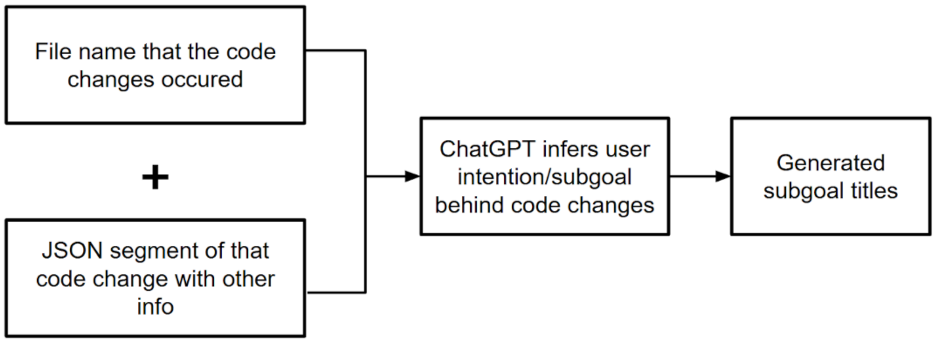
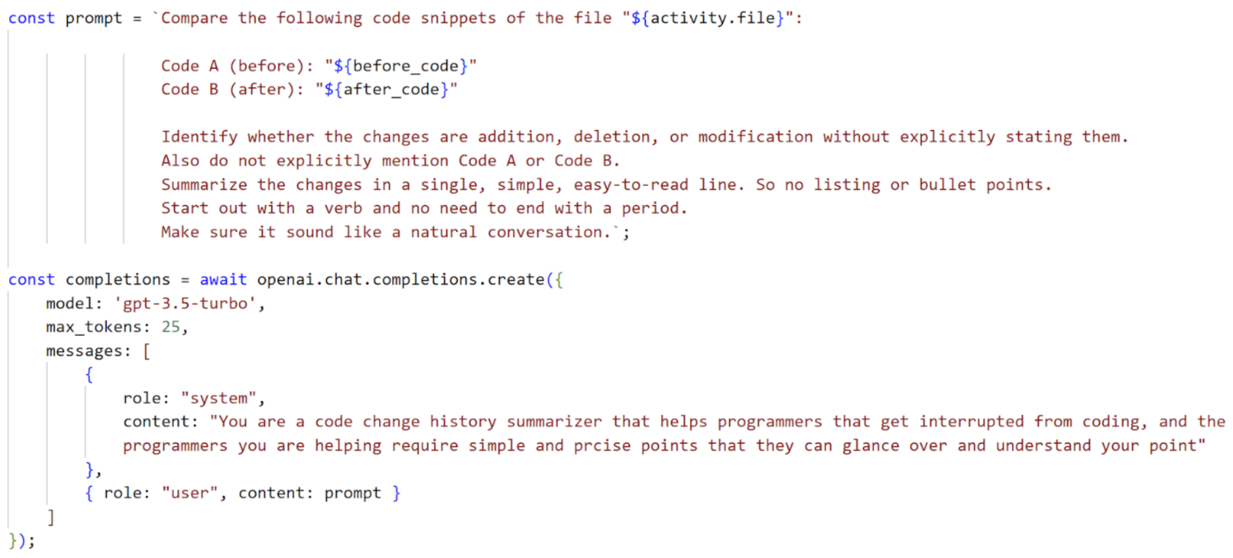

File name that the code changes occured
+
JSON segment of that code change with other info
ChatGPT infers user intention/subgoal behind code changes
Generated subgoal titles

Figure 3.1. Diagram shows the process to get ChatGPT generate subgoals



```
const prompt = `Compare the following code snippets of the file "${activity.file}":

                Code A (before): "${before_code}"
                Code B (after): "${after_code}"

                Identify whether the changes are addition, deletion, or modification without explicitly stating them.
                Also do not explicitly mention Code A or Code B.
                Summarize the changes in a single, simple, easy-to-read line. So no listing or bullet points.
                Start out with a verb and no need to end with a period.
                Make sure it sound like a natural conversation.`;

const completions = await openai.chat.completions.create({
    model: 'gpt-3.5-turbo',
    max_tokens: 25,
    messages: [
        {
            role: "system",
            content: "You are a code change history summarizer that helps programmers that get interrupted from coding, and the
            programmers you are helping require simple and prcise points that they can glance over and understand your point"
        },
        { role: "user", content: prompt }
    ]
});
```

Figure 3.2. Prompt given to ChatGPT API enabling dynamic subgoal generation

Upon receiving a response, the system displays the generated title in the interface as a subgoal label. Users have the option to manually revise the subgoal title if they find it unclear or inaccurate. This flexibility allows the summary to evolve as needed while still reducing the cognitive burden of writing descriptions manually. The design supports seamless integration into the programmer's workflow without interrupting the coding process.

3.3 Summary Generation Evaluation

To evaluate the reliability and effectiveness of the summary generation function, we conducted a comparative study involving two large language models: ChatGPT and Gemini. The objective was to determine whether these models could generate accurate and helpful subgoal titles based on real examples of code editing history.

For this study, we selected 20 pairs of code changes sampled from a pre-existing code history documentation file formatted as JSON. Each entry contained a "before" and "after" code segment along with a manually written subgoal title that served as the ground truth. Both ChatGPT and Gemini were asked to generate subgoal titles for the same 20 examples using exactly the same prompt format. We also used the pre-existing JSON file with ready summarized subgoal in it as a baseline for accuracy.

