Exploring the impacts of semi-automated storytelling on programmers' comprehension of software histories

John Allen
Computer Science & Engineering
Washington University in St. Louis
St. Louis, USA
johnjallen@wustl.edu

Caitlin Kelleher
Computer Science & Engineering
Washington University in St. Louis
St. Louis, USA
ckelleher@wustl.edu

***Abstract*—Software developers have difficulty understanding the rationale and intent behind original developers' design decisions. Code histories aim to provide richer contexts for code changes over time, but can introduce a large amount of information to the already cognitively demanding task of code comprehension. Storytelling has shown benefits in communicating complex, time-dependent information, yet programmers are reluctant to write stories for their code changes. We explored the utility of narratives made by generative AI. We conducted a within-subjects study comparing the performance of 16 programmers when recalling code history information from a list-view format versus a comparable AI-generated narrative format. Our study found that when using the story-view, participants were 16% more successful at recalling code history information, and had 30% less error when assessing the correctness of their responses. We did not find any significant differences in programmer's perceived mental effort or their attitudes towards reuse when using narrative code stories.**

***Index Terms*—code histories, storytelling, software versioning**

I. Introduction

Software development is inherently a story that often has many twists and turns. While developers fix bugs and implement functional requirements in software, they often face a sequence of dynamic challenges, and must solve a series of problems along the way. Many challenges that arise during development are unforeseen, and are tackled in an ad hoc manner, leading developers to seek out web resources in order to overcome these challenges [1]. These web resources often influence developer's design choices in the written code [1], [2]. The challenges developers face, and the resources developers use to overcome them are rarely recorded by current software versioning systems. With traditional versioning resources, developers often find it difficult to understand the intent and rationale behind the way code is written [3], [4].

Research suggests that presenting rich historical context about code artifacts may improve developers' ability to understand the original developer's intentions [5]. Much of this research has focused on improving traditional list-view representations of code edits over time, such as improving git commits [6]. However, historical information can also impose additional mental demands on software developers [5]. Software development is already a cognitively demanding task [7], and tools that add rich historical information must consider the cognitive costs associated with that information [8].

Storytelling is a powerful tool in helping people understand complex, time-dependent information with reduced cognitive effort [9], [10]. Capturing the story behind development, and presenting it as a narrative instead of a traditional list of edits may be a powerful way to provide historical context for the design of specific code artifacts. Some research has explored asking developers to write stories for their code [11], [12], but no work that we are aware of has investigated the usability of code stories for future developers.

Additionally, these studies have found that programmers do not want to invest more time in writing a story for their code [12]. Recent developments in generative AI have opened the door for automated story generation. In this paper, we explored the potential utility of GPT-4 generated code history stories, with minimal corrections made by the authors. To generate stories, we compiled a dataset of code changes for two software projects. We organized each set of historical information into a hierarchical "list-view" of concise code changes and web searches that loosely resembles a well-maintained git repository in which each commit represents one cohesive and small change. We then use GPT-4 to convert the information from the list-view into a narrative "story-view", in which the information is written in a narrative paragraph. We demonstrate examples of the two history formats in Figure 1.

We presented these story-view code change histories, as well as their list-view counterparts to 16 programmers in a user study to investigate the impacts of the narrative format.

We posed four distinct research questions:

- RQ1: Are participants better at recalling historical code information after studying the list or story view?
- RQ2: Are participants more aware of their correctness when recalling historical information after studying the list or story view?

NSF Grant No. 2128128

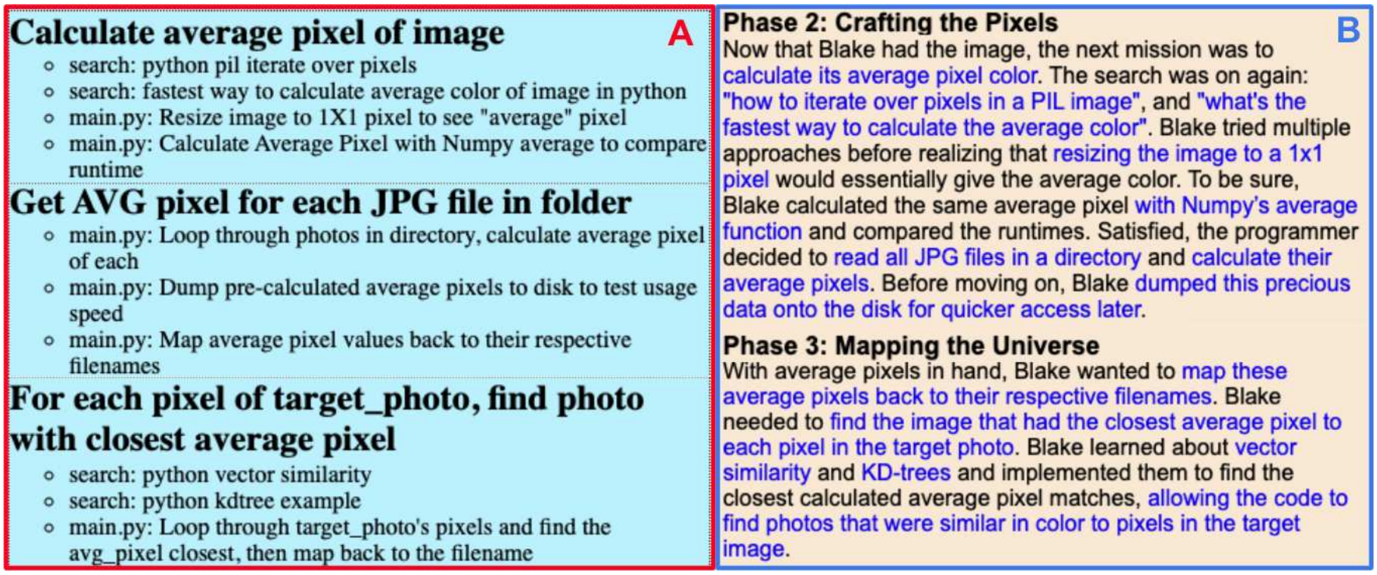

Fig. 1. The list-view history format (A) includes information in a hierarchical list format, with short descriptions for each action the developer takes within a hierarchical subgoal. The story-view (B) composes the information from the list into a overarching narrative in natural language.

- RQ3: Do participants perceive code bases as more reusable after studying the list or story view?
- RQ4: Do participants experience a higher cognitive demand after studying and answering questions about the list or story view?

Our results suggest that participants are better and more confident at recalling historical information when studying the story-view than the list-view, especially when recalling time-dependent historical information. We found no significant differences between programmers' perceptions of the reusability of the underlying code base or their perceptions of mental effort. Although it is not statistically significant, we noted a trend towards lower mental effort for participants when using the story.

II. Background

Our research builds on research in Program Comprehension and Narrative Learning. Specifically, our paper provides insight into how narrative learning can impact top-down program comprehension.

A. Program Comprehension

Understanding an unfamiliar code base is usually a cognitively overwhelming task [7], [13]. Programmers browse information ineffectively [14], [1], and often struggle to parse through and understand a code base [4].

Research has investigated the strategies programmers employ when attempting to understand unfamiliar code. Notably, research groups program comprehension strategies into two categories: “bottom-up” and “top-down” approaches. Bottom-up program comprehension typically includes programmers looking at low-level, implementation code, and figuring out what the smallest pieces of code are doing, and then building up from there in order to understand how they integrate together to account for the behaviors of the software [15], [16].

Top-down approaches involve programmers identifying high-level functionalities, components, and architecture of the software, and then exploring their implementations in order to understand how they are composed [17], [18].

Most research in program comprehension suggests that programmers apply both bottom-up and top-down approaches when attempting to comprehend an unfamiliar code base [19], [20]. Typically, programmers iterate through top-down and bottom-up cycles of understanding a program, beginning with a top-down understanding of the highest level components in a piece of software, identifying parts that are relevant to them, and then exploring their implementations using a bottom-up approach [21].

1) Barriers in Program Comprehension: Research has pinpointed bottlenecks during program comprehension. Programmers struggle when attempting to answer questions about unfamiliar code bases [22], [23]. Research has repeatedly shown that the questions programmers find most difficult to answer involve the intent and rationale behind the original developer's design decisions [4], [3], [24]. This information is often not included in the software or documentation itself [4], however, some work investigates how to incorporate this historical information into the workflow of programmers (see Section III-B2).

B. Narrative Learning

Since the dawn of human language, storytelling has been used to communicate complex information between individuals [9]. Stories have been used in teaching throughout all of human history, and are deeply ingrained in both human culture [25], [10] and cognition [10], [26]. Research suggests that our brains have special mechanisms for processing and remembering stories [26], suggesting that there is a benefit to receiving information in a story format.

Research suggests that using storytelling to present content may help people process and remember the new information. One study showed that applying a narrative format to information improved participants' speed in reading through it and ability to recall the information [27]. Other work showed that people were over six times more likely to remember information presented as part of a story rather than presented in a list [28].

Research shows a positive impact of storytelling in various contexts. Storytelling has been shown to increase students' math performance by helping them link new ideas to concepts they are already familiar with [29]. Erkut showed that children who learned geometry through means of storytelling improved more than those who learned the same content without storytelling [30]. Jung showed that storytelling-based curricula can also improve students' self efficacy and performance in math [31]. The power of storytelling has also been built into science curricula [32]. Research has shown that introducing scientific concepts in stories can increase student motivation and learning [33].

1) Storytelling in Programming: Storytelling has also been used to motivate programming. Storytelling Alice showed that using story-based examples to introduce programming to young girls increased their motivation to program [34]. Other work has introduced storytelling to help novice programmers compose more complex programs [35].

Generally, research in narrative learning suggests that storytelling improves learners' attention [36], [37], motivation [38], [34], [10], and memory recall of information [36], [39]. Our work explores how storytelling can be applied to historical software information in order to improve the understandability of the code for future developers.

III. Related Work

Our work is related to research in Code Histories and Capturing Design Rationale, and is closely related to a small, newer body of work in storytelling for code histories.

A. Code Histories

Research focuses on 1) the motivations for improving access to historical software information and 2) exploring how new code history tools help programmers.

1) Need for Improved Historical Information: Research suggests that programmers' main difficulties during program comprehension are related to inadequate access to historical information about software. Surveys and observational studies have shown that developers often have a hard time answering questions about the design decisions behind software artefacts [3], [4]. Typically, the only resources developers have when searching for historical development information about code is in versioning control systems, and sometimes, the original developer themselves [4], [40]. While developers prefer asking the original developer questions directly about their rationale, the original developer is often unavailable as a resource [4].

Currently, the standard for version history involves git-style source control. Research suggests that version control commit messages are not sufficient to answer developers' questions [41]. Often, developers incorporate multiple unrelated changes into a single commit, resulting in "tangled commits" [41], [42], [43], which make it difficult for future developers to understand the reasoning for particular code changes [41], [44]. Additionally, relevant historical information may not appear in git commits, as they may be in uncommitted changes [40].

2) Code History Tools: Some research has explored untangling git-style commits [6]. Other work has investigated entirely new history capture systems, recording every change at the keystroke level [45], [5], [46]. This introduces a large amount of extraneous information [5]. Some work investigates allowing the programmers to group these fine-grained changes at various levels of abstraction [5], and automatically removing historical data that programmers deem irrelevant [13].

B. Capturing Design Rationale

Research in Capturing Design Rationale explores 1) the rationale information needs of programmers and 2) tools that help software developers and designers record their rationale for design decisions. Some work explores how storytelling can be used to present the rationale behind software elements.

1) Design Rationale: A growing body of work highlights the importance of capturing the design rationale behind specific code changes [47], [48], [49]. This work emphasizes that understanding the original developer's rationale is a barrier when working in unfamiliar code bases [48].

Lee emphasizes the need for design rationale capture to include practical justifications for design decisions [50]. Conklin and Burgess argue that design rationale should be captured with minimal impact to the regular design process [47].

It is possible to capture some information that may help programmers reason about design rationale. As developers implement solutions to problems, they often visit resources online [1], [51], [52]. These resources include additional context behind these developers' intentions behind code additions, and may be useful for future developers. However, most design rationale or code history systems do not capture this information, and evidence suggests it will be increasingly difficult to track the origin of software artifacts retrospectively [53].

2) Providing Additional Context to Programmers: Some tools allow developers to attach additional information to their code changes as they write code. Deep Intellisense links lines of code to relevant external documentation such as bug reports, emails, and source control check-ins [54]. HyperSource ties the websites developers visit to subsequent code changes [55]. Crystalline passively collects and organizes web resources developers visit as they write code in order to 1) improve developer decision-making, and 2) track additional context for decisions made [56]. Meta-manager allows developers to attach information about provenance and rationale behind code changes in the IDE [57]. However, none of these studies incorporate web information into a full code history, or investigate other programmers' comprehension or usage of the original developer's web foraging activity.

3) Storytelling for Code Histories: Very little work investigates incorporating information about the design rationale behind historical software changes as a story. Two papers closest to our work are:

Wuilmart [12] investigated asking developers to write stories to describe the processes they follow while completing programming tasks. This work demonstrated that developers have difficulty crafting stories, and do not want to add additional documentation duties to their workload.

Sodalite [11] supports developers authoring long-form stories behind their code changes by linking the text they create to code elements in their IDE. Sodalite also helps by detecting when these stories become out-of-date, and informs the author of a need to update their story, as well as readers that a story may be out of date. Sodalite helps readers identify documents relevant to given parts of the code's story.

Still, documenting software is tedious work that developers avoid when possible [58], [59], [56], and developers are unlikely to want to author stories for every code change they make [12]. We are not aware of any work that has investigated the usability of code stories during program comprehension or the use of generative AI to author code stories.

IV. Creating Code Histories

In order to explore the impacts of presenting software history as a story to developers, we first created list-view representations of code change histories for two code bases. The list-view representations consisted of short descriptions for code changes and information seeking activities. We then created story-view representations for both code bases. We provide snapshots of both formats in Figure 1. In this section, we describe the process of capturing historical information, labelling development activities, and organizing these histories.

A. History Capture

We captured the history of actions two developers took while each developed an independent project.

1) Code Bases: We selected disparate code bases to capture a breadth of project domains. The Python Code Base uses image processing and data manipulation libraries to generate a photo mosaic of a target image assembled from a directory of given images. This code was written by a member of the research team as a personal project. The Web Code Base used a traditional web development framework to create a tile-based game. In contrast to the Python Code Base, the Web Code Base was written by an online live code streamer, "Coder Coder", as a personal project [60]. The developers of each code base developed their respective project in a natural way that included unanticipated problems, allowing us to capture authentic creation histories for both.

As each developer worked, we captured their code state each time they tested their program. We determined instances of testing differently for various code projects. For execution-based softwares, we defined testing as executing the program, whereas for web-development projects we defined testing as interacting with localhost. We chose this approach because we believe that testing the code indicates the programmer believes they have made a meaningful change and will get some important feedback from the output of the given code state. We are aware of no other research which studies the usability of testing-captured software histories. In addition to code changes, we also record the web searches and website visits the developer made as they developed their project. This data provides additional context behind the code changes that followed information searches.

B. History Data Processing

We then combined and labeled the set of code changes and web activities for each project.

1) Activity Segmentation: We used different processes to group sequences of code changes and web activities. We grouped code edits into linear chunks based on the magnitude of consecutive code changes made. We grouped web activities into linear chunks based on the textual similarity of their titles.

The entire evolution of a software history base can be seen as a long sequence of code changes. In order to segment these changes into high-level "subgoals", we explored various heuristics that grouped consecutive changes together. We experimented with different heuristics until we were satisfied that we were grouping together meaningfully similar changes. This process is described further in Appendix D.

The purpose of segmenting consecutive code changes and web activity into groups was to offload the work of the original developer of segmenting their own actions to only labeling the segmented code changes. We acknowledge that there are many other ways to group related code changes or web activity together.

2) Labeling Developer Activity: The research team labeled each code edit grouping using a high-level, one-line summary of a sequence of code changes. In order to resemble standard practices in historical code change descriptions, we designed these labels to mimic the style of typical git commits. An example from the Python task is: "main.py: Calculate Average Pixel with Numpy average to compare runtime" (see Fig 1). A member of the research team who had witnessed the entire development process labeled each code edit group.

C. Creation of List-View

Traditional git commits are commonly presented as a sequence of short descriptors, as shown in Figure 2-A. Emulating this style, we also presented historical information in a similar, list-view fashion (see Figure 2-B). It is important to note, however, that in addition to code changes, we also include the developers' web activity as they wrote code. We organized the hand-labeled code change and web visit clusters and organized them into a hierarchical goals.

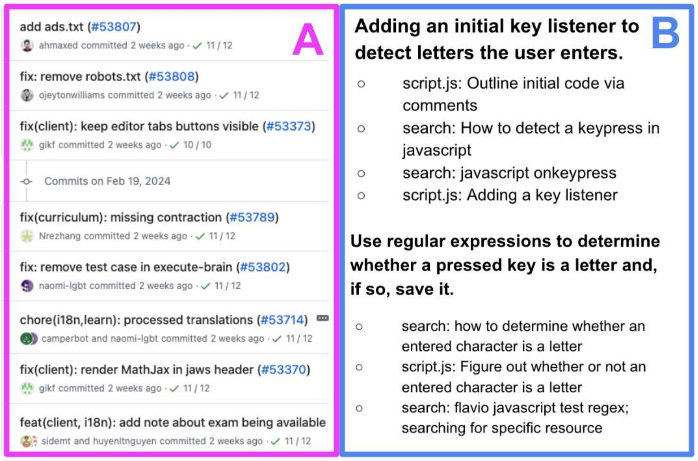

Fig. 2. The list-view format (B) includes short, descriptive text to describe code changes and actions a developer performed as they modified a software base, and is comparable in style to a git commit history (A)

1) Hierarchical Activity Organization: We noticed that consecutive clusters of code changes and web activity were often related under a unifying subgoal of the developer. So, we manually segmented sequences of actions into groups, and labeled the overarching "goal". For example, "Calculate average pixel of image" in Fig 1, is a label for a goal-level grouping of actions, which includes two search groupings followed by two code edit groupings. The entire list-view history of each code base consists of a sequence of such goal-level groupings, which each had their own code edit and/or web groupings.

D. Creation of Story-View

Evidence suggests that using storytelling to present sequential information may help people process and remember the information [27]. To explore storytelling as an alternative to traditional list-oriented presentations of software history, we constructed narrative-formatted "story-view" descriptions for each code base's history.

Describing code changes in a narrative format would add a nontrivial amount of work to a developer's workload [12]. Further, programmers do not wish to spend extra time crafting stories for code they write [12], [56]. So, we explored using generative AI to create these stories based on the list-view history for a given code base.

1) AI Code Story Generation: We prompted GPT-4 to create a narrative story for each list-view history, and then we made slight manual modifications to the resulting story to ensure consistency with the list-view, and to keep the descriptions of the original developer ungendered in order to avoid gender bias in our study [61]. The changes we made to the generated stories were minimal, and typically involved editing sentences that omitted a detail from the original list-view history. In general, the story for each code history required very few edits, making this process much less burdensome than writing a code story from scratch. More details and examples of the prompting we utilized are included in Appendix A.

To present these stories to programmers, we transformed each story into a simple website. Details which were direct references to programmer actions were highlighted in blue, as shown in Fig 1-B. We chose this design to be consistent with future work in which we may add actionable hyperlinks to the code actions within a story.

V. Methods

We performed a within-subjects study of 16 student programmers performing recall tasks using 1) a list-view history and 2) a story-view history of two separate code bases.

A. Participants

We recruited sixteen participants pursuing degrees related to computer science at a private university in the United States. To ensure that participants had adequate computing backgrounds to make progress on reuse through re-purposing tasks, we recruited only students who had taken at least two programming classes or who had at least three months of work or internship experience programming.

Eight of our participants identified as female, seven as male, and one preferred to not respond. Our participants had substantial programming experience, reporting an average of eight computing courses and more than seventeen months of work experience. Twelve participants were enrolled in undergraduate programs. One was pursuing a Master's, and the remaining three participants were pursuing their PhD. Two undergraduate participants were not majoring in CS: one pursuing a degree in Data Science, while the other majored in Finance. All other participants' primary degrees were in Computer Science or Engineering.

B. Study Procedures

We designed a within-subjects study of 16 programmers completing two tasks: one where they study and answer questions about a list-view code history, and the other about a story-view code history. In order to control for the different tasks and ordering effects, we employed a Latin-squares study design to assign participants their task order, as well as which task they received the story-view for and which one they received the list-view. Our sample size of 16 perfectly balances the frequencies of all conditions.

We gave participants an introductory task where they had five minutes to study the list-view code history of a sample code base, and then five minutes to study the story-view of the same code base, then given a practice exam that asked the same types of questions that would be on the official user test.

After the introduction, participants received their first task, where they had up to 15 minutes to study either the list-view or story-view of a new code base. After the participant was content with their studying, or the 15 minutes ran out, they took a quiz on the history behind the code base. After this, they began their second task, where they again had up to 15

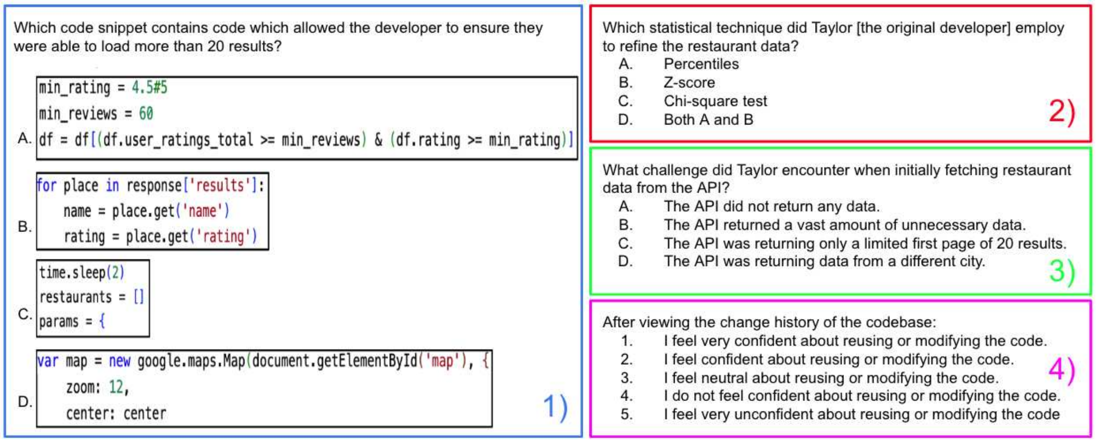

Fig. 3. Set of sample questions of the various types: 1) code interpretation, 2) factual, 3) historical, and 4) attitudes towards reuse.

minutes to study the second code base's history using the view which they did not have during the first task, and then took a quiz over its content. We did not include delays or distraction tasks between the studying and quiz phases because during actual development, there is nothing preventing a programmer from working with the code directly after looking through its history. Finally, we interviewed participants about their experience with both history view types.

It is important to note that participants in our study only had access to code histories, not the code bases themselves. We did this to ensure that participants focused on studying the historical information, preventing them from defaulting to program comprehension habits such as studying code which would be identical regardless of the historical information format attached to it. This allows us to more cleanly compare the participants' performance with each view type. We discuss this decision further in Section VII-B2.

C. Questionnaires

We created a questionnaire for each of the two code bases that was independent of view type. Each questionnaire included questions regarding 1) factual information about the code base, 2) historical information about the code base, 3) code snippet interpretations, 4) attitudes towards reuse, and 5) mental effort. We show samples of each question type in Figure 3. For multiple-choice questions about factual information, historical information, or code interpretation, the questionnaire also asks participants to rate their confidence on a 5-point Likert scale from "Strongly Uncertain" to "Strongly Confident".

1) Factual Questions: Factual questions were designed to measure participants' ability to recall specific facts about each code base, including its features and implementation. Figure 3-2 shows an example of a factual question. This is a factual question because it asks about a specific, isolated detail regarding what was implemented in the code, and did not have other development factors impacting it.

2) Historical Questions: We used historical questions to measure participant's ability to recall time-dependent facts about the evolution of each code base. These questions asked about the relationship between multiple events during the development process, such as an error in response to an implementation choice, or a data structure selection in response to a particular result or error. Research shows that developers have difficulty in answering questions about the intentions about design decisions [4], so these questions were aimed at teasing out *why* design decisions were made, or how the code changed over time, especially in response to unexpected obstacles. An example of a historical question is shown in Figure 3-3. This is a historical question because it asks about the resultant error from an initial design choice, so it is part of a longer process.

3) Code Interpretation Questions: Previous work has shown that programmers have difficulty in tying high-level functionalities to their concrete implementations during software reuse [8], [62]. So, we also asked participants to find implementations of high-level functionalities mentioned in the history. Participants picked out which snippet they believed was responsible for a given behavior. An example of a code interpretation question is shown in Figure 3-1.

These questions were designed not to be self-evident; it was not easy for participants to reason through the snippets in order to select the correct answer. Instead, they had to utilize historical information in order to successfully reason about what the correct code snippet does. For example, in the code snippet in Figure 3-1, the correct answer is **C**. The other snippet options are incorrect: **A.** filters results, **B.** iterates through 20 results from a response, and **D.** defines a Google Map instance. Participants are only able to figure out the correct answer if they remember that the developer introduced a delay to successfully load all API search results.

4) Attitudes towards reuse: In order to explore how the history format impacts the perceived accessibility of code bases, we asked participants questions about their attitudes towards reusing each code base after studying the history only. These questions were in Likert format, where lower values

TABLE I
PARTICIPANT RAW AND ADJUSTED ACCURACY (HISTORICAL AND FACTUAL), CONFIDENCE ERROR, MENTAL EFFORT, AND ATTITUDES TOWARDS REUSE

	Raw				Residuals of OLS Controlling for Task Type, Order			
metric	**List mean (std)**	**Story mean (std)**	**Wilcoxon stat**	**Wilcoxon p**	**List mean (std)**	**Story mean (std)**	**Wilcoxon stat**	**Wilcoxon p**
Confidence Error	1.506 (0.386)	1.052 (0.373)	14	0.0034	0.227 (0.352)	-0.227 (0.370)	11	0.0017
Mental Effort	3.537 (0.862)	3.137 (0.695)	27	0.1093	0.200 (0.821)	-0.200 (0.666)	36.5	0.1046
Accuracy	0.669 (0.110)	0.778 (0.110)	18	0.0076	-0.055 (0.108)	0.055 (0.110)	21	0.0131
Factual Accuracy	0.759 (0.144)	0.798 (0.163)	59	0.6685	-0.020 (0.084)	0.020 (0.120)	47.5	0.3225
History Accuracy	0.637 (0.262)	0.851(0.152)	30	0.0507	-0.107 (0.161)	0.107 (0.161)	5	0.0003
Attitudes Avg	3.618 (0.889)	3.625 (0.736)	58.5	0.932	-0.003 (0.878)	0.003 (0.729)	64.5	0.8999

are linked to more positive attitudes towards reuse. Figure 3-4 shows an example of this type of question. Participants answered these questions before the code snippet questions, so that they did not see any actual code before answering this question, ensuring that their perceived difficulty of reuse was based strictly on the history view they studied.

5) Mental Effort: Finally, at the end of each quiz, participants completed the NASA-TLX Mental Effort survey to estimate their cognitive load throughout the task.

D. Analysis

In order to prepare our data for statistical analysis, we define a meaningful metric for evaluating participant confidence and describe our process to control for covariate variables.

1) Confidence evaluation: Confidence is not always a good thing. For instance, being confident about an incorrect answer can lead to invalid assumptions about a code base, which can ultimately lead to 'insurmountable barriers' during code reuse [63].

To explore the degree to which participants were appropriately confident, we defined a metric to favor high reported confidence scores when the participant is correct, and low confidence scores when the participant is incorrect. This essentially measures confidence error by combining *Do programmers know the answer?* and *Do programmers know if they know the answer?*.

We calculate the confidence error C based on whether or not each question was answered correctly or not, as shown in the equation below:

$$E_{\text{confidence}} = \begin{cases} 5 - C_{correct} & \text{if answer is correct} \\ C_{incorrect} - 1 & \text{if answer is incorrect} \end{cases}$$

Research in retrospective metacognition has leveraged similar confidence error metrics [64], [65], [66], and other research suggests that self-regulated learning can be improved with higher retrospective metacognition [65], [67]. For each participant, we calculate this confidence error for every question. We include this metric to better capture how aware programmers are of their understanding of a code base, as this is likely to lead them to more efficiently invest their time during program comprehension.

2) Controlling for covariate variables: We note that the task type and task order may have had an effect on each of our dependent variables. In order to control for the covariate effects of task order and task type, we fit an Ordinary Least Squares model to each dependent variable using these two covariates, and compared the residuals of the model across list-view and story-view conditions using the same statistical methods as for the raw data. A guide to interpreting these values is included in Appendix E. In addition, we present stratified data visualizations to demonstrate the covariate effects.

3) Statistical Testing: For each participant, we calculated accuracy (%) across factual, historical, snippet interpretation questions, as well as accuracy across the entire set of questions for a given code base. We also calculate average confidence error for each participant, as well as average scores for attitudes towards reuse questions and mental effort. We then compare the set of the 16 participants' accuracy, confidence, attitude, and mental effort scores for their list-view test to their story-view test using the Wilcoxon signed-rank test. We do this for both the raw data and for the covariate-adjusted residuals.

We use the Wilcoxon signed-rank test to evaluate significant differences because we are comparing repeated measures across conditions of a moderately small sample size (16), and the majority of our measured distributions do not fit the assumption of normality required by other statistical tests, such as MANOVA. We report values for effect size as well as p-values for each metric. We note that since we compare accuracy in three groupings (all, factual, historical), we interpret the p-values with respect to the Bonferonni correction.

VI. RESULTS

We find that participants are 16% more accurate and have 30% less confidence error recalling historical information when studying the story-view compared to the list-view, especially when recalling time-dependent historical questions, and that there are no significant differences between participants' time studying or recalling historical information, perception of reusability of underlying code base, or mental effort between participants when using the story-view and the list-view (see Table I).

We begin this section by describing the effects of confounding variables, and then present our results in further depth within the context of our research questions.

A. Confounding Variables

We note that two covariate factors: task-type and task-order had an impact on metrics for individual user tasks. Figure 4 shows distributions for overall question accuracy, confidence

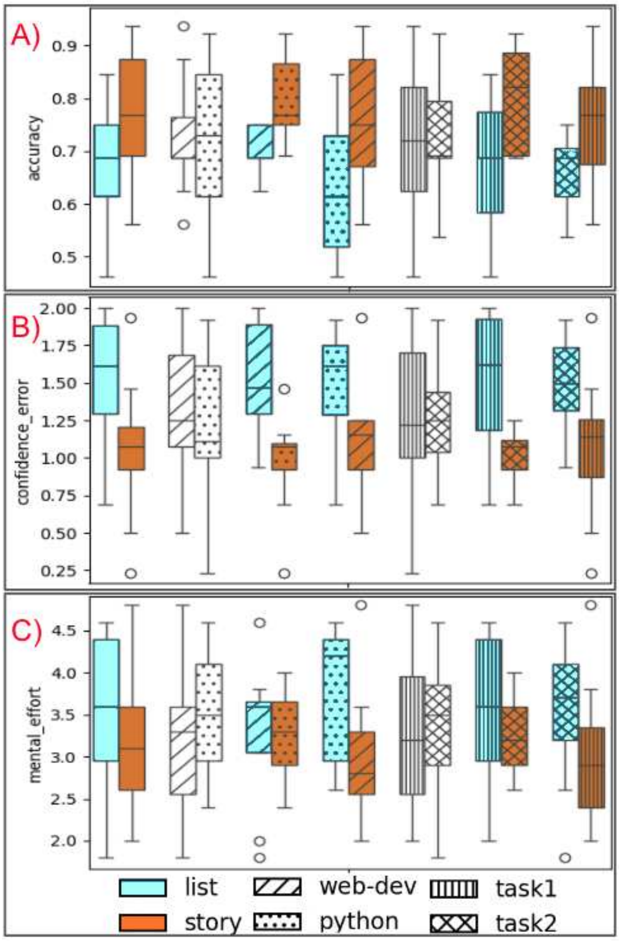

Fig. 4. Box plots of A) accuracy, B) confidence error, and C) mental effort stratified across the independent variables: view-types, tasks, and task order. Adjacent pairs of distributions represent within-subjects comparisons, however, between-subjects comparisons are also visualized. Full discussion of the confounding variables is in Section VI-A.

error, and average reported mental effort across 1) history format type, 2) task type, and task order. For each metric, this figure also visualizes the interaction of history format type with both task type and task order.

The graph displays data in pairs of boxplots, each representing contrasting within-subject distributions. This representation also allows for between-subject comparisons. For example, the fifth and sixth boxplots for any metric represent the eight users who used the list-view for the web-dev task and the story-view for the Python task. Conversely, boxplots seven and eight represent the remaining eight users who were given the list-view for the Python task and story-view for the web-dev task. A between-subject comparison, such as comparing column five (list-view, web-dev) with column eight (story-view, web-dev), illustrates the differences in user metrics between the two view formats for a specific task.

Figure 4 shows that participants are more accurate, have less confidence error, and experience lower mental effort when studying the story-view (orange) than when studying the list-view (blue) throughout stratification of task types and task orders. For example, Fig 4-A shows that participants consistently had higher response accuracy when using the story-view.

We note that in the python task, there is more variance in participants' accuracy (Fig 4-A), and higher reported mental effort (Fig 4-C). We also note a tightening effect between task 1 and task 2; across all metrics shown in Figure 4, the distribution of measurements tightens during task 2. This is likely due to participants acclimating to the task conditions and involving less overhead mental processing costs of getting acquainted to the task structure.

B. *Information Recall*

Here we address **RQ1**: *Are participants better at recalling historical code information when studying the story view?*

In general, participants had a higher accuracy on questions after studying the story-view code history. We note that task-type and task-order may have impacted participants' accuracy averages distributions. In Figure 4, we visualize the distributions of participant accuracy averages across all tasks, as well as the various interactions between view-type and task-type or type-order.

Regardless of task-type and task-order, 12 (75%) participants scored a higher accuracy when using the story-view, while the remaining 4 (25%) scored lower accuracy. For factual questions, 9 (56%) participants' accuracy increased when using the story compared to their list-view task, and 7 (44%) decreased. Finally, for historical questions, 11 (69%) participants' accuracy increased when using the story-view compared to their list-view task, while the remaining 5 (31%) performed slightly worse when using the story-view.

When comparing multiple-choice question accuracy over all questions, participants performed significantly better ($p < 0.016$) in the story-view condition than in the list-view condition. After studying the list-view of a code's history, participants averaged 67% accuracy on the multiple choice questions. After studying the story-view, participants answered correctly 78% of the time.

When answering factual questions, participants in list-view and story-view conditions performed similarly, at 76% and 80%, respectively. In contrast, participants performed better at answering historical questions during the story-view (85% accuracy) than after studying the list-view (64% accuracy). The difference in historical accuracy between list-view and story-view was significant when controlling for task type and task order ($p < 0.016$). We observed no significant difference between conditions for code interpretation questions. Resulting average raw scores, adjusted scores, and significance tests for each question type are shown in Table I.

Participants' sentiments aligned with our finding that they performed better at remembering time-dependent information when using the story-view. When asked to comment about their experience during the user test, User 10 reflected, "In terms of order in which [the developer] made the [web code base], in [the story-view], the order in which they did things was really clear to me, while in the [list-view] one, it was a little bit harder to remember like, the order in which things were implemented."

When describing why they had a harder time remembering data structures referenced in the list-view history, User 6 echoed a similar sentiment when they argued that the structure of the story helped them remember why a particular data structure was implemented, typically in response to a prior design decision not working out. They continued, "but [the list-view] is like 'search this', 'implemented that', 'search this', 'implemented that', so it was also harder to remember the order in which things happened because there wasn't that story behind it."

C. Confidence in Quiz Responses

Next we address **RQ2**: *Are participants more aware of their correctness when recalling historical information when studying the list or story view?*

Participants demonstrated more certainty in their responses when using the story-view than when using the list-view, having a significantly lower ($p < 0.01$) confidence error in the story condition, indicating that story-view users generally had higher confidence when correct, and lower confidence when incorrect. Participant confidence error averages are shown in Table I, and their distributions are visualized in Figure 4.

User 15 described a level of uncertainty they experienced when recalling information from the list-view history. They recounted: "When I'm thinking about [the list-view], I'm just recalling the facts, rather than thinking through the story I read. I also feel less confident because I'm just thinking about the facts, and sometimes I feel less confident because I just think about the fact and I'm like 'okay do I remember it back well, or not?'".

D. Attitudes towards reuse

Thirdly, we report on **RQ3**: *Do participants perceive code bases as more reusable when studying the list or story view?*

We found no significant differences in attitudes towards reuse between the view types. Average scores and statistical tests are reported in Table I. Note that in this scale, lower scores represent more positive attitudes. In general, participants noted that both formats provided valuable information about the history of the code, but were unlikely to feel comfortable about reusing the code without seeing the code bases themselves. User 8 summed up this experience concisely: "It doesn't matter the format, I don't feel confident modifying [the code] because I haven't done it before."

Participants mentioned that if they were required to reuse either code base, many would want access to both formats, finding benefits to each one: the list-view for quick, concise reference to specific details as-needed, and the story-view to give an overall sense of what is in the code, and to bring out the intentions of the original developer. User 15 summed up this paradigm, "I feel like if I was going to modify the code bases, I want to look at both. I want the story to understand what the developer wants done, and how they intended to structure things, but I want the log so that I can do in-depth technical analysis for actually making a change". User 15 continues, "In terms of actually understanding the code, I don't feel anywhere near as confident [with the list-view] compared to the [story-view] style. Because like with [the list-view], I see, okay, they made these changes, a little bit of why they made the changes, but it's mostly 'here's all the changes they made. Here's the things the code's doing.' But, I don't necessarily see as much intrinsic motivation for why they're doing one change or another, versus like, with the story, I can see that."

User 11 also highlighted the strengths of each format: "the [story-view] is definitely easier to follow, and it's easier to like, retain information. I think you really 'get' behind what the thought process was and like, how someone develops code. The [list-view] one, it's more concise but it's harder to get in a person's head". When asked why they might want to get in the original programmers' head, User 11 responded "If the code works? Maybe not, but definitely debugging will help a lot. Like, 'Why do you have this? Oh, you were trying to go down this path? Okay, like this makes more sense'. Or maybe you're trying to modify code and be like, 'Why is this here?' It's like: 'Oh, this was just to make it faster. Okay. Well that doesn't matter to me, I can adjust this'".

E. Mental Effort

Finally, we address **RQ4**: *Do participants experience a higher cognitive demand when studying the list or story view?*

Participants reported slightly lower mental effort when using the story-view, however, this trend was not statistically significant (p = 0.11). Mental effort averages for each condition are listed in Table I and box plots are shown in Fig 4. 11/16 (69%) of participants reported lower mental effort when using the story-view, 3 (19%) reported higher effort, and the remaining 2 (13%) reported no difference. User 2 described the story to be a more natural means of collecting information: "Like, [the story-view] just reads in my mind faster than just seeing the coding steps, but that might also be because I'm just more familiar with regular English words than coding languages".

Overall, participants benefited from the familiarity of the list-view format, and the relatability of the story-view. User 8 described this perfectly when describing how the list-view was more similar to tools they were already familiar with. They described the list-view as, "slick-quick access to information, more organized, and easier to parse." They continued, "It's definitely more standard, I mean everything is just more standardized, more like an encyclopedia or something". Although User 8 enjoyed the consistency of the list-view, they indicated higher engagement with the story-view, reflecting, "Yeah, I feel the story one was nice. It was nice to read. It was fun to read. And then like that compared to the [list-view] when it was like just the the boring, bunch of abstract stuff I don't understand, I just felt very apathetic in a way".

User 15 had a similar description of the two formats, stating "I'm not having to parse a list, analogy wise, it's like if I were reading a textbook vs. reading a fiction book. Reading the list feels like reading a dense textbook or git commit history, reading the story is like I'm reading someone's description of how they're doing this. it feels like less effort even if it's containing similar information".

Finally, User 14 hinted that the story reduced the effort needed to apply their imagination in order to empathize with the original developer, when reading the story-view: "I don't need to add so much imagination to see what is going on with [the developer] for when they were programming the code."

Summed up, our findings are suggestive that storytelling may help programmers apply less cognitive effort to relate with the original developer, but more work with a larger subject pool is necessary to provide more evidence.

VII. Discussion and Future Work

Here we discuss the implications of our results, and how further work could investigate the impact of storytelling within a broader range of developer workflows.

A. Improved Understanding of Code History

Our results suggest that storytelling may help programmers clarify the relationships between historical development events by highlighting the interactivity of the distinct information searches, web visits, and code changes. This may help programmers work with unfamiliar code. On one end, improved confidence in program comprehension may empower programmers to modify code [68], a critical step in making sense of unfamiliar code [52]. On the other hand, having reservations about assumptions that are actually incorrect may help programmers avoid being stuck by "insurmountable barriers" [63] when working in a new code base. Some work suggests that code histories should be integrated directly into IDEs to support programmers in their natural workflow [11], [56], [8]. Future work should consequently investigate including stories within the IDE themselves to explore their utility to developers in their natural workflow.

1) Supplementing Current History Formats: Participants valued the story-view for connecting how distinct developers actions are related through time, and the list-view for being succinct and similar to existing tools, like Git. Our results suggest that stories are not a replacement for traditional list-view style code change histories, but could supplement them to help programmers relate to the original developer. Future work should also explore the story format across different problem-solving styles [69].

2) Expanding to other code history scopes: In this study, we focused on code stories for smaller, personal projects for three reasons, 1) the agency developers have to make decisions, 2) the emphasis on the direct transfer of knowledge from one developer to another, and 3) the feasibility of capturing a project of this scope. However, developers working or onboarding onto larger projects could also benefit from storytelling. Future work should investigate storytelling as it scales to larger, distributed projects, or as it is applied to other levels of historical granularity, such as summarizing git commits, pull requests, and production version updates. Finally, while we used web history and labeled code changes to create histories, future work should include incorporating other historical development information into the story, such as functional requirements changes, formalized testing episodes, versioning conflicts, and think-aloud data [70].

B. Limitations

1) Measuring Memory Retention: A threat to construct validity in our study is that we measure and report data about participants' factual memory retention of a code history. In practice, developing an understanding of how a piece of code was written has a complex relationship with making meaningful edits. Further, including a delay or distractor task may be more appropriate in assessing programmers' recollection of facts, and should be investigated in future work. Additionally, future studies should investigate storytelling code histories through a sensemaking lens.

2) Programmers did not have access to code bases: A threat to external validity is that participants in our study only had access to code histories, not the code bases themselves. We chose to focus on the summary views to ensure that participants were reasoning from these views alone. Further work should include the code bases as well in order to explore how programmers can leverage insights from the story and list views when modifying programs.

3) Activity Labeling: While we mimicked git-style commit messages to label activities, a breadth of potential labeling styles could be applied to developer actions. As with git commit messages, these descriptions are subjective to their author, which may impact external validity. Future work should investigate a more diverse set of labeling styles, and also investigate using generative AI to author and organize code change labels.

Finally, it is important to note that storytelling can vary across people, cultures, and media. Our work is only a small step in discovering the effects of storytelling on programmer's understanding of software histories. Further work should investigate the impact of a diverse set of media and narrative structures on programmer understanding of code history.

VIII. Conclusion

Programmers performed 16% better on memory recall tasks when given access to story-formatted information, and 34% better on questions which involved the time-dependent relationships with other pieces of information. Story users were 30% more accurate in assessing their correctness during the story condition than in the non-story task. Participants valued both formats for different reasons. Programmers valued the list-view for offering concise descriptions and that it is similar to code history formats they were already familiar with, like Git. Programmers valued the story format for piecing together seemingly unrelated code changes into a larger narrative, its ease of reading, and for fostering empathy for the original developer. These findings indicate that code stories should complement, but not replace other versioning systems. Programmers experienced slightly less mental effort when using the story-view but this was not statistically significant. Finally, future work should investigate code history stories in actual programming contexts.

REFERENCES

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] E. Greene, *Storytelling: art and technique: art and technique*. ABC-CLIO, 1996.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] K. Erdos and H. M. Sneed, "Partial comprehension of complex programs (enough to perform maintenance)," in *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No. 98TB100242)*. IEEE, 1998, pp. 98–105.

[24] 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.

[25] I. F. Goodson, G. Biesta, M. Tedder, and N. Adair, *Narrative learning*. Routledge, 2010.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

[33] C.-M. Hung, G.-J. Hwang, and I. Huang, "A project-based digital storytelling approach for improving students' learning motivation, problem-solving competence and learning achievement," *Journal of Educational Technology & Society*, vol. 15, no. 4, pp. 368–379, 2012.

[34] 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.

[35] Q. Burke and Y. B. Kafai, "Programming & storytelling: opportunities for learning about coding & composition," in *Proceedings of the 9th international conference on interaction design and children*, 2010, pp. 348–351.

[36] S. M. Kromka and A. K. Goodboy, "Classroom storytelling: Using instructor narratives to increase student recall, affect, and attention," *Communication Education*, vol. 68, no. 1, pp. 20–43, 2019.

[37] S. Bolkan, "Storytelling in the classroom: Facilitating cognitive interest by promoting attention, structure, and meaningfulness," *Communication Reports*, vol. 34, no. 1, pp. 1–13, 2021.

[38] C. R. Lucarevschi, "The role of storytelling on language learning: A literature review," *Working Papers of the Linguistics Circle*, vol. 26, no. 1, pp. 24–44, 2016.

[39] I. McGregor and J. G. Holmes, "How storytelling shapes memory and impressions of relationship events over time." *Journal of personality and social psychology*, vol. 76, no. 3, p. 403, 1999.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] A. E. Hassan, "The road ahead for mining software repositories," in *2008 frontiers of software maintenance*. IEEE, 2008, pp. 48–57.

[45] 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.
[46] 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.
[47] E. J. Conklin and K. B. Yakemovic, "A process-oriented approach to design rationale," *Human–Computer Interaction*, vol. 6, no. 3-4, pp. 357–391, 1991.
[48] S. Mehrpour and T. D. Latoza, "A survey of tool support for working with design decisions in code," *ACM Computing Surveys*, vol. 56, no. 2, pp. 1–37, 2023.
[49] K. A. Alsafwan, "Toward better understanding and documentation of rationale for code changes," 2023.
[50] J. Lee, "Design rationale systems: understanding the issues," *IEEE expert*, vol. 12, no. 3, pp. 78–85, 1997.
[51] D. M. Arya, J. L. Guo, and M. P. Robillard, "How programmers find online learning resources," *Empirical Software Engineering*, vol. 28, no. 2, p. 23, 2023.
[52] C. Kelleher and M. Brachman, "A sensemaking analysis of api learning using react," *Journal of Computer Languages*, vol. 74, p. 101189, 2023.
[53] G. Rousseau, R. Di Cosmo, and S. Zacchiroli, "Software provenance tracking at the scale of public source code," *Empirical Software Engineering*, vol. 25, pp. 2930–2959, 2020.
[54] 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.
[55] 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.
[56] 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.
[57] A. Horvath, A. Macvean, and B. A. Myers, "Meta-manager: A tool for collecting and exploring meta information about code," 2024.
[58] D. Fontanet Losquiño and T. Urdell, "Why do developers struggle with documentation while excelling at programming," B.S. thesis, Universitat Politècnica de Catalunya, 2014.
[59] 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.
[60] C. Coder. (2022, Feb) Live coding a wordle clone (5 hrs) — html sass js. YouTube. [Online]. Available: https://www.youtube.com/watch?v=PNGgQzw6PQg
[61] R. L. Roper, "Does gender bias still affect women in science?" *Microbiology and Molecular Biology Reviews*, vol. 83, no. 3, pp. e00 018–19, 2019.
[62] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.
[63] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 2004, pp. 199–206.
[64] M. Siedlecka, B. Paulewicz, and M. Wierzchoń, "But i was so sure! metacognitive judgments are less accurate given prospectively than retrospectively," *Frontiers in psychology*, vol. 7, p. 171892, 2016.
[65] Z. Ç. Özcan, "The relationship between mathematical problem-solving skills and self-regulated learning through homework behaviours, motivation, and metacognition," *International Journal of Mathematical Education in Science and Technology*, vol. 47, no. 3, pp. 408–420, 2016.
[66] J. Kruger and D. Dunning, "Unskilled and unaware of it: how difficulties in recognizing one's own incompetence lead to inflated self-assessments." *Journal of personality and social psychology*, vol. 77, no. 6, p. 1121, 1999.
[67] A. M. Robey, M. R. Dougherty, and D. R. Buttaccio, "Making retrospective confidence judgments improves learners' ability to decide what not to study," *Psychological Science*, vol. 28, no. 11, pp. 1683–1693, 2017.
[68] J. Eliasson, L. K. Westin, and M. Nordstrom, "Investigating students' confidence in programming and problem solving," in *Proceedings. Frontiers in Education. 36th Annual Conference*. IEEE, 2006, pp. 22–27.
[69] M. Guizani, I. Steinmacher, J. Emard, A. Fallatah, M. Burnett, and A. Sarma, "How to debug inclusivity bugs? an empirical investigation of finding-to-fixing with information architecture," 2020.
[70] R. Krosnick, F. Anderson, J. Matejka, S. Oney, W. S. Lasecki, T. Grossman, and G. Fitzmaurice, "Think-aloud computing: Supporting rich and low-effort knowledge capture," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–13.

APPENDIX

A. GPT-4 Prompting and Output Examples

The research team used GPT-4 through ChatGPT's web interface with no hyperparameter tuning. We experimented with various story-creation prompting techniques, and ultimately landed on the prompt structure below, which in this example specifies data for the app used in the demo code history.

1) Prompt: I am going to send a list of goals and actions a programmer took while making an app to find top-rated restaurants in a given area, and I want you to create a short story describing the main phases of development the software underwent. The story should be written in a way that helps you empathize with the original developer's challenges, but not be overly whimsical.

The format is in the form:
begin example
Goal 1:
Action 1
Action 2
Goal 2:
Action 3
Action 4
Action 5
end example

Goals will be high level goals that include multiple actions. Actions will include changes to specific files, or web searches for information. Please include all footnotes to reference facts in your story with their original source "Actions". Here is the data:

- **Goal: Plug into Google Maps API, query for restaurants within search radius of St. Louis**
  - search: what is my ip;
  - search: google maps change ip address api access;
  - revisit: Google Cloud Platform;
- **Goal: Get more than the first page of 20 results returned by Maps API for search radius**
  - search: why does google maps api only return a few results;
  - `example.py`: Adjust loop counter using `tokenCount`
  - `example.py`: Checking to see what the response from Maps API is
  - `example.py`: Add delay between pages of results, so page would not be empty
  - search: google maps api get more than 60 restaurants;
  - `example.py`: Add longer delay and check response
  - `example.py`: Checking the value of `next_page_token` from Maps response

- **Goal: Break search radius circle into overlapping subcircles in order to pull more results**
  - search: use google maps api to get a total count of restaurants in city;
  - search: meter to longitude and latitude conversion;
  - search: circles and angles overlap playground;
  - `example.py`: Defining a subcircle to be used to represent part of the larger radius more finely
  - search: equal circles with 60 degree overlap;
  - search: read from file in javascript;
  - search: google maps change API key;
  - `example.py`: Convert meters to latitude and longitude differences
  - `example.py`: adjust latitude and longitude to be placed north and east of starter circle, instead of south and west
  - `example.py`: shift every other column of subcircles horizontally to cover area more efficiently
  - search: 30 60 90 triangle;
  - revisit: Algorithm - Circle Overlapping - Mathematics Stack Exchange;
  - `example.py`: adjust distance between subcircles based on hand calculations
  - search: how many degrees in circle;
  - revisit: https://chat.openai.com/;
  - `example.py`: Loop through 5X5 grid and print where centroids of subcircles should be
- **Goal: Plot found restaurants by number of reviews and average review score**
  - revisit: Google Maps;
  - `visualizeData.py`: Plot found restaurants by `user_ratings_total` and average rating
  - search: numpy log;
  - `visualizeData.py`: attempt to make scatterplot interactive, to see the label associated with each data point
  - revisit: Google Maps;
- **Goal: Filter for "good" restaurants and add labels to scatterplot**
  - `visualizeData.py`: filter restaurants by `min_rating` and `min_reviews`, and annotate the datapoints with labels
  - `visualizeData.py`: calculate z-score for rating and num_reviews
  - `visualizeData.py`: only display 90th percentile outliers for rating and num reviews
  - `example.py`: adjust search radius and number of subcircles
  - revisit: Google Cloud Platform;
  - `example.py`: adjust centerpoint of search radius
  - revisit: Google Cloud Platform;
  - search: google maps types of places;
  - `visualizeData.py`: fixed typo in csv file name

B. GPT-4 Raw Output

In the bustling heart of St. Louis, a programmer named Alex sought to solve a culinary conundrum: finding top-rated restaurants in the area. Like any software developer embarking on a new project, Alex was about to undergo a roller-coaster of discovery, frustration, and innovation.

Alex's first hurdle[1] was the Google Maps API, which would be the backbone of his application. After entering the search command, 'what is my ip,'[1] he wondered if the change of his IP address affected API access and hence, looked up 'google maps change ip address api access.'[2] He then revisited Google Cloud Platform[3] to make sure he was in the clear. It was the typical back-and-forth dance familiar to developers everywhere, adjusting parameters, revisiting platforms, and laying the first bricks of the project.

His elation, however, was short-lived. As he pulled the restaurant data, Alex noticed that the API was only serving a limited first page of 20 results[4]. Why would this be the case? He dove back into the web with 'why does google maps api only return a few results' to find some answers[5]. Delving into the code of example.py, he tinkered with loop counters[6], examined API responses[7], and incorporated delays[8]. But it was still an uphill battle; he had to increase the delay[9] and pay attention to the `next_page_token` from the API response[10] to get beyond the restrictions.

But Alex was far from done. He pondered how to effectively break the search radius into overlapping subcircles to pull even more results. His journey took him into the depths of geometry, playing with meters, longitude, latitude[12,13], and circle overlaps[14]. Every piece of the puzzle, from converting meters to latitude and longitude differences[17] to adjusting the placements of subcircles[18], was a challenge in itself. With some hand calculations and help from mathematical discussions on Stack Exchange[20,21], Alex was able to finely map out a 5X5 grid where the centroids of these subcircles should be[24].

Having collected a robust set of data, Alex then shifted his focus to visual representation[26]. He began plotting restaurants by number of reviews and average review score in `visualizeData.py`[30]. To give depth to his visualizations, Alex delved into making the scatterplot interactive[32], allowing viewers to see labels associated with each data point. This was a storytelling tool, and it brought the data to life.

However, quality mattered to Alex. He sought only the "good" restaurants. Filtering became a pivotal action, as he adjusted parameters for ratings and reviews in `visualizeData.py`[30]. Through statistical techniques like calculating z-scores[31] and zeroing in on the 90th percentile outliers[32], Alex created a more refined experience for the end-user. And to keep the data fresh and relevant, he made adjustments in `example.py` like modifying the search radius[33,35] and revisiting the Google Cloud Platform[34,36].

The journey was filled with intricate tasks, from fixing typographical errors in file names[38] to deciphering the different 'types of places' in Google Maps[37]. But through the amalgamation of these detailed actions, Alex crafted a software tool that would guide many food enthusiasts to the best spots in St. Louis.

Footnotes:

1) search: what is my ip;
2) search: google maps change ip address api access;
3) revisit: Google Cloud Platform;
4) search: why does google maps api only return a few results;
5) `example.py`: Adjust loop counter using `tokenCount`;
6) `example.py`: Checking to see what the response from Maps API is;
7) `example.py`: Add delay between pages of results, so page would not be empty;
8) `example.py`: Add longer delay and check response;
9) `example.py`: Checking the value of `next_page_token` from Maps response;
10) search: use google maps api to get a total count of restaurants in city;
11) search: meter to longitude and latitude conