DevTales: A Tool for Providing Narrative Code Histories into Developer Workflows

John Allen
McKelvey School of Engineering
Washington University in St. Louis
Saint Louis, USA
johnjallen@wustl.edu

Somin Park
McKelvey School of Engineering
Washington University in St. Louis
Saint Louis, USA
mia.p@wustl.edu

Caitlin Kelleher
McKelvey School of Engineering
Washington University in St. Louis
Saint Louis, USA
ckelleher@wustl.edu

***Abstract*—Understanding unfamiliar code often requires insight into the original developer's intentions, design rationale, and on-the-fly problem-solving strategies. Traditional version-control systems capture only coarse commits, leaving web foraging and developer notes disconnected. We present DevTales, a VSCode extension that integrates fine-grained subgoal labels, captured web-foraging activity, and LLM-generated narrative summaries directly with code in the IDE. We evaluated DevTales in three mixed-methods user studies spanning debugging, understanding rationale, and code reuse. Results show that subgoal labels and "See in Code" links enable rapid top-down navigation and code localization, while embedded web resources clarify unfamiliar artifacts. Narrative summaries proved valuable for answering deep "why" questions by explicitly linking causal relationships among history items, but were avoided during quick sensemaking contexts due to their verbosity. We discuss design implications for history-aware code search, adaptive narrative summarization, and inference transparency in LLM-generated stories.**

***Index Terms*—Code History, Software Engineering, Program Comprehension, Storytelling**

I. Introduction

Software systems undergo significant evolution throughout the software life cycle; developers implement new functionalities, refactor and optimize code, and fix bugs. Developers' design intentions are often influenced by emerging problems at the time of implementation and the resources the developer found to overcome these problems. Unfortunately, the reasoning behind these design decisions is often lost, as current code versioning systems do not support historical information at this level of granularity. As a result, software developers often report that when working with unfamiliar code, information about the original developers' intentions is difficult to track down, but helpful for understanding the code in order to complete new software evolution tasks [1].

Research suggests that high-level **subgoal labels** can help programmers grasp the original code author's intentions [2], and that recording the developer's **web foraging** may help future programmers understand the rationale behind specific code fragments [3]. Presenting both of these information sources can help programmers reason about the functionalities within code, but programmers 1) struggle to connect the subgoals and web resources to the code implementation itself and 2) experience additional cognitive load [4]. Presenting this context in narrative form may reduce this additional strain [5], however, it remains unclear when and how programmers practically use narrative-formatted code histories during software evolution tasks.

To address these gaps, we introduce DevTales, a VSCode extension which builds upon these findings by 1) bidirectionally connecting programmer subgoals and web resources to code and 2) utilizing storytelling to convey this historical and contextual information to reduce additional cognitive load. We explore two research questions:

- **RQ1**. How do programmers utilize the subgoals and original web resources of the original developer during software evolution tasks?
- **RQ2**. In what scenarios do programmers turn to the narrative-formatted histories? Why, and how is it helpful?

We conducted three complementary studies to evaluate DevTales across diverse programming scenarios (see Table I):

- **Study 1 (Debugging)** explores DevTales' role in supporting concept→code localization and top-down navigation in familiar domains.
- **Study 2 (Rationale)** focuses on DevTales helping programmers answer questions about rationale behind code.
- **Study 3 (Reuse)** investigates DevTales usage during broader comprehension tasks required for code repurposing in unfamiliar domains.

Our mixed-methods findings highlight that with DevTales, programmers 1) utilize subgoals to anchor top-down program comprehension quickly find relevant code, 2) leverage the web resources of the original developer to understand and debug code, and 3) turn to storytelling when trying to understand the implicit temporal "cause-effect" relationships of code history artifacts.

II. Related Work

Our work builds on research in 1) program comprehension, 2) presenting historical rationale of code, and 3) mitigating cognitive load through subgoal labeling and storytelling.

A. Program Comprehension

Programmers are often inefficient in trying to understand an unfamiliar code base [6]. Program comprehension can occupy

NSF Grant No. 2128128

TABLE I
USER STUDY DEMOGRAPHICS, TASKS, AND CONNECTION TO RQS

	Task Type	Code Base	RQs	Programming Classes (std)	Months Intern (std)	N	BS/MS/PhD
Study 1	Testing/Debug	Wordle Clone	1	7.9 (4.5)	11.9 (15.7)	10	4/6/0
Study 2	History Questions	Python-Mosaic	1,2	6.3 (3.0)	13.8 (13.7)	7	2/5/0
Study 3	Code Reuse	Python-Basketball	1,2	13 (5.7)	9.8 (10.8)	4	1/2/1

more than half of a programmer's development time [7]. While navigating and understanding a program can become a cognitively overwhelming task [8], [9], programmers also must turn to the web and other sources of documentation for additional help [7].

1) Program Comprehension Strategies: Programmers typically blend *bottom-up* and *top-down* strategies to when making sense of code [10], [11], [12]. **Bottom-up** approaches include programmers reasoning about individual lines of code, and building up a mental model of larger purpose from there [13], [14]. In contrast, when programmers employ **top-down** strategies, they scan the software at the highest level of abstraction to understand the code's purpose and organization before delving into low-level details [15], [16]. As Brooks notes, top-down reasoning first requires programmers to understand the problem domain before tracing how developers mapped their solution to code [15].

History-aware tools should support **both** bottom-up program comprehension strategies by linking individual lines of code to related historical development context, and top-down program comprehension by linking context back to individual lines of code [4].

2) Hard-to-answer questions about code: As programmers attempt to understand the nature of a code base, they have many questions [17]. While some can be answered by testing the code or using the web, programmers find that the questions that are "hard-to-answer" involve the code's provenance, history, and rationale [1]. Research in code histories highlights that capturing additional context behind software may help programmers answer these traditionally "hard-to-answer" questions [3]. Our work contributes to this body of research by exploring how programmers utilize intent and history directly embedded into the IDE.

B. Capturing & Presenting Code Histories

Software history typically comes through versioning control, like git commits, which generally do not fully convey the original developer's reasoning or goals [1]. "Tangled commits", where a programmer makes multiple, unrelated code edits before committing their code [18], lead to unrelated edits when being grouped together [19], [20]. This 1) introduces real program comprehension challenges to understanding historical changes, and 2) obscures the intentions behind the code changes, leading to unmet information needs [6], [21]. To overcome these issues, research has focused on capturing and presenting code change information at finer granularity, and labeling code changes with the subgoals of the original developer.

1) Capturing fine-grained code changes: Some research has investigated untangling commits [22], but a more significant portion of this work investigates capturing fine-grained changes developers make while editing code files [20], [23], [24]. These systems typically capture developer actions at the keystroke level, resulting in a large volume of historical actions.

2) Presenting Historical Information: Program comprehension can be cognitively overwhelming to developers [9], [8], and introducing a large swath of historical information can contribute to the cognitive demands of a developer trying to understand the code [25], [26]. Since fine-grained code change history systems capture additional historical edits, they have the additional challenge of presenting this information in a useful way [9]. Grouping related code changes together may help programmers by reducing the amount of information they must process. Yoon provides a way of grouping similar changes at the line, function, and class levels of granularity [26]. Other work groups code changes only when the structure of the program's abstract syntax tree (AST) is modified [27], [28], providing a higher level of abstraction.

3) Subgoal Labels and Linking Context: Subgoal labeling decomposes problems into smaller subproblems, helping learners understand problem-solving approaches in smaller pieces [29]. Understanding the problems solved by software helps programmers better understand the code itself [15]. Providing subgoal labels for code has been shown to help programmers understand the concepts within it [30], understand new concepts more quickly [31], and utilize programming concepts [32].

Resources such as documentation and websites often influence software implementations and design [33], [34]. Linking this additional context has been shown to help programmers answer questions about the origin of software artifacts [3], [4]. Some work investigates linking software and these inspirational resources [33], [35]. Crystalline automatically collects and organizes the websites that developers visit while writing code [36]. Meta-manager gives developers the ability to directly link external sources to code changes [3].

Some work has presented subgoal labels and additional context to programmers working with unfamiliar code [4], [37], [38], but found that programmers had difficulty tying historical artifacts and external code to the code as it existed [4], [38]. To the best of our knowledge, this work is the first to investigate subgoal-labeled code change histories with original developer web traces that are tied directly to code itself.

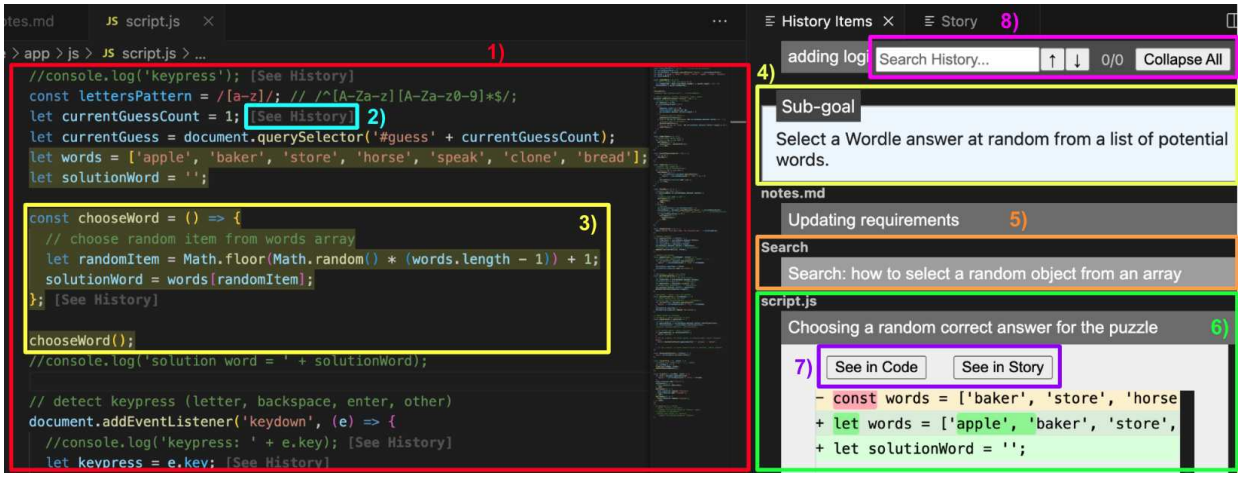

Fig. 1. DevTales Subgoal View: The DevTales Interface includes (1) a text editor on the left-hand side with (2) links from the code to history and (3) subgoal-related code highlighting. On the right, DevTales lists the Subgoals of the original developer (4), their web searches (5), and the code changes (6) the programmer made during that subgoal. Within a code change diff, DevTales provides a pointer to the related code and to where the history item fits into the Story view (7). Finally, DevTales allows users to search (8) through the history items.

C. Storytelling for Code Histories

Program comprehension is a cognitively demanding task [8], and adding additional historical information for developers can add to this overload [25]. Storytelling may have the potential to deliver historical code information with more manageable additional cognitive load.

1) Storytelling to Improve Code History Comprehension: Storytelling has long been used to improve learning of complex materials [39], [40], [41], [42], [43], [44], [45]. People are quicker at reading through information when it is presented in a narrative format [46], and better at recalling information from it [46], [47]. This includes when recalling information about historical code changes [5].

Incorporating finer code changes and contextual clues in code histories can introduce an information overload to already cognitively demanding software evolution tasks [25], and is typically reserved for tutorials [48]. Some recent work has suggested using storytelling to present a code's historical information. Sodalite looks to improve developers' ability to write and keep track of these narrative stories by automatically connecting the text developers write to software elements, and detecting when they go out of date [49]. Wuilmart asked programmers to create narrative descriptions of how they approached and completed programming tasks [50]. This work showed that developers struggled to narrate their experience and do not enjoy the extra documentation [50], [51], [52], suggesting an opportunity for research in the automatic generation of stories to describe code changes.

Although some research has examined the *creation* of code stories [50], [53], [49], very little work has explored their *usability*. One recent study suggests that programmers are better at recalling historical code information when presented in narrative form compared to a traditional list [5]. To our knowledge, no prior work has investigated how programmers actually use narrative-formatted histories in software evolution tasks. Our work uniquely integrates subgoal labeling, web foraging information, and automatically-generated narrative histories directly into the IDE, evaluating their combined effectiveness across diverse evolution scenarios.

III. DevTales: Rationale and Implementation

DevTales is a VSCode extension which embeds the original developers' historical code changes, subgoals, and external resources into the IDE. Building on prior research highlighting the need to reduce cognitive load associated with utilizing information from separate data sources [25], [54], we designed DevTales around three principal design goals:

1) **Two-way code history linking:** Lines of code are annotated with links to the original developer's subgoals and web foraging activity. Conversely, the history items also link back to the related lines of code, if they are still included in the current code state. This supports both top-down (subgoal → code) and bottom-up (code → subgoal) program comprehension strategies.
2) **Narrative presentation:** DevTales generates a *code story* that summarizes historical changes in a narrative format using recent advances in LLMs. While DevTales includes a hierarchical subgoal view that provides a straightforward listing of all history items, the *code story* weaves the historical code changes, subgoals, and web activity into a coherent narrative.
3) **IDE Integration:** DevTales is built into VSCode, so history appears in place.

A. History Capture

Following approaches in [37], [5], [25], DevTales records code state each time the developer tests their code, as well as all web searches and website visits made during development. We then consolidated subsequent, line-overlapping changes and web activity into subgoals, and labeled them each with a

concise one-line summary. For the three code histories created for our user studies, the one-line summaries for code changes and subgoals were labeled manually by either a member of the research team who had watched the entire development process (Study 1) or the code's original author (Studies 2 and 3).

To record the web foraging activity during development, we used the web capture process described by Pham et al [37], which allowed developers to select which browser tabs they wanted recorded or ignored during development, ensuring that only development-related activity was included.

B. History Views

DevTales provides information through two different views.

1) Subgoal View: The Subgoal view, shown in Figure 1, presents historical code changes, web searches, web visits, and labeled subgoals of the original developer as a hierarchical list on the right-hand side of the interface, next to the code editor on the left. When navigating from the history items into the code base, users can scroll through the historical information. Users can browse through the high level subgoals of the original developer and see web searches and code changes made within each subgoal. Both the web search tabs and code change tabs are expandable. Expanding the "search" tabs will reveal all of the subsequent searches and websites the original developer visited when completing that subgoal. Expanding the code change events will reveal the actual git diff of the code state at the time the change was made.

This is where DevTales connects history and code. From a code change, users can navigate to the related code if it remains in the code base, or the code change event in the context of the Story view. When a user clicks "See in Code", or "See in Story" (Figure 1-7), DevTales navigates to the related artifact and highlights it, as shown in 1-3. If the historical code addition no longer exists in the current code state, the "See in Code" button will be replaced with "Changes Removed". Users may also search for historical subgoals, code changes, web searches, and web visits in the floating search bar at the top of the Subgoal view.

2) Story View: Some work suggests that using storytelling to present software's historical information may help programmers process and remember that information [5]. We extend this work by investigating how programmers can benefit from narrative code histories in actual software evolution tasks. So, we incorporated narrative histories into DevTales, as shown in Figure 2, to explore how programmers use this resource in actual programming scenarios. The Story view includes a narrative description of all of the subgoals, code changes, and web searches included in the Subgoal view. From this view, programmers are able to click hyperlinks embedded in the story, and link to either the same history item in the Subgoal view, or the code base itself. Conversely, programmers may also use links in the code editor or the Subgoal view to navigate to relevant elements in the story (Figure 2-3).

The stories themselves we generated using LLMs, similarly to methods used in [5]. However, unlike [5], we do not perform any manual edits to the outputted story for two reasons: 1) recent advances in LLM reasoning models (DevTales uses OpenAI's GPT-o1) are more thorough and accurate in summarizing code histories as stories, and 2) this study seeks to understand the usability of automatically-generated code stories. A sample of a narrative software history is shown in Figure 2, and the full prompt is included in Appendix IX-A.

C. Formative Pilot

We designed an initial VSCode prototype extension through a combination of parallel iterative design and cognitive walkthroughs, and then conducted a formative refinement of DevTales. We recruited 2 pilot participants and 5 students from a large private university who were specializing in Computer Science. We tasked participants with using DevTales to transform a Wordle clone [55] into a memory matching game. We chose this task because it required participants to 1) reason about the core components of the code and how they interact through the logic, and 2) navigate through a breadth of files in order to locate and comprehend these functionalities. This design allowed for robust interaction between participants and the code itself, and created many potential use cases for participants to utilize DevTales. While participants completed this task, we observed how they navigated subgoals, web history, and the narrative summary when understanding and modifying the code. After each test, we performed a semi-structured interview to extract additional insights. We used these observations and insights to shape the final DevTales interface. As problems emerged during formative testing, we updated the interface between user tests to include:

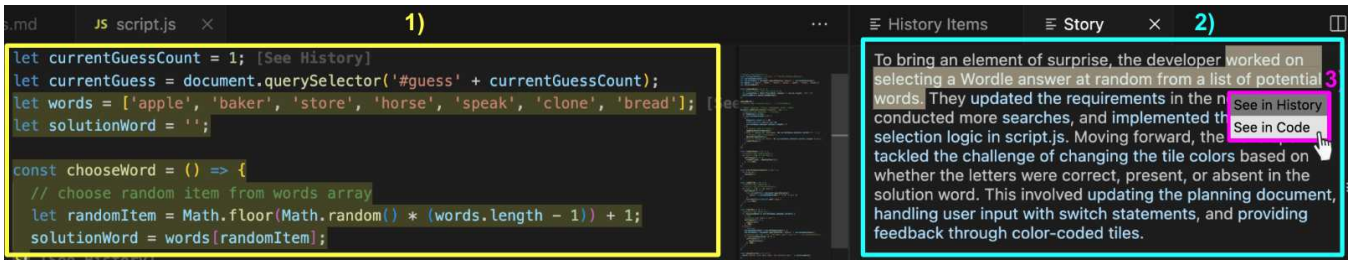

Fig. 2. DevTales Story View: The Story View, attached to the same text editor on the left-hand side (1), provides a narrative summary of all code changes and web foraging activity on the right-hand panel (2). This view also links and highlights code changes to relevant sections of the narrative story (3).

- **Dynamic highlighting** animation to point user attention to relevant code or history item once DevTales navigates to that item.
- **Reactive search feature** that provided instant search results with every key press event in search bar.
- **3-way linking** between the Subgoal view, Story view, and the code itself.
- **Annotations with related subgoal information** when user hovers their mouse over code, allowing them to decide more quickly whether they want to investigate the history further.
- **"Sticky"** property for key interaction mechanisms like search bar and "See in Code" buttons.

IV. Study 1: Testing, Identifying, and fixing a bug

We conducted three user studies, each targeting a distinct software evolution context, to capture how DevTales supports a variety of cognitive demands (see Table I). **Study 1** explored how programmers used DevTales in code testing and debugging scenarios.

A. Task Goals

We provided 10 student programmers (see Table I) with the *Wordle Clone* code base [55] (see Section III-C), as well as its DevTales history. The *Wordle Clone* is comprised of HTML, CSS, and JavaScript. This codebase included four naturally-occurring bugs known to the research team. So, we designed a set of 10 functional requirements to test how programmers use DevTales to address these bugs. We list these functionalities, as well as their expected outcomes in Table II. We asked participants to:

1) Verify each functional requirement is satisfied
2) Identify root issues preventing functional requirement satisfaction
3) Propose how they would change code to fix issues

We chose this code base for its common structure and familiar domain. This project is representative of a simple web-dev project: 1) connecting HTML, CSS, and JavaScript together, 2) handling user input, and 3) including several "setup" functions when the website is loaded. Due to these features, it is likely that the barriers programmers may face when navigating this code base are common to a myriad of other web projects. In addition to increased generalizability, we chose this code base because we want to encourage the high-level reasoning associated with top-down program comprehension, and research shows that programmers are more likely to apply this strategy when working with domains they are familiar with [56]. By including a familiar domain, we believed that participants would spend less time understanding *what the code is doing* and focus more on *how the code functions*.

B. Data Collected

We tracked which bugs participants correctly identified and how they navigated DevTales to find the root cause of found bugs. Upon completion of the user tests, we performed a `diff` between each participant's project files and the original set of files. This pulled out all of the code comments they added that were intended to address functional issues with the software. We identified how many distinct edits each participant made to each file, and then re-watched the user test to determine how each user discovered each code area in which they made their changes. We record if and how participants used DevTales to find relevant software artifacts.

In addition to code changes, we tracked the web sites that participants visited while trying to understand how the code worked and how to fix bugs. We report the frequency in which developers use the same websites as the original developer to understand the code.

Finally, we also record think-aloud data of participants as well as interview responses as qualitative supplements to the quantifiable behaviors, and present our findings using mixed-methods approaches similar to other work in this area [3].

TABLE II
Program functionality tests, their expected results, and the outcome of the participants

Program Functionality Test	Expected Result	N Correct
Letters are entered on keypress	FAIL	1/10
Backspace removes previous added letter	PASS	10/10
Guesses are submitted on Enter	PASS	10/10
Guesses can only include valid 5-letter words	FAIL	5/10
Letters are yellow if present in guess and solution word, but not in correct location	PASS	10/10
Letters are green if present in guess at the correct location	PASS	10/10
Letters are gray if not in solution word	PASS	10/10
A string from the "words" array is selected at random to be the test word	FAIL	6/10
Game completed if word is guessed	FAIL	2/10
Game terminated after 6 incorrect guesses	FAIL	10/10

C. Findings

Our first study addresses **RQ1:** *How Programmers Used Subgoals, Code Changes, and Original Web Resources*. We found that DevTales helped participants 1) map concepts to code rapidly using code-mapped subgoal labels and 2) use the original web resources to understand and fix broken code.

1) DevTales Helped Participants Map Concept to Code: During Study 1, the task eventually required programmers to find the code responsible for found issues and propose a fix. Programmers struggle to navigate an unfamiliar code base for specific functionalities [4], and then when they do find the relevant code, they often have questions about the intentions behind the lines of code themselves [17]. DevTales helped participants find and understand relevant code by facilitating top-down program comprehension and providing additional contextual information for the relevant lines of code.

In practice, participants often skimmed through the DevTales History subgoals to understand what problems were solved by the software, or searched DevTales for key terms, and used the Code Linking feature to find code related to a relevant subgoal. 8/10 (80%) of Study 1 participants used the

code linking feature to locate code responsible for behaviors they were interested in understanding (top-down comprehension). These participants in turn were very effective at quickly finding the relevant code for each bug they encountered, finding the lines of code relevant for the issues they identified as bugs 100% of the time.

Only two participants did not use Code Linking. P1 and P3 generally ignored the code history tool, and the VSCode IDE in general, instead examining the script in the console via Chrome DevTools. Currently, there is no work that researches the use of code history tools embedded into a browser's developer tools, and this surprise finding may point the need for future work to examine this phenomenon.

While 8/10 (80%) of participants used DevTales to find code, finding the relevant code doesn't guarantee an understanding of it. For example, P7 attempted to add a `console.log()` statement in order to print the solution word every time the game was loaded in order to more quickly determine if the solution word was selected randomly each time, instead of having to actually solve the Wordle puzzle each time the page is loaded. He turned to DevTales and entered "word" into the search bar. DevTales scrolled to the code change for "Choosing a random correct answer for the puzzle". He expanded the diff and immediately clicked "See in Code". P7 thinks out loud: "So, we have this list of words [he highlighted the `words` array]... `let solutionWord = chooseWord()`.." He traced the logic saw a call to `Math.random()` in `chooseWord()`, and repeated aloud the requirement being tested: "*All entries have an equal probability of being selected*", before positing "which is kind of guaranteed by the randomness of the selection". It's worth noting that although P7 had the initial goal of printing out the variable responsible for holding the solution word, DevTales led him to code which he did not know existed, that answered his true goal of understanding how solution words were selected. When looking at the lines of code, P7 misses the indexing error in the random word selection, and incorrectly marks the functionality test as PASS. Although P7 did not thoroughly investigate the accuracy of that line of code, DevTales brought the relevant code before the user's attention by effectively mapping the problem domain (ie choosing a word) to the code responsible.

2) Leveraging Original Developer Web Searches: DevTales linked domain knowledge to code by linking visited web resources within the histories. During Study 1, participants frequently used DevTales to understand the objects and logic that the original developer used in the Wordle Clone implementation 6/10 (60%) of the participants visited a webpage that the original developer also visited. Participants turned to these resources when they were trying to understand the code well enough to suggest a fix.

P10 exemplified how DevTales' web-linking was beneficial. After identifying that the Wordle Clone doesn't accept capital letters, P10 turned to the DevTales history items, and found the subgoal related to determining whether a key is a letter. He expanded the first tab to see the web searches that the original developer made when working on this subproblem. Then, he clicked the code change item below and promptly uses the "See in Code" button to navigate to the keypress handling logic. He compares the current code with the code historical code shown in the related DevTales diff. After examining the current code, he determined that there will be an issue if the CAPS lock is on. He hovered his mouse over the various searches the original developer made when implementing this logic, and clicked the link to a Stackoverflow example the original developer visited titled "How to check if character is letter in javascript?". DevTales opened the webpage, and P10 found that the first answer to this question included a RegEx that excludes capital letters. P10 returned to the IDE and scrolled to where the original developer defined their Regex, which included the erroneous Regex from the Stackoverflow answer. He modified the code to include a more inclusive RegEx and tested the code to confirm it worked. Later, when asked about this, he responded by saying "At first, I'm not sure what they use in [the code]. So I just searched through the code history and then found out 'oh actually they use a regex. Let me look at that regex–what they use for the match'. He mentions that when the original developer brought in the regex code example, they did not include the uppercase A-Z, and that is what he fixed.

Interestingly, participants 1 & 3 unknowingly found web resources the original developer used; when understanding and trying to fix the code, they made such similar searches as the original developer that they found the original websites by happenstance. This combined with programmers' DevTales usage suggests the potential for history-aware browser extensions that can suggest or smartly link historical searches to code.

V. Study 2: Answering Hard-to-Answer Questions about Unfamiliar Code

Study 2 explores how programmers use DevTales to answer historically "hard-to-answer" questions [1] about code.

A. Task Goals

When studying unfamiliar code in bottom-up program comprehension, programmers specifically struggle to answer questions about the code's *rationale*, *history*, and *provenance* [1]. Further, when comprehending code in unfamiliar domains, programmers must perform the additional cognitive step of understanding new concepts [56], [57]. So, Study 2 investigates how participants use DevTales to answer these "hard-to-answer" [1] questions about code in an unfamiliar domain.

We tasked 7 student programmers (see Table I) to answer questions about the *rationale*, *history*, and *provenance* of a single-file Python script performing various image-processing tasks to generate a "Photo Mosaic". Drawing on Brooks' argument that programmers must first grasp the problem domain before mapping it into the code [15], we selected a code base with low domain familiarity to encourage participants to focus on *what the code does*. This setup encouraged bottom-up comprehension strategies and allowed us to explore how DevTales helps programmers overcome the additional

cognitive step of mapping new concepts to code [56], [57], [15]. Finally, Python is fundamentally different than web development, allowing us to explore DevTales' applicability across programming frameworks. The high-level logic of the code base is included in Appendix IX-B.

To directly assess programmers' ability to answer questions about the intentions or history behind software artifacts, we created a set of questions similarly to [3]. We list all questions, solutions, and participant outcomes in Table III.

B. Data Collected

As participants worked to answer questions, we recorded their foraging activity through both the code base and the DevTales interface. We report 1) the frequency at which participants correctly answer questions or identify the relevant historical information and 2) how participants used DevTales to collect information in order to answer each question.

C. Findings

During Study 2, we found that DevTales helped participants employ top-down program comprehension on unfamiliar code by connecting abstracted goals to code (RQ1) and programmers turned to story formats to understand rationale and implicit information (RQ2).

1) RQ1: DevTales Facilitated Top-Down Program Comprehension: During Study 2, Participants were often unfamiliar with many of the image-processing concepts included in the python code, and they used DevTales to better understand what the code was doing at a high-level. This allowed the programmers to explore the codebase using top-down program comprehension as opposed to the bottom-up comprehension that programmers usually employ when working in unfamiliar domains. As a result, 100% of participants in Study 2 utilized top-down program comprehension via subgoal navigation, and looked at the code implementation only as a last resort. This was a successful strategy: for 8/11 (71%) of the *hard-to-answer questions about code*, at least 6/7 (85%+) participants used DevTales to successfully find the relevant resource(s) to the question. For the remaining three questions, at least half (4/7) participants correctly found the relevant historical information, as shown in Table III.

2) RQ2: Programmers turned to story formats to understand rationale and implicit information: When answering historically *hard-to-answer* questions about code during Study 2, participants valued the story format to understand the temporal relationships between code history items by offloading the cognitive demand associated with inferring these relationships. We also note that some participants avoided using the Story view because of the high cost associated with the large amount of text it included.

3) Temporal Relationships and Rationale: Participants frequently turned to the Story view when answering questions about the *rationale* behind certain software designs. They valued how stories explicitly tied related history items like searches and code changes together into a coherent narrative. Whereas only a small percent of participants used the story in Study 1 (20%) when debugging code as-it-existed, 5/7 participants of Study 2 used the story to answer the majority of questions they were given.

4) Cognitive Offloading Implicit Relationships of History Items: DevTales includes both *explicit* information (developer visited website A) and *implied* information (website B was more useful than website A, and led to code change C). Participants struggled to piece together the implied connections between history items, and turned to the story to do this for them.

Participants applied great cognitive effort to piece together the implied knowledge that DevTales' Subgoal view provided. For example, P5 had never seen or heard about the KD-Tree datatype before that the original developer used. When asked how she figured out how it was being used, she recalled turning to DevTales: "So I searched 'KDTree' [in DevTale's search bar], and I found that the developer did a Google search [for KDTrees] so I went to one search *before* they looked up 'KDTrees'". She pointed to the previous developer search for 'python vector similarity', before continuing, "and they found KDTrees, and I think they thought that this datatype might be good for their photo vector comparison". She then clicked one of the websites listed under the developer's historical search and the documentation for `scipy.spatial` opened in the browser. "So I think that KDTrees appear here in this documentation, so I assume they found KDTree here and decided to use it." In fact, that's exactly how the original developer found KDTrees.

Whereas P5 underwent several mental steps in order to trace and connect the developer's searches, web visits, and code changes to determine why the original developer used KDTrees, the story format negates those additional mental steps in many cases by stitching those same actions into a cohesive narrative. When trying to understand questions about the rationale behind the code, 5/7 (71%) of participants utilized the Story view, reporting that the story helped them understand the relationship between different historical developer actions over time. The story offloaded the reasoning of the participants who trusted it at face value, reducing the cognitive effort associated with reasoning over the implied relationships of code history items.

P7 hits the nail on the head when describing the ways the story explicit connects original devleoper actions, stating "The story, it really explicitly tells me, for example they imported `pandas` and `scipy` into the main script at first, and it tells me however, 'merely importing these libraries did not solve the problem'." The story does the mental labor of weaving together sequences of the original developer's code changes. He concludes: "So it's more obvious to me what *happened*." P4 justified her extensive use of the story for its "clarity" in bringing out "WHY did the developer use this or that?".

5) Avoiding the Story View: The Story view is inherently more verbose than the Subgoal view; 3/7 participants stated that they avoided the story for this reason. However, once they did start using it, they rarely reverted back to the Subgoal view. P2 initially avoided using the story; while most participants

TABLE III
HARD-TO-ANSWER QUESTIONS ABOUT CODE

Question	Category	Solution	Pass/Total (%)
What Python library does the developer use to work with images? Did they attempt a different API first? If so, which one?	History	They find the code history item where imageio is imported, and the code where it is replaced by PIL.	4/7 (57%)
What does this line of code do? Were other approaches attempted? Why was this approach chosen? line 21: resized_image = img.resize((1,1), Image.Resampling.LANCZOS)	Rationale	They find the historical subgoal of "get average pixel of image". They find the web search "fastest way to calculate average color of image in python"	6/7 (86%)
What motivated the choice of KMeans clustering for mosaic generation?	Provenance	They identify the webpage suggesting using KMeans for Finding Dominant Colors	4/7 (57%)
Which function came first, the make_mosaic_avg, or make_mosaic_kmeans?	History	They identify when each appeared sequentially in the history	6/7 (86%)
Why did the original developer use the KDTree datatype?	Rationale	Navigate to the web searches regarding vector similarity, and find KD Tree web page	7/7 (100%)
Why is Mean Squared Error used to compare image distance?	Rationale/ Provenance	Find the original developer's search for "python image similarity", and the webpage describing MSE, or Subgoal for Detecting Duplicate Images	7/7 (100%)
Why does the code check for duplicate images before placing a new image in the mosaic?	Rationale	They navigate to the subgoal "Ensure similar-looking images aren't adjacent on photo mosaic"	7/7 (100%)
What is meaningful about the threshold value of 0.85?	Intent	They identify the relevant code change within the subgoal "Ensure similar-looking images aren't adjacent on photo mosaic"	6/7 (86%)
How was this 0.85 threshold value determined?	History	They find the consecutive code changes where this threshold is modified	4/7 (57%)
Why does the make_mosaic_kmeans function include the parameter "first_run"?	Rationale	They find the history item responsible for caching images, so they only have to be calculated once	6/7 (86%)
What factors impact the number of nearest neighbors to use in the sliding window of make_mosaic_avg?	Rationale	They identify the subgoal relating to optimizing the runtime of these comparisons in the sliding window	7/7 (100%)

had never worked in image processing before, P2 had. He mentioned that he had used mean-squared-error to compare images in a video, frame-by-frame, and initially assumed that the developer who wrote this python code used it for the same reason. He started writing his assumption about the intention of this code artifact into the answer, and then decided to check the code story to be sure. He noted that he was surprised at how wrong he was, and from that point on during the study, utilized DevTale's code history to understand the developer's intentions. He stated that the "Story might give me a meaningful thought about what exactly happened, because it is a human language. Code is not a human language; I mean it takes time for you to figure out what changes were made exactly, but in the story, you can directly mention the point of what happened in a single line, so I prioritized stories over the code snippets."

Other users avoided the Story view, instead preferring the quick hits of the Subgoal view. P5 stated that she preferred the Subgoal view for its familiarity and brevity, saying: "I just felt like reading through the code and [subgoals] is like more what I used to do. I feel like the story over here is very text-heavy. I feel like it's very difficult to find specific information in a large amount of text." As she describes her decision, she scrolls through the story, reading bits of it, and then states "I don't know, maybe I should [have] read through this."

VI. STUDY 3: IDENTIFYING AND TRANSPLANTING CODE FOR NEW PURPOSES

Finally, we conducted a study to investigate the broader task of code repurposing.

1) Task Goals: Consider a developer assigned their first job at a start-up: *repurpose the underlying engine of a basketball video shot-detection app to instead detect student hand raises for a smart classroom. The developer knows what the app does at a high level, and which functionalities may be useful, but is unfamiliar with the code itself.* We explored this type of reuse because it requires deep understanding of the code's purpose and architecture [58].

We gave 4 student programmers (see Table I) a code base that detects when a person shoots a basketball based on 2D pose estimations through the DeepLabCut API [59]. The code labels each shot as a "make" or "miss" based on audio data, and summarizes in natural language a free throw shooting session in terms of shots, makes, and misses using OpenAI's API. The code base also centered on the niche domain of image processing (like Study 2), but, like the Wordle code (Study 1), was distributed across a more complex file structure. These features challenged participants to 1) **locate** relevant code across many files, and 2) **understand** code in an unfamiliar domain.

We tasked participants with answering a Code Repurposing Questionnaire (See Table IV in Appendix IX-D). The questionnaire asked participants *code location questions* to identify code critical for the reuse scenario, and *code understanding questions* to show that they understand the code well enough to make meaningful modifications. We asked participants to answer structured questions rather than modify code directly to isolate concepts from syntax. This approach 1) minimizes differences in participants' Python fluency and 2) lets us observe how DevTales supports locating and reasoning about relevant code elements without the confounding overhead of

syntax.

2) Data: During this task, we recorded user responses to each question. We also recorded the entire screen of each participant, as well as think-aloud data. Finally, we recorded a brief followup interview with each participant, and asked them why and how they used DevTales in pursuit of their goals, as well as expectations they had with the tool that may have not been met. We report 1) the frequency programmers were able to locate code, 2) how often they correctly answered questions about found code and 3) how they used DevTales to do these things.

A. Findings

Using DevTales, participants successfully located and identified code for 83% of code identification questions, and correctly answered 75% of code understanding questions. We reiterate our research questions:

Regarding **RQ1**, we found a continuation of behavioral trends from the first two studies. Participants again:

- Leveraged the subgoal labels as anchors for top-down program comprehension.
- Employed DevTales' integrated code navigation links to locate code associated with found subgoals.
- Consulted the original developer's web resources to disambiguate API usage and data structures when examining unfamiliar code.

To address **RQ2**, we found that while participants occasionally turned to narrative-formatted histories, they were often overwhelmed by the extended length and cohesive structure of the text. Instead, they defaulted to concise subgoal labels as quick, standalone references.

1) Subgoals for top-down scanning: All 4 participants heavily leaned on DevTales' subgoal search feature to identify relevant code. DevTales supported various search strategies as participants attempted to understand how the developer processed data from Deeplabcut's outputted `.csv` file. P1 searched "rows" and "columns" in DevTales. This search would have yielded nothing if it only had searched the code. Instead, P1 found the subgoal where the original developer was loading and interpreting the data output by Deeplabcut. P1 quickly found a code change that included reading in the output CSV data and analyzing it, and used the "See in Code" feature to find the relevant code. In contrast, P2-4 searched "csv" in DevTales. After first parsing through the subgoal where the original developer specified Deeplabcut to output in CSV format, P2 finds the subgoal associated with loading and analyzing the CSV data. After using the "See in Code" feature, P2 was then able to find the function responsible for working with this data. While they searched for different terms, they all found the code responsible for processing Deeplabcut output.

2) Original web resources to understand code: All 4 participants utilized the web resources of the original developer when attempting to understand how the code worked. For example, after P1 used the keyword search for "rows" and "columns" to locate the code responsible for handling Deeplabcut's output, they expanded the web history of the original developer during the associated subgoal. They saw the searches that the original developer was making at the time, and clicked on a link to a question that the developer asked ChatGPT "How can I plot all series in a pandas dataframe with a multicolumn label applied?". DevTales opened a ChatGPT conversation and scrolled to the question the original developer had asked. P1 then read through ChatGPT's response to this question, and found that the columns referred to (X,Y) coordinates for every body part. He stated "I like this search feature, *just going to exactly what they looked up*. It's very intuitive, because now it feels like I'm *getting the idea of what they thought of when they were writing this code*." He then goes to the next code change after this prompt and clicked "See in Code". Between looking at the code and the chat prompt and response, he correctly answered that the file is plotting "body parts and their relative location".

3) Avoiding longer stories: Although the narrative-formatted histories provided rich contextual detail, all four participants found them too verbose when performing the time-constrained task. After scrolling quickly through several screenfuls of narrative text, P1 commented that it was difficult to find relevant information in such a structure. In addition to being longer, the story in this text did not necessarily explicitly bring out any details critical to the task at hand. Whereas Study 2 focused on the explicit connections between code history items, Study 3 did not require participants to understand their connections, and thus, participants did not invest additional time in reading the stories. P3 summed this experience up by stating that they turned to the story to "see the whole picture, and get an overview of what kinds of major projects [the original developer] is doing", before continuing "I would like to use this more. The reason why I didn't use it later on was, it's very text-heavy, and I know I was crunching for time, like, I only have an hour.. But if I'm actually coding and programming, and I come across a huge GitHub.. I'd actually sit down and read through the whole [story] first–but in the 1-hour timeline, I feel like it was too much information, and I feel like it was better to sift through [the Subgoal view] for what I just need based on the task I was doing."

VII. Discussion

Across three studies, participants consistently sought the *lightest resource that answered their immediate question*, and only advanced to a richer context when the cost was justified.

A. RQ1: Subgoals and Web Resources for Quick Orientation

In all three studies, programmers used subgoal labels as high-level anchors and sign-posts for rapidly mapping high-level intentions to the corresponding code. Connecting these subgoals to code itself helped participants locate implementations, while tying in web resources helped programmers clarify unfamiliar concepts, data structures, and API usage.

Design Implication 1: History-aware code search: Today's code-search models aim to overcome the "semantic gap" between natural language queries and code [60], [61], [62], [63]. Some work attempts to enrich the semantic depth of

software artifacts [60], [64]. DevTales provides new signals–subgoal phrases and web foraging information–that can add semantic depth that code indexing currently lacks. Future studies could compare DevTales-benchmark hybrid search approaches against common benchmarks for top-k recall and usability testing.

B. RQ2: Code Stories & How Programmers Used Them

Programmers favored LLM-generated stories for *why* questions (rationale, provenance, intent)–valuing their explicit weaving of cause-effect relationships between web searches and code changes. By offloading the cognitive burden of inferring these implicit relationships, *stories* helped programmers answer these "hard-to-answer" questions. However, when under tight time constraints or focused on localized code-identification tasks (Studies 1 and 3), programmers found stories too verbose and instead leaned on subgoal labels.

Design implication 2: Targeted, Adaptive Micro-Stories Participants turned to long-form stories when they had deep rationale questions (RQ2), but avoided stories when they needed quick access to anchoring information (studies 1 & 3). Even during study 2, while 71% of participants used the Story view to answer "hard-to-answer" questions, none read the code story in its entirety. Instead, they often found a subgoal they thought was relevant to their task through the Subgoal view, and then used DevTales to navigate to the related section of the story to better understand how the subgoal fit into the larger picture. Our results support the design of adaptive stories that 1) narrate only a specified range of history items and 2) adapt detail to user preferences.

Design Implication 3: Distinguishing Explicit and Implicit Information: LLM-generated code stories weave together explicit and implicit information:

- *Explicit facts:* Directly recorded historical actions (i.e. "developer visited scipy.spatial documentation")
- *Implicit inferences:* Causal links the LLM deduces through reasoning (i.e. "this website influenced the switch to the KDTree datatype".)

During study 2, participants valued the LLM's inferences to understand the causal connection between the history items. Since these inferences are fallible, a meaningful next step is to label inferred versus explicit information within a story, letting users quickly judge trustworthiness of the story's claims. This work could build on recent work in factuality detection [65].

Another interesting direction for future work is automatic LLM-labeling of code changes and subgoal groupings. When conducting a brief analysis of LLM's potential for summarizing code changes, we found that LLMs are highly capable of generating code change labels similar in detail as we used in this study. An example prompt and output are shown in Appendix IX-C.

C. Limitations

Our work includes multiple threats to validity.

1) Internal Validity: Participants were mostly aware that LLMs generated the Story view. This may have impacted their trust in the accuracy of its summarization. Also, our studies each imposed strict time limits, pressuring participants into faster, shallower program comprehension strategies. Another threat to Internal Validity is that we do not compare DevTales to git histories directly. Our method includes finer code change details than typical git commits, and also includes web resources. Since our goal was to understand how developers use these additional resources, and there is no comparable baseline, we conducted these exploratory studies. However, more work is needed to compare against other history resources, like git.

2) External Validity: One threat to external validity is our sample of participants. The three studies (N=10,7,4) each draw from the population of student-programmers. As such, our findings may not generalize to wider experience levels. A second threat to external validity is the size of the code bases. Real-world projects are often larger than the code bases used in this study, and may introduce new challenges, like programmers inadvertently going off-task.

3) Construct Validity: Our subgoal labels were generated post-hoc by the research team, and may not reflect the level of abstraction that is optimal for each of the software evolution tasks. More work should investigate subgoal labels at various levels of abstraction.

VIII. Conclusion

We presented DevTales, a VSCode extension that weaves historical subgoal labels, web-foraging traces, and LLM-generated narrative histories directly into the IDE. Through three user studies, we found that 1) embedded web resources help programmers disambiguate unfamiliar software artifacts and answer otherwise unanswerable questions, 2) subgoal labels with code linking empower efficient top-down code navigation, and 3) narrative histories support deeper reasoning about temporal and causal relationships between code history items, but their verbosity makes them less useful for quick sensemaking. These findings advance our understanding of how to integrate multifaceted code histories into developer workflows, and inform design guidelines for lightweight, trustworthy storytelling features in history-aware tools. Future research should explore adaptive stories and evaluate history-aware IDE features in larger industrial-scale projects.

References

[1] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Evaluation and usability of programming languages and tools*, 2010, pp. 1–6.

[2] L. E. Margulieux, B. B. Morrison, and A. Decker, "Reducing withdrawal and failure rates in introductory programming with subgoal labeled worked examples," *International Journal of STEM Education*, vol. 7, pp. 1–16, 2020.

[3] A. Horvath, A. Macvean, and B. A. Myers, "Meta-manager: A tool for collecting and exploring meta information about code," 2024.

[4] J. Allen and C. Kelleher, "An exploratory study of programmers' analogical reasoning and software history usage during code re-purposing," in *Proceedings of the 2024 IEEE/ACM 17th International Conference on Cooperative and Human Aspects of Software Engineering*, 2024, pp. 109–120.

[5] ——, "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)*. IEEE, 2024, pp. 148–162.

[6] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 344–353.

[7] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[8] I. Crk and T. Kluthe, "Assessing the contribution of the individual alpha frequency (iaf) in an eeg-based study of program comprehension," in *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, 2016, pp. 4601–4604.

[9] K. Maruyama, T. Omori, and S. Hayashi, "Slicing fine-grained code change history," *IEICE TRANSACTIONS on Information and Systems*, vol. 99, no. 3, pp. 671–687, 2016.

[10] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.

[11] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *Journal of Systems and Software*, vol. 7, no. 4, pp. 341–355, 1987.

[12] C. L. Corritore and S. Wiedenbeck, "An exploratory study of program comprehension strategies of procedural and object-oriented programmers," *International Journal of Human-Computer Studies*, vol. 54, no. 1, pp. 1–23, 2001.

[13] N. Pennington, "Comprehension strategies in programming," in *Empirical Studies of Programmers: Second Workshop, 1987*, 1987, pp. 100–113.

[14] J.-M. Burkhardt, F. Détienne, and S. Wiedenbeck, "Object-oriented program comprehension: Effect of expertise, task and phase," *Empirical Software Engineering*, vol. 7, pp. 115–156, 2002.

[15] R. Brooks, "Towards a theory of the comprehension of computer programs," *International journal of man-machine studies*, vol. 18, no. 6, pp. 543–554, 1983.

[16] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on software engineering*, no. 5, pp. 595–609, 1984.

[17] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 23–34.

[18] K. Herzig and A. Zeller, "The impact of tangled code changes," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 121–130.

[19] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 351–360.

[20] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi *et al.*, "A fine-grained data set and analysis of tangling in bug fixing commits," *Empirical Software Engineering*, vol. 27, no. 6, p. 125, 2022.

[21] 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)*. IEEE, 2015, pp. 1–10.

[22] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 341–350.

[23] J. Park, Y. H. Park, S. Kim, and A. Oh, "Eliph: Effective visualization of code history for peer assessment in programming education," in *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*, 2017, pp. 458–467.

[24] Y. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 223–233.

[25] J. Allen and C. Kelleher, "Exploring analogical reasoning and history use in software re-purposing," in *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2023, pp. 242–244.

[26] Y. Yoon and B. A. Myers, "Semantic zooming of code change history," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2015, pp. 95–99.

[27] R. Robbes and M. Lanza, "A Change-based Approach to Software Evolution," *Electronic Notes in Theoretical Computer Science*, vol. 166, pp. 93–109, Jan. 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066106005317

[28] L. Hattori and M. Lanza, "Syde: a tool for collaborative software development," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: Association for Computing Machinery, May 2010, pp. 235–238. [Online]. Available: https://doi.org/10.1145/1810295.1810339

[29] R. Catrambone, "The effects of labels on learning subgoals for solving problems." 1989.

[30] A. Decker, L. E. Margulieux, and B. B. Morrison, "Using the solo taxonomy to understand subgoal labels effect in cs1," in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, 2019, pp. 209–217.

[31] L. E. Margulieux, B. B. Morrison, and A. Decker, "Design and pilot testing of subgoal labeled worked examples for five core concepts in cs1," in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 2019, pp. 548–554.

[32] B. B. Morrison, L. E. Margulieux, B. Ericson, and M. Guzdial, "Subgoals help students solve Parsons problems," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 2016, pp. 42–47.

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

[34] C. Kelleher and M. Ichinco, "Towards a model of api learning," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2019, pp. 163–168.

[35] B. Hartmann, M. Dhillon, and M. K. Chan, "Hypersource: bridging the gap between source and code-related web sites," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011, pp. 2207–2210.

[36] M. X. Liu, A. Kittur, and B. A. Myers, "Crystalline: Lowering the cost for developers to collect and organize information for decision making," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–16.

[37] V. T. T. Pham and C. Kelleher, "Code histories: Documenting development by recording code influences and changes in code," *Journal of Computer Languages*, vol. 82, p. 101313, 2025.

[38] E. Agrawal, O. Alam, C. Goenka, M. Iyer, I. Moise, A. Pandian, and B. Paul, "Code compass: A study on the challenges of navigating unfamiliar codebases," *arXiv preprint arXiv:2405.06271*, 2024.

[39] C. Kelleher, R. Pausch, and S. Kiesler, "Storytelling alice motivates middle school girls to learn computer programming," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 1455–1464.

[40] B. Casey, J. E. Kersh, and J. M. Young, "Storytelling sagas: An effective medium for teaching early childhood mathematics," *Early Childhood Research Quarterly*, vol. 19, no. 1, pp. 167–172, 2004.

[41] S. Erkut, "Multicultural adventure stories as medium for teaching math in the early elementary grades: An evaluation," in *Wellesley Centers for Women Board of Overseers meeting. Wellesley, MA*, 2003.

[42] D. B. Jung and J. E. Kim, "A study on the effects of storytelling-linked integrated math programs on young children's mathematical disposition and self-efficacy," *Korean Journal of Childcare and Education*, vol. 11, no. 2, pp. 151–175, 2015.

[43] S. Rowcliffe, "Storytelling in science," *School science review*, vol. 86, no. 314, p. 121, 2004.

[44] P. B. Armstrong, *Stories and the brain: The neuroscience of narrative*. JHU Press, 2020.

[45] R. Landrum, K. Brakke, and M. McCarthy, "The pedagogical power of storytelling. scholarship of teaching and learning in psychology, 5 (3), 247–253," 2019.

[46] A. C. Graesser, N. L. Hoffman, and L. F. Clark, "Structural components of reading time," *Journal of Verbal Learning and Verbal Behavior*, vol. 19, no. 2, pp. 135–151, 1980.

[47] G. H. Bower and M. C. Clark, "Narrative stories as mediators for serial learning," *Psychonomic science*, vol. 14, no. 4, pp. 181–182, 1969.

[48] A. Y. Wang, A. Head, A. G. Zhang, S. Oney, and C. Brooks, "Colaroid: A literate programming approach for authoring explorable multi-stage

tutorials," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–22.
[49] A. Horvath, A. Macvean, and B. A. Myers, "Support for long-form documentation authoring and maintenance," in *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2023, pp. 109–114.
[50] P. Wuilmart, E. Söderberg, and M. Höst, "Programmer stories, stories for programmers: Exploring storytelling in software development," in *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming*, 2023, pp. 68–75.
[51] D. Fontanet Losquiño and T. Urdell, "Why do developers struggle with documentation while excelling at programming," B.S. thesis, Universitat Politècnica de Catalunya, 2014.
[52] 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*, 2006, pp. 492–501.
[53] A. Kuhn and M. Stocker, "Codetimeline: Storytelling with versioning data," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1333–1336.
[54] J. Sweller, J. J. van Merriënboer, and F. Paas, "Cognitive architecture and instructional design: 20 years later," *Educational Psychology Review*, vol. 31, no. 2, pp. 261–292, 2019.
[55] C. Coder. (2022, Feb) Live coding a wordle clone (5 hrs) — html sass js. YouTube. [Online]. Available: https://www.youtube.com/watch?v=PNGgQzw6PQg
[56] T. M. Shaft and I. Vessey, "The relevance of application domain knowledge: The case of computer program comprehension," *Information systems research*, vol. 6, no. 3, pp. 286–299, 1995.
[57] A. J. Ko and B. Uttl, "Individual differences in program comprehension strategies in unfamiliar programming systems," in *11th IEEE International Workshop on Program Comprehension, 2003.* IEEE, 2003, pp. 175–184.
[58] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.
[59] A. Mathis, P. Mamidanna, K. M. Cury, T. Abe, V. N. Murthy, M. W. Mathis, and M. Bethge, "Deeplabcut: markerless pose estimation of user-defined body parts with deep learning," *Nature neuroscience*, vol. 21, no. 9, pp. 1281–1289, 2018.
[60] S. Huang, B. Cai, Y. Yu, and J. Luo, "Excs: accelerating code search with code expansion," *Scientific Reports*, vol. 14, no. 1, p. 29166, 2024.
[61] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 344–354.
[62] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, pp. 3149–3185, 2017.
[63] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code search is all you need? improving code suggestions with code search," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
[64] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 260–270.
[65] I. Chern, S. Chern, S. Chen, W. Yuan, K. Feng, C. Zhou, J. He, G. Neubig, P. Liu *et al.*, "Factool: Factuality detection in generative ai–a tool augmented framework for multi-task and multi-domain scenarios," *arXiv preprint arXiv:2307.13528*, 2023.

IX. Appendix

A. Story Generation Prompt

We used the following prompt to generate the Story view using OpenAI's o1 model:

```
You are a creative and detail-oriented storyteller.
    I will provide you with a list of code/history
    items ("the input"), and I want you to transform
     them into an HTML story that is written in a
    narrative style, with the following requirements
    :

1. **HTML Structure**:
  - Begin with <!DOCTYPE html> and <html lang="en">,
      and include a <head> section with a <title>,
     then a <body> section.
  - End with a closing </html>.

2. **Story Format**:
  - Turn the entire code history into a first-person
      story-like narrative with paragraphs (<p>
     tags).
  - You may group multiple history items together
     into a single paragraph as you weave them into
      a story, but preserve chronological flow.
  - Incorporate brief transitions to make it feel
     more like a journey rather than a strict log.

3. **Subgoal Lines** (e.g., subgoal: Initial
    Research & Environment Setup):
  - Render them in your story as something like:

The first <strong>subgoal</strong> was clear: <em>
    Initial Research & Environment Setup</em>.

  - Make sure "subgoal" itself is styled in <strong>
      and the text that follows is within <em>.

4. **ChatGPT Prompt Lines** (e.g., ChatGPT Prompt:
    How to get multi word input... (http://localhost
    :...)):
  - Treat these as references or external resources
     the developer looked up in the story.
  - For the link portion (the text in parentheses),
     create a normal HTML anchor that opens in a
     new tab. For instance:

<a href="http://localhost:8000/..." target="_blank">
    visit to the prompt about multi-word input</a>

  - The anchor text can be short or descriptive, but
      ensure you include the URL in the href and
     set target="_blank".

5. **File + Action Lines** (e.g., track_shooting.
    py_21: Commented out create_pretrained_project
    ... or notes.txt_50: Appended a note...):
  - Turn these into parts of the story referencing
     what the developer did.
  - Use a clickable span with a custom class and an
     action_id attribute. For example:

<span class="clickable" action_id="track_shooting.
    py_21">
     They commented out create_pretrained_project,
        focusing instead on a direct analyze_videos
         call.
    </span>

  - You can paraphrase the action after the colon to
      fit your story style.

6. **Hyperlink Repeats**:
  - If the same URL repeats, you can either repeat
     the link or mention it as "revisited the same
     resource." Use your best judgment.

7. **Overall Tone**:
  - Write in a narrative, cohesive tone, as if you
     are telling a developer's story or quest. Use
     transitions like "Embarking on the next stage
     ...", "After several trials...", etc.

---
```

```
Here is my sequence of historical code events.
    Please transform them into a complete HTML
    document following all the above instructions.

**INPUT** (each line may be subgoal:, ChatGPT Prompt
    :, filename_lineNumber:, etc.):

subgoal: Initial Research and Environment Setup
ChatGPT Prompt: How to get multi word input from
    user in python?(http://localhost:8000/
    BasketballChats/REFERENCES_ADDED_Multi-word
    Input in Python.htm#PROMPT_ID_1)
subgoal: Initial DeepLabCut Code Integration
notes.txt_9: Add personal notes comparing pose
    estimation libraries and elevating admin
    privileges for installation.
track_shooting.py_10: Initialize the Deeplabcut
    script to create a new project and extract
    frames from the provided video.
track_shooting.py_12: Update the config file path
    with a date-based directory and comment out the
    create_new_project call.
subgoal: Exploring OpenPose and Mac Installation +
    Pretrained DLC Model
ChatGPT Prompt: How do I install openpose in a conda
     environment?(http://localhost:8000/
    BasketballChats/REFERENCES_ADDED_Human Pose
    Estimation Tools.htm#PROMPT_ID_14)
.....
Data Continues...
.....
```

B. Python-Mosaic Code Base: Image-Processing Logic

The *Python-Mosaic Code* includes Python code which takes in as input a directory of `.png` image files, and outputs a `.png` output image. This project is all composed in a single Python file, which we believed would reduce code navigation time and allow for more comprehension time. The project uses several external APIs such as PIL, Pandas, and NumPy, which is representative of many Python projects; the logic of this code base includes input and output handling of images, image processing, clustering algorithms, similarity metrics, performance optimizations, and more.

The final logic of Python-Mosaic Code Base:

1. Read in target image file
2. Read all images from directory
3. Resize all images to be same size and square shape
4. Calculate average pixel for each photo
5. Downsample target photo
6. For each remaining pixel of target photo:
  - **6a.** Find image from photos directory with most similar average pixel
  - **6b.** Detect if there are potential duplicates in a nearby sliding window
  - **6c.** If no duplicates are detected, append image to matrix parallel to target photo

7. Stitch together matrix of images
8. Save created image to disc

C. LLM-Generated Code Diff Labeling

We used the following prompt to generate code diff labels using OpenAI's o1 model.

```
I am providing you with the complete development
    history of a software project designed to help
    basketball players track and analyze their free
    throw shooting. The history includes various
    entries such as code changes, web searches,
    website visits, and developer prompts.

Your task is to act as an expert in documenting code
     histories and annotate each code change with a
    succinct, one-sentence summary derived from its
    diff and the surrounding context. Each code
    change is marked by a delimiter in the format:

* Code Change: [number] *

This marker indicates the beginning of a new code
    change entry, which continues until the next
    code change marker or any web activity entry.
    There are exactly 100 code changes in total.

For each code change, extract and record:

code_change_number: The number following "Code
    Change:" (as an integer).

filename: The filename associated with the code
    change.

label: A concise one-sentence summary that captures
    the essence of the modifications made.

Your final output must be a valid JSON file
    containing an array called "code_changes". Each
    element of the array should be a JSON object
    with the keys "code_change_number", "filename",
    and "label".

Please ensure that:

Each of the 100 code changes is processed.

Each code change is accurately associated with the
    correct filename.

The JSON output is well-formed.

Now, here is the data:


* Code Change: 101 * - /Users/usertests/Desktop/
    BasketballApp//explore_dlc_output.py
---
+++
@@ -60,7 +60,7 @@
    curr_frame = shot_indicator_frames[i]
    prev_frame = shot_indicator_frames[i-1]
    if curr_frame - prev_frame > frame_rate: # no
        way this is the same shot one second+ later
- split_indices.append(i-1)
+ split_indices.append(i)

 print(split_indices)
 print(shot_indicator_frames)

* Code Change: 102 * - /Users/usertests/Desktop/
    BasketballApp//explore_dlc_output.py
---
+++
@@ -70,6 +70,7 @@
 for i in split_indices:
    shot_index_segments.append(shot_indicator_frames
        [curr_index:i])
    curr_index = i
```

```
+shot_index_segments.append(shot_indicator_frames[
    curr_index:])
 print(shot_index_segments)
 exit()
 #plt.plot(valid_frame_indices,
     wrist1_likelihood_rolling[valid_frame_indices],
      label='WRIST_1_likelihood_rolling')#, s=2)

 Data continues...
```

OUTPUT:

```
{
  "code_changes": [
    {
      "code_change_number": 101,
      "filename": "/Users/usertests/Desktop/
          BasketballApp//explore_dlc_output.py",
      "label": "Adjusts the shot-splitting logic by
          appending the current index instead of the
          previous index."
    },
    {
      "code_change_number": 102,
      "filename": "/Users/usertests/Desktop/
          BasketballApp//explore_dlc_output.py",
      "label": "Ensures the remaining shot frames are
          added as the final segment by appending
          shot_indicator_frames[curr_index:]."
    },
    {
      "code_change_number": 103,
      "filename": "/Users/usertests/Desktop/
          BasketballApp//explore_dlc_output.py",
      "label": "Introduces shot_peaks based on medians
           of frame segments and begins defining
          shot_indices around the first peak."
    } .....
}
```

D. Code Repurposing Questionnaire

TABLE IV
CODE REPURPOSING QUESTIONNAIRE

Question	**Category**	**Solution**	**Results**
In which file does the original project use a computer vision API to extract human pose data from the basketball videos?	Locating	track_shooting.py	3: Correct 1: No response
What is the name of the API?	Understanding	Deeplabcut	3: Correct 1: No response
Which output data file produced by the computer vision API does the original developer load to detect free throws?	Locating	finds (long name) output csv file	3: Correct 1: Incorrect
How is the data in that output file structured? What does each column represent? What does each row represent?	Understanding	Indicates that the file is keypoints and their locations	4: Correct
In which file and function does the original developer detect free throws using the pose data?	Locating	explore_dlc_output.py, get_shot_peaks()	4: Correct
Within this function, identify the line of code responsible for detecting the shot action.	Locating	shot_indicator_frames = ...	4: Correct
How would you modify this line to instead detect if one hand is raised? (ONLY one wrist is above the person's forehead)	Understanding	Change the condition from OR to XOR	4: Correct
What would the new line of code be?	Understanding	replace — with ^operator on numpy arrays	1: Correct 3: Syntax Err
Which file and function does the developer call the OpenAI API to generate a natural language summary of the basketball session?	Locating	promptGen.py - call_open_ai()	4: Correct, 1: File only
Which line of code (and in which file) sends the basketball data into this function?	Locating	call_open_ai(prompt_data), explore_dlc_output.py	2: Correct, 2: Incorrect
How would you modify the prompt to summarize hand raises instead of basketball performance? Include what would be needed to address the logic if hand_raised_count $<$10 or not.	Understanding	Modify prompt variable to specify new goals	3: Correct, 1: Out of Time