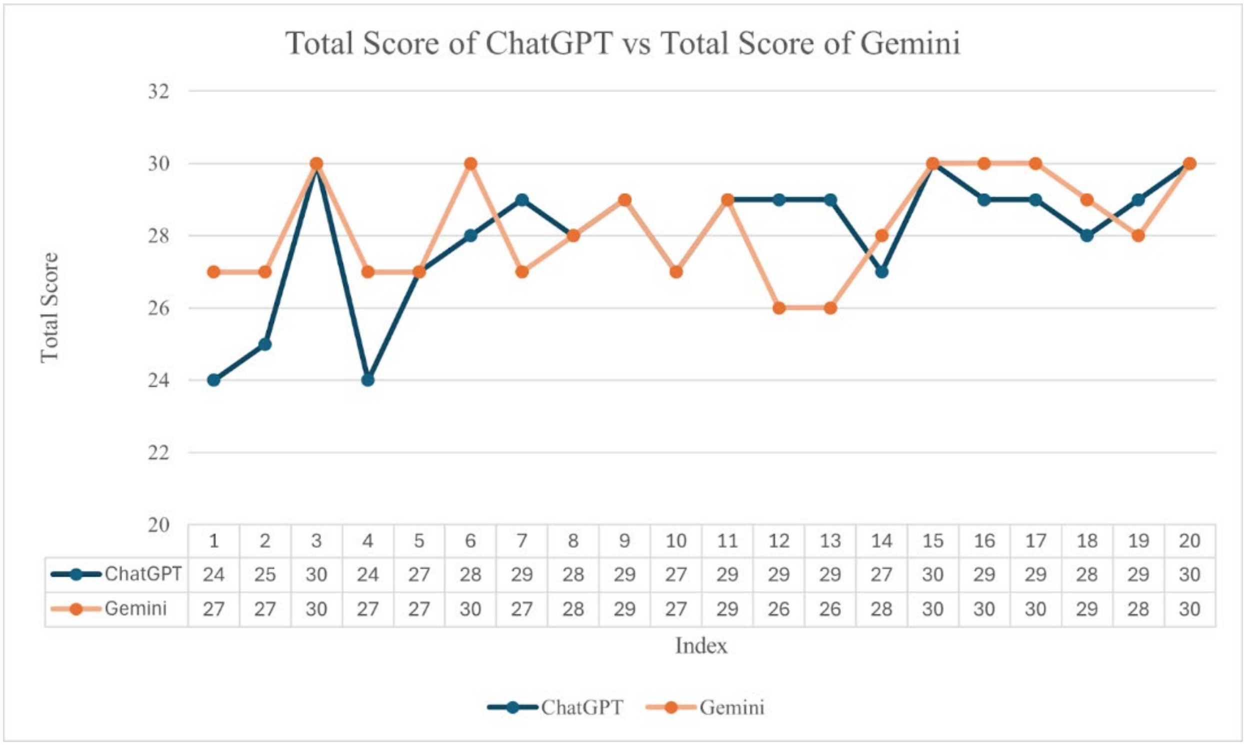
For each LLM, we generated 20 subgoal titles and evaluated them based on three criteria:

1. Accuracy: How closely the generated title matched the original subgoal title.
2. Readability: How easy it is to read the generated title.
3. Preciseness: Whether the title correct and sharply defined.

Each generated title was scored from 1 to 10 on each of these criteria, with higher scores indicating better performance. The scores were human rated. In addition to three category ratings, we also considered a metric called Overlap Accuracy, which measures the overlapping words between the generated title and the original title. We calculated word-level overlap-based accuracy using the formula:

$$\text{Overlap Accuracy} = \left(\frac{\text{Total Overlapping Words with Original}}{\text{Total Words Generated}}\right) * 100$$

This measure allowed us to evaluate how closely the generated title aligns with the original subgoal in terms of content, without requiring human grading the 3 categories, which can contain biases.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ChatGPT	24	25	30	24	27	28	29	28	29	27	29	29	29	27	30	29	29	28	29	30
Gemini	27	27	30	27	27	30	27	28	29	27	29	26	26	28	30	30	30	29	28	30

Figure 3.3. Final score of the 2 LLMs

	ChatGPT	Gemini
Total Words	367	273
Overlapping Words	164	112
Overlap Accuracy	44.69%	41.03%
Total Accuracy	9.25 / 10	8.90 / 10
Readability	9.80 / 10	9.85 / 10
Conciseness	8.95 / 10	9.50 / 10

Final Score (sum)	560	565
Average per Item	28.0	28.25

Table 3.1. Table with final score combined with overlapping score

The combination of subjective ratings and objective overlap metrics provided a more holistic evaluation of how well each language model performed, as shown in Table 3.1. In the overlapping accuracy category, we observe that the overlapping scores for both models are relatively low. This is due to both LLMs having generated more detailed summarizations compared to the original summary. As illustrated in Table 3.2, both ChatGPT and Gemini generated much more detailed summaries than the original ones. This can be both beneficial and detrimental as the search function will certainly benefit from a summary with more details, but it will negatively impact user readability. The impact of longer generated subgoals should be considered in future studies.

Pre-existed subgoal summary	ChatGPT	Gemini
Cleaning up build update letters function	Removed some logging and made the letter update directly affect the dataset property of the currentGuess element.	Removed a global variable declaration for \`currentLetters\`, and inline its usage within \`updateLetters\` function, also removed some console logs

Table 3.2 Table with an example demonstrates the difference between the original subgoal, ChatGPT generated subgoal, and Gemini generated subgoal

Based on the final data, we can see that both LLMs are capable of generating useful subgoal titles. There are minor differences between the two models. For example, ChatGPT has a slightly higher accuracy than Gemini, while Gemini is more concise and readable overall. Overall, Gemini received in total a higher score than ChatGPT. However, we can conclude that either

LLM can accurately or precisely generate a subgoal title. Ultimately, we have decided to proceed with the ChatGPT API due to its slight advantage in accuracy.

3.4 Discussion

The implementation of summary generation shows potential in helping programmers answer hard-to-answer questions about their code. By automatically generating summaries in natural language that transform lower-level raw data into more understandable sentences, we open an opportunity for the user to more quickly grasp the intent behind each code edit without needing to establish an understanding through raw code. This helps programmers understand the intention of a change in past edits more efficiently and accurately. The summaries act as accessible annotations that can guide users through their coding process, especially when they are trying to recover from interruptions or during moments of confusion.

Chapter 4: Supporting Hard-to-Answer Questions

Chapters 2 and 3 focused on improving how code editing history is captured and summarized; together, they provided a key basis for the search function implementation. Now we have an interface that is clear and informative, and we can dynamically generate summaries that better convey what the user is trying to do in each subgoal. But we still have the problem of what happens when the user continues to work on their project and eventually the list of subgoal summaries gets incredibly long. It can be difficult to connect each summary to a particular question the user wants to answer. This chapter seeks to enable programmers to ask questions and then be able to see relevant parts of history.

4.1 Function Goal

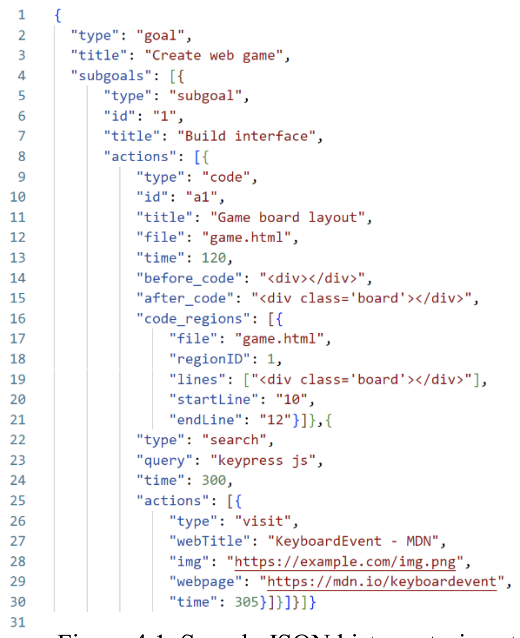
IDEs often include history recording features. However, they do not allow for natural language search within the recorded history. As a result, programmers are often forced to dig through pages of commit logs in order to find a specific code edit. On top of that, programmers often commit infrequently so a single commit may consist of a week or even weeks' worth of code edits [13]. In *Maintaining Mental Models: A Study of Developer Work Habits* by LaToza et al., it was discovered that contrary to popular belief, historic documentation does make understanding design rationale easier [4]. Therefore, we aim to present relevant historical changes done when questions arise.

As the user continues to edit their code, the list of subgoal summaries will also grow quickly. Oftentimes, programmers resort to manually digging through their commit history logs. However, this can be inefficient when the commits are often, or the project is large.

To present the relevant information while addressing the issue of a fast-growing list of subgoals, we will add an additional function to this interface that allows programmers to view the relevant parts of a fine-grained history so that they can understand what happened. In addition to summarizing accurate and meaningful subgoals, the interface will utilize the subgoal summaries to sift out relevant information when programmers ask questions in natural language about their development history.

4.2 Technical Difficulties

Incorporating LLMs like ChatGPT and Gemini into the interface introduced a number of technical challenges. While the intention was to improve user experience and code comprehension efficiency, doing so required providing the LLM with sufficient contextual information without exceeding token limits.



```
{
  "type": "goal",
  "title": "Create web game",
  "subgoals": [{
    "type": "subgoal",
    "id": "1",
    "title": "Build interface",
    "actions": [{
      "type": "code",
      "id": "a1",
      "title": "Game board layout",
      "file": "game.html",
      "time": 120,
      "before_code": "<div></div>",
      "after_code": "<div class='board'></div>",
      "code_regions": [{
        "file": "game.html",
        "regionID": 1,
        "lines": ["<div class='board'></div>"],
        "startLine": "10",
        "endLine": "12"}]},{
      "type": "search",
      "query": "keypress js",
      "time": 300,
      "actions": [{
        "type": "visit",
        "webTitle": "KeyboardEvent - MDN",
        "img": "https://example.com/img.png",
        "webpage": "https://mdn.io/keyboardevent",
        "time": 305}]}]}]}
```

Figure 4.1. Sample JSON history storing structure

A key challenge involved the structure and size of the JSON used to store the development history. This JSON file, which included subgoal summaries, code edits, timestamps, web resources, and other metadata, became extremely large after only a few user actions. As shown in Figure 4.1, even a single entry containing one code change and one accessed web resource results in a considerable amount of data.

Early testing revealed that if the entire JSON file was stringified and passed to ChatGPT to generate a response where we expected a raw JSON response with the same structure from ChatGPT, the model would either exceed the token limit and return an error or sometimes take up to five minutes to respond. This delay made the feature impractical for real-time use.

Another issue was identifying which portion of the recorded history to pass to the LLM. Different user questions required different types of context: some were about previously accessed web resources, while others asked about the evolution of the codebase. Without knowing which part of the recorded history was relevant, we risked either omitting critical information or overwhelming the model with unnecessary data.

These problems highlighted the need for a more selective, dynamic prompt generation process that could determine the intent of a user's question and adjust the input data accordingly.

4.3 Final Implementation of Dynamic Searching Feature

To provide targeted answers based on user questions, we designed a two-stage LLM pipeline using both Gemini and ChatGPT. This setup ensures that prompts include only the most relevant data, minimizing delays caused by excessive input. Referring to the hard -to-answer questions compiled by LaToza et al. [2], we hypothesize that user questions typically fall into two categories: those related to code editing history and those regarding previously accessed web resources, as illustrated in Figure 4.2.

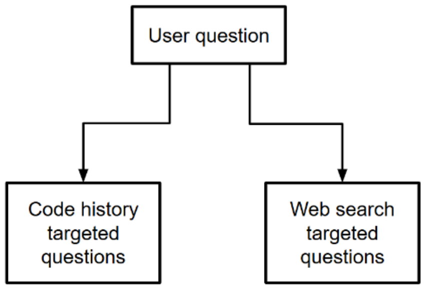
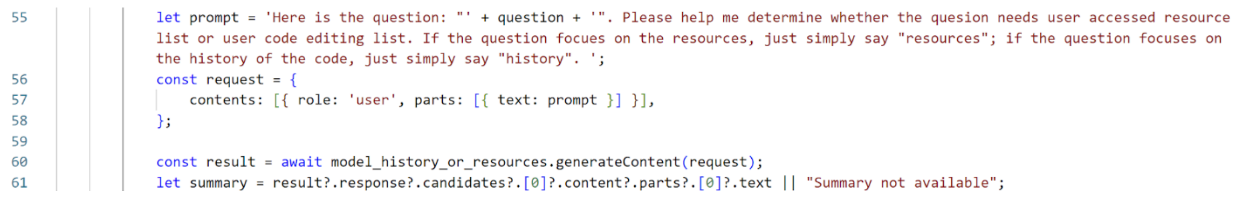

User question
Code history targeted questions
Web search targeted questions

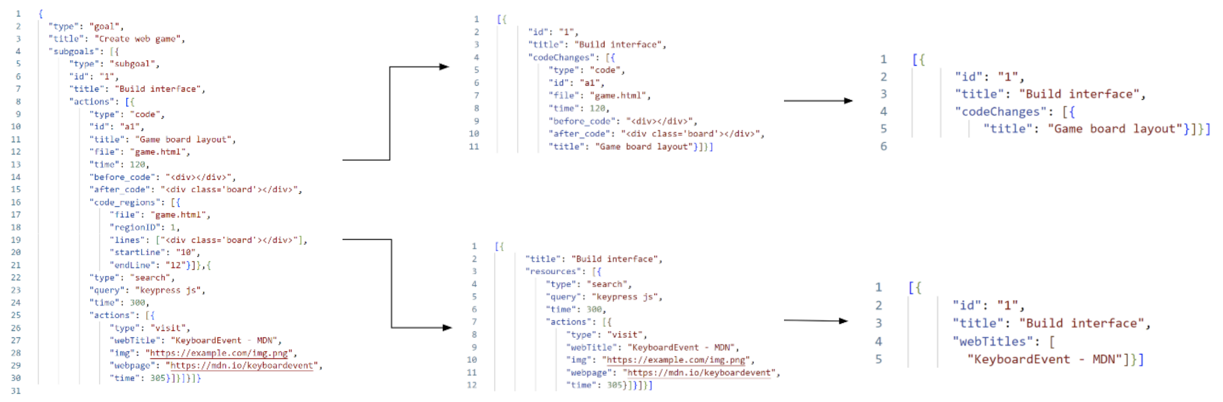
Figure 4.2. Flow chart showing the two types of potential user questions

The first stage of the pipeline uses Gemini to classify the question. As shown in Figure 4.3, we send a prompt asking Gemini to respond with either "resources" or "history." A response of "resources" indicates that the user is inquiring about web documentation or visited pages, while "history" refers to past code edits. This classification guides the next step in the pipeline.



```
let prompt = 'Here is the question: "' + question + '". Please help me determine whether the quesion needs user accessed resource list or user code editing list. If the question focues on the resources, just simply say "resources"; if the question focues on the history of the code, just simply say "history". ';
const request = {
    contents: [{ role: 'user', parts: [{ text: prompt }] }],
};

const result = await model_history_or_resources.generateContent(request);
let summary = result?.response?.candidates?.[0]?.content?.parts?.[0]?.text || "Summary not available";
```

Figure 4.3. Prompt given to Gemini API to identify whether user questions is about their own code edits or online research histories



```
{
  "type": "goal",
  "title": "Create web game",
  "subgoals": [{
      "type": "subgoal",
      "id": "1",
      "title": "Build interface",
      "actions": [{
          "type": "code",
          "id": "a1",
          "title": "Game board layout",
          "file": "game.html",
          "time": 120,
          "before_code": "<div></div>",
          "after_code": "<div class='board'></div>",
          "code_regions": [{
              "file": "game.html",
              "regionID": 1,
              "lines": ["<div class='board'></div>"],
              "startLine": "10",
              "endLine": "12"}]},{
          "type": "search",
          "query": "keypress js",
          "time": 300,
          "actions": [{
              "type": "visit",
              "webTitle": "KeyboardEvent - MDN",
              "img": "https://example.com/img.png",
              "webpage": "https://mdn.io/keyboardevent",
              "time": 305}]}]}]}
```

```
[{
        "id": "1",
        "title": "Build interface",
        "codeChanges": [{
            "type": "code",
            "id": "a1",
            "file": "game.html",
            "time": 120,
            "before_code": "<div></div>",
            "after_code": "<div class='board'></div>",
            "title": "Game board layout"}]}]
```

```
[{
        "id": "1",
        "title": "Build interface",
        "codeChanges": [{
            "title": "Game board layout"}]}]
```

```
[{
        "title": "Build interface",
        "resources": [{
            "type": "search",
            "query": "keypress js",
            "time": 300,
            "actions": [{
                "type": "visit",
                "webTitle": "KeyboardEvent - MDN",
                "img": "https://example.com/img.png",
                "webpage": "https://mdn.io/keyboardevent",
                "time": 305}]}]}]
```

```
[{
        "id": "1",
        "title": "Build interface",
        "webTitles": [
            "KeyboardEvent - MDN"]}]
```

Figure 4.4. JSON simplification process

As illustrated in figure 4.4, the complexity of the JSON structure, even though it is very comprehensive, can lead to significant delays in the response time of the ChatGPT API. This complexity comes from the nested hierarchy and inclusion of information such as overall goal, overall title time, image, etc. These delays undermine the purpose of the interface, which is to quickly present relevant information without requiring the user to manually sift through history. To address this, we introduced a preprocessing step that filters and separates the JSON into two concise, information-dense arrays: one for code changes organized by subgoal, and another for web search history, Figure 4.4 clearly demonstrates the “splitting” process. This design supports the earlier classification step, which labels each user query as either "history" or "resources." Based on this result, only the relevant array is passed to ChatGPT, reducing response time and improving usability. However, we had decided to filter further as separating the code changes history and web search history indeed decreased the time took to generate result JSON, but the time to generate result is still longer, sometimes creating illusions that the search returned no results.

Therefore, we have decided to further filter the two separate arrays, as illustrated in Figure 4.4. The already split two arrays will be filtered into a shorter array containing only vital information such as id and title. This way, we are utilizing and relying solely on the generated title to filter useful information. As demonstrated in Chapter 3.3, the accuracy of the generated summary has shown that it can be trusted. Therefore, we expect the filtering process done here will also be trustworthy, as we are only passing in the summaries generated.

To construct an effective and context-aware prompt, we include these two components for the ChatGPT API call, as shown in Figure 4.5:

1. The question asked by the user
2. The relevant array of filtered information based on the classification done by Gemini

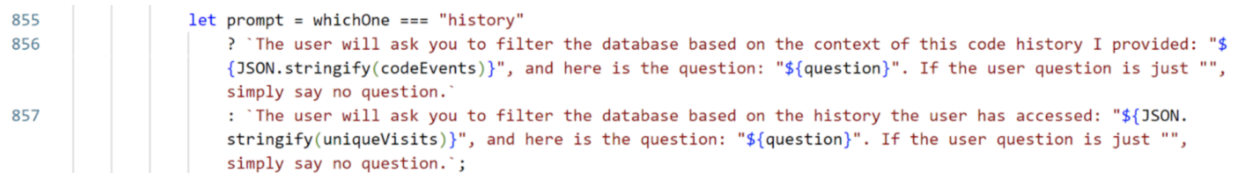

```
let prompt = whichOne === "history"
    ? `The user will ask you to filter the database based on the context of this code history I provided: "${JSON.stringify(codeEvents)}", and here is the question: "${question}". If the user question is just "", simply say no question.`
    : `The user will ask you to filter the database based on the history the user has accessed: "${JSON.stringify(uniqueVisits)}", and here is the question: "${question}". If the user question is just "", simply say no question.`;
```

Figure 4.5. Prompt given to ChatGPT to filter out unrelated information base on the user questions

Historical subgoals and accessed web search history are passed in as a part of the prompt along with the question. The overall process of the LLM pipeline is shown in Figure 4.6. It illustrates the different routes that the user question will take in order to get fast and accurate results.

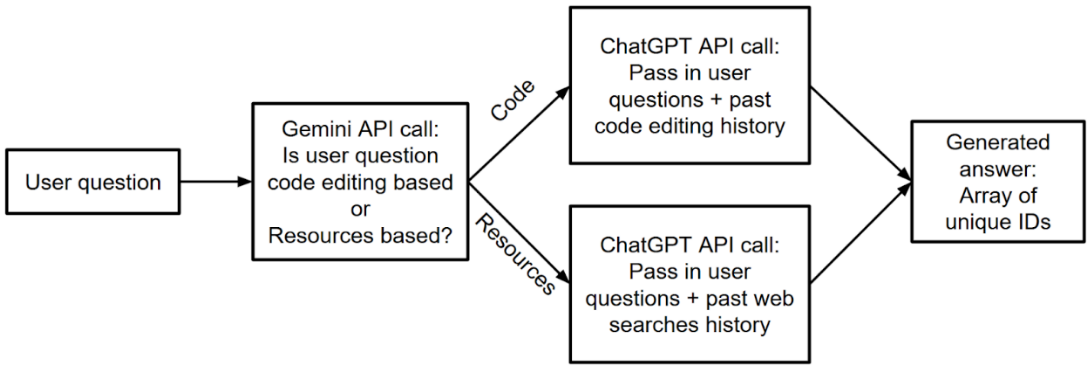

User question
Gemini API call: Is user question code editing based or Resources based?
Code
Resources
ChatGPT API call: Pass in user questions + past code editing history
ChatGPT API call: Pass in user questions + past web searches history
Generated answer: Array of unique IDs

Figure 4.6. LLM pipeline

This way, ChatGPT will be able to generate and display a specific portion of the code change history with the context of all the code editing histories that the programmer has made. With this function implemented, the programmer can access part of the history without having to go through the histories one by one and check whether it is the history that they are looking for. Being able to look up a specific part of history and search history allows the programmer to better interpret the code, thereby helping programmers answer questions that previously could not be answered.

4.4 Future Formative Studies

To determine whether users will be able to use this function and whether ChatGPT will have enough information to accurately identify which parts of history are relevant for answering hard-to-answer questions, we will propose a formative study.

For this formative study, we propose the following procedure:

1. Codebase: The codebase will be an interactive game like Wordle clone. Codebases we can consider are Flappy bird, Connect Four, or Hangman.
2. Method
    a. The participant will be given the codebase and a list of tasks they are expected to work on. The interface they will be interacting with will be the final developed interface with all the functionalities presents.
    b. The interface will be preloaded with a long list of recorded histories. In this study, the preloaded histories will contain a lot of back and forth with some functions. This is to mimic the real-life scenario where programmers often test and scratch their code. This is also to test whether search function will display the relevant version of the function.
3. Tasks
    a. The tasks will focus on the functions that had many versions in the past. All the past versions should be recorded and present in the interface as part of the historical data. The content of the task will most likely be revert the function back to a specific version of history.
    b. The tasks will also consist of questions where history itself does not necessarily provide answers. For example, a question such as “how would you explain to

your coworker that this is the most efficient way of doing this" or "is this code working correctly" will not have an obvious answer in the recorded history.

This proposed formative study will reveal important information about the limitations of the search function and how far we can push these limits. It will also provide insight into the validity of the search results as the list of subgoals grows longer and as programmers navigate back and forth between different parts of the code.

Chapter 5: Pilot Studies

In order to realistically see whether the interface is ready for user study, we conducted several pilot studies. Using the pilot study results, we will be able to determine whether the tasks are engaging enough for the users to interact with the interface, thereby giving us potential data to work with. There are six pilot studies that have been done for this part of the study. The very first pilot study is vastly different from the other five, but it was equally important.

5.1 Initial Pilot Study Tasks

The first study consists of ten checking problems and two coding problems. The participants needed to navigate through the histories to find out whether the ten functions were implemented or not. The format of the ten function check problems is as follows:

Functional Requirement	PASS/FAIL
Letters are entered on keypresses	[user input]
Backspace removes previously added letter	[user input]
…	…

Table 5.1. Initial pilot study tasks, the participant was asked to determine whether any of the functional requirements were not met

However, through this user study, we found that users usually follow their coding routine without interacting with new tools. The very first pilot study participants did not interact with the interface at all. Instead, they loaded the wordle implementation itself to test the program and see whether the functions were implemented. For the two coding problems, similar problems were discovered. We found that the user was uninterested in interacting with the interface. This pilot made us realize that the pilot studies tasks needed to be altered from now on and we needed to find a way to force the participants to engage with the interface in order to see its potential.

5.2 Updated Tasks

In order to test whether history query helps, we first need our users to interact with the interface. After the first pilot study, we learned from the first study result and changed tasks to be mainly question-based rather than more coding-based. The first pilot study tasks contained questions or coding assignments that do not necessarily require participants to check recorded history. This allows participants to answer questions without using the interface. To prevent this from happening, we altered our task design to require engagement with the interface. Users will not be able to answer the questions without having to use the interface to check the pre-documented histories.

The updated pilot study consists of two sections: a familiarization task and the actual programming problem. Participants are given a Wordle interface where the Wordle is already successfully implemented. The codebase simulates a real-world scenario in which a developer inherits an unfamiliar project. To mirror this context, we provide a chronological development history containing both code edit records and web resource visits, as a programmer might encounter when joining an ongoing project. The codebase includes key files such as index.html, style.css, and script.js, with additional support files available for participants to explore.

The programming questions are as follows:

Familiarity task: Before starting the main programming tasks, participants will complete a short warm-up task designed to help them understand how to navigate the interface and locate specific types of information. Below is the question that was used for the pilot studies:

1. Find out more about promise objects and why they were not used.

Main Programming Tasks: After finishing the familiarity tasks, participants will move on to the main questions, which require deeper code comprehension.

1. In which file does the original project use a regular expression to detect whether a user entered a valid letter? What is the final form of the regex?
2. In which file is the flip animation for submitted letter tiles implemented? How is the timing of the animation controlled or modified?
3. In what file did the developer choose a solution word at random from a list, and what method did they use to select it?
4. How was tile color logic being implemented? Did they use a different method in the past? Was it successful? If it is, what logic did they use? If not, what did they use instead?
5. Why was the switch statement removed in favor of a different logic approach?
6. What edits were done for the clean up?
7. Was promise used? If it was used, copy the code down. If it were to be used, what is your implementation of it? You can test it in the code.
8. How can I mimic the key pressing function to implement a hint function?

Questions 1–3 are naïve questions where users should be able to find answers quickly since they are mostly function or variable locating questions. They are a baseline for the later part of the questions where they increase difficulties. Questions 4–6 are formatted to reference the questions mentioned in LaToza et al.'s paper. They either require a degree of understanding of the code or sift through the long, pre-loaded histories. Questions 7–8 are implementation problems mimicking a real-life scenario in which the participants are given a programming assignment to complete. Together, these tasks reflect a range of challenges developers face and help us evaluate whether the LLM powered interface effectively supports users in answering both simple and complex questions about their code.

5.3 Pilot Study Results

The results of the pilot study raised important questions about both task design and user behavior. While the tasks were created to evaluate the system's ability to support natural language questions using a large language model, we found that participants consistently relied on keyword-based searches. This happened even after they were told that natural language queries were fully supported. One possible explanation is that the tasks were too specific, which made it easy for participants to locate answers using just keywords rather than full questions. This behavior contradicts our initial hypothesis that user questions could be meaningfully categorized into two distinct types: those about code history and those about web resources. Since participants rarely phrased their input as full questions, that classification did not hold in practice. These findings indicate that both the task instructions and the interface may need to better guide users toward natural, question-based interaction.

The results indicate that we will need additional pilot studies and possibly make changes to the interface in order to get to a testable behavior.

5.4 Discussion

The pilot study provided important information on how users interact with the LLM-integrated search feature and the history-based interface. While the technical implementation functioned as intended, user behavior significantly deviated from our expectations. Most notably, despite explicit instructions and demonstrations highlighting that the interface can support natural language questions through LLM APIs, all participants defaulted to entering short keyword searches rather than full-sentence questions. This may also indicate that the interface does not respond with LLM-like answers, confusing the participants about how the search function works. This behavior raised an important problem. It deviates from the foundational assumption behind our LLM API pipeline, which depends on detecting user intent through sentence-level question classification. Since users rarely entered complete questions, our system's classification stage became ineffective, as the Gemini model was unable to reliably distinguish intent from vague or ambiguous keywords.

Another problem we discovered is that a person's familiarity with coding and the specific programming language very much determines how well they perform. One of the pilot study participants was not a Computer Science student, which resulted in their question-answering speed being dramatically different from the rest.

5.4.1 Modifications for Future Studies

To prepare for the actual user study, we should consider the following modifications that attempt to resolve the problem of users not treating the search function as LLM powered:

- Redesign user instruction: Future participants will be given more active and structured familiarity tasks that demonstrate example searches and encourage full-sentence input.
- Consider interface redesign for Search Input: Adding prompts or placeholder text such as "Why did I do this…" could help reframe the search box as a conversational input rather than a keyword field.

To address the problem of the user's programming ability being hard to quantify and compare, we should consider the following to balance the ability between individuals:

- Consider giving each participant two different tasks, where one uses the original interface and the other uses the redesigned interface. We should compare the participant's speed and accuracy between the two tasks, rather than having two groups of participants, as it is difficult to control everyone's ability at a similar level.

While the pilot studies challenged our initial assumptions, these insights are critical for refining both the interface and the methodology. With revisions to the interface and participant guidance, the study can be improved significantly. Once these changes are implemented and re-tested, the system should be better prepared for a full user study focused on evaluating LLM-assisted code history exploration.

Chapter 6: Proposed Evaluation

Creating an interface where programmers' subgoals are summarized and queried by ChatGPT is not enough to definitively demonstrate that the interface improves programmer's ability to answer hard-to-answer questions. If we were to evaluate the effectiveness of the interface in real-world scenarios, we need to conduct user studies. As we improve and learn from pilot studies, we can proceed with the proposed evaluations to determine the interface's overall usefulness.

6.1 Hypothesis

We hypothesize that both the accuracy and speed of the hard-to-answer questions will both increase as the participants utilize the enhanced interface that supports history-based queries. To test this hypothesis, we will introduce the proposed interface evaluation in the following sections.

6.2 Demographic

This research will call for one group of participants. Each participant will go through two phases in the study: the controlled phase and the experiment phase. Different from the pilot study, we decided to compare the personal differences when interacting with two different interfaces. We will recruit undergraduate/graduate students with a requirement of decent knowledge in programming. They should have already taken at least three programming courses. All the participants will ideally have a similar amount of experience in coding. We want to mimic the real-life scenario as closely as possible, as the ones using this tool should be people who are in the computer science industry and would benefit from it.

6.3 Method

The goal of this study is to evaluate whether AI-generated inferred subgoals can help programmers answer hard-to-answer questions about the code by helping users more efficiently find and revisit code changes and access resources. To test the hypothesis that the interface will improve users' ability to answer these hard-to-answer questions, the user study is divided into two main sections: the controlled section and the experimental section.

The controlled section will be a programming project where the user will have access to the original interface developed by Parnin et al. It will serve as the baseline condition, helping us measure how users perform without the added support of subgoal summaries or searchable history.

The experimental section will ask participants to complete programming tasks with the same difficulty level, but with access to the enhanced interface. This version of the interface will include the subgoal summary generation feature and the history searching feature. The goal here is to evaluate whether the enhanced interface better supports programmers, particularly in answering hard-to-answer questions about the code. If our hypothesis is correct, the participants will perform better in the experimental section.

The overall procedure of the study will follow a within-subjects design. We will have the prepared user study ready on the school computer. The participants will then be asked to work on the first given code base. There will be a set of coding questions followed by related coding exercises. The participants will have an hour to work on the tasks. The interface given will also have a list of pre-existing histories, creating the scenarios where the participants are getting the code base from another programmer without information on the reasoning behind how the code was written. Participants are highly encouraged to express their thoughts out loud, since we may want to record that as additional information. As they edit more on the given code base, the extension will record every code edit they perform and combine in-progress edits into subgoals at appropriate times. As the list of combined subgoals grows, we will ask the participants different types of history-related questions about the edits done by supposed previous programmers and even their own past edits. After task one is completed or the participants reach the time limit, we will give them a quick break while we set up the second portion of the test where a different interface is used. If the participants are given the original interface first, the second task will use the updated interface, and vice versa.

Lastly, we would like to invite the participants for a quick interview to ask them about their experience in this study, what was beneficial, and what left frustrations.

6.4 Task Design

As mentioned earlier, the proposed user study will consist of two parts. One part will remain similar to the pilot study, where the user is given the Wordle code base. The questions remain largely unchanged, but we will adjust the familiarization tasks. The familiarization tasks mentioned below will be suitable for both parts, as we want to rotate the order of code bases and which interface it comes with. In other words, participants could get the same code bases with different interfaces. However, all the participants will be able to use both interfaces, it is just a matter of order and which codebases they got paired with.

The familiarization task for the original interface will go as follows:

1. Please add animations of your choice to the wordle/2048 header.
2. Find the recorded history of your animation implementation on the interface.

The familiarization task for the improved interface will go as follows:

1. A colleague is trying to clean up a module but notices an unusual condition inside *function A*. They are wondering what problem this condition was originally added to fix, and is it still needed? Use the code history to help answer their question.
2. A colleague wants to know why *function A* weren't used in implementing *feature B*, use the code history to find out what you can to answer their question.

The second part of the study will be a brand-new codebase. A codebase for the classic game 2048 clone will be given to the user.

1. While reviewing the code, you notice that the developer used a complex solution instead of a more straightforward one. You're trying to understand: Why did the developer choose this specific approach rather than using a different method to achieve the same result?
    a. Rationale question types mentioned in LaToza et al.
2. You're trying to modify and enhance a feature and come across a helper function *function A* with no documentation. To decide whether you can safely modify it, you ask: What does *function A* do and how does it contribute to the overall goal of the program?
    a. Intent and implementation questions mentioned in LaToza et al.
3. While running the program, you notice several warnings on the front end. To investigate further, you ask yourself: Is this program bug-free? If not, what is causing the bug, and has anyone attempted to address it before?
    a. Debugging questions mentioned in LaToza et al.

4. To improve overall loading time and to impress your boss, you're reviewing functions that could be optimized. You identified *function A* and wonder: What changes can be made to optimize the code without affecting the behavior of existing users?
    a. Refactoring questions mentioned in LaToza et al.

The questions above are designed to reflect the types of real-world questions that programmers often ask, as identified in LaToza's research. By providing participants with opportunities to actively work with these question types, we aim to evaluate whether the enhanced interface can actually support programmers in answering such questions.

6.5 Data

There will be three types of data that we will record: user input data, video data, and interview data. There will also be two different user input data we will record in this case, the user's code history and their responses to the programming questions. The participants' code history data will be documented through screencast recordings, code editors, and the code history extension. The programming question response will be documented in a .txt document. The sections of code that the participants wrote during the testing will be recorded along with the success/fail output that the code produces. The interface will record all the code edits done by the user, and the recorded data will then be saved in a JSON file. The Chrome extension will allow users to cite any web pages that they find helpful. The data that will be recorded are as follows: web page links, actions such as opening a new link and navigating between URLs within the same tab through linking and reloading. The extension will not be recording the content of the page that the participants are looking at, but it will record the input string participants used to search for certain information. Video data will be a recording of the whole process where the participants try to solve the coding prompt that was provided to them. We will conduct an additional interview with the participants the day after the test is performed. We will take notes as we chat and discuss their experiences, including their thoughts on both the programming exercises they completed and the helpfulness of the interface.

Chapter 7: Conclusion

This thesis research investigated how programmers can better recover context and answer hard-to-answer questions by utilizing code editing histories, web browsing histories, and AI-generated subgoal summaries. The study focused on designing an interface that helps programmers better understand both their own code and others' by recording and organizing historical information in a more meaningful way. This research also highlights that traditional commit logs and comments are not sufficient in conveying the reasoning or thought process behind code decisions, and they rarely provide a clear picture of how the code has evolved over time.

The key contribution mentioned in this paper includes a transformation of the original LaToza et al. interface to an AI-powered prototype interface that dynamically stores code editing information with generated inferred user subgoals and the ability to search within the history. Along with the changes mentioned, a potential user study is also proposed.

One of the key creations in this research is the use of the ChatGPT API to dynamically infer and summarize user subgoals based on sequences of code edits. This function frees programmers from having to manually commit their edits and write their own commit messages. The automatic summary ensures that no information is forgotten.

Another key function created in this research is the ability to query within a user's own code editing history. What is unique about this project is that we try to use recorded history alone to attempt to answer those otherwise difficult questions.

A proposed user study outlines how this system could be evaluated for its effectiveness in helping programmers comprehend unfamiliar codebases and reconnect with their own past decisions.

Future work will include conducting the user studies and analyzing the results. Another potential direction for future work will be to improve the searching function considering that users might not use natural language. We can investigate how searching with phrases can change the searching function. For example, if a user searches with a phrase or a single word, AI filtering might not be needed compared to asking questions in natural language where AI can be necessary.

As we get results from the user studies once we conduct them, we will continue to improve the interface functionality based on the feedback given by the user study participants.

References

[1] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, “How Do Software Engineers Understand Code Changes? - An Exploratory Study in Industry”.

[2] T. D. LaToza and B. A. Myers, “Hard-to-answer questions about code,” in *Evaluation and Usability of Programming Languages and Tools*, Reno Nevada: ACM, Oct. 2010, pp. 1–6. doi: 10.1145/1937117.1937125.

[3] A. Begel and T. Zimmermann, “Analyze this! 145 questions for data scientists in software engineering,” in *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad India: ACM, May 2014, pp. 12–23. doi: 10.1145/2568225.2568233.

[4] T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: a study of developer work habits,” in *Proceedings of the 28th international conference on Software engineering*, Shanghai China: ACM, May 2006, pp. 492–501. doi: 10.1145/1134285.1134355.

[5] A. J. Ko, R. DeLine, and G. Venolia, “Information Needs in Collocated Software Development Teams,” in *29th International Conference on Software Engineering (ICSE’07)*, Minneapolis, MN, USA: IEEE, May 2007, pp. 344–353. doi: 10.1109/ICSE.2007.45.

[6] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, “Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?,” in *ECOOP 2012 – Object-Oriented Programming*, vol. 7313, J. Noble, Ed., in Lecture Notes in Computer Science, vol. 7313. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–103. doi: 10.1007/978-3-642-31057-7_5.

[7] V. T. T. Pham and C. Kelleher, “Code histories: Documenting development by recording code influences and changes in code,” *J. Comput. Lang.*, vol. 82, p. 101313, Mar. 2025, doi: 10.1016/j.cola.2024.101313.

[8] M. Wyrich, J. Bogner, and S. Wagner, “40 Years of Designing Code Comprehension Experiments: A Systematic Mapping Study,” *ACM Comput. Surv.*, vol. 56, no. 4, pp. 1–42, Apr. 2024, doi: 10.1145/3626522.

[9] E. Agrawal *et al.*, “Code Compass: A Study on the Challenges of Navigating Unfamiliar Codebases,” May 10, 2024, *arXiv*: arXiv:2405.06271. doi: 10.48550/arXiv.2405.06271.

[10] A. Horvath, A. Macvean, and B. A. Myers, “Meta-Manager: A Tool for Collecting and Exploring Meta Information about Code,” in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, Honolulu HI USA: ACM, May 2024, pp. 1–17. doi: 10.1145/3613904.3642676.

[11] J. Allen and C. Kelleher, “Exploring the impacts of semi-automated storytelling on programmers’ comprehension of software histories,” in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Liverpool, United Kingdom: IEEE, Sep. 2024, pp. 148–162. doi: 10.1109/VL/HCC60511.2024.00025.

[12] C. Parnin and R. DeLine, “Evaluating cues for resuming interrupted programming tasks,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Atlanta Georgia USA: ACM, Apr. 2010, pp. 93–102. doi: 10.1145/1753326.1753342.

[13] M. Codoban, S. S. Ragavan, D. Dig, and B. Bailey, “Software history under the lens: A study on why and how developers examine it,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Bremen, Germany: IEEE, Sep. 2015, pp. 1–10. doi: 10.1109/ICSM.2015.7332446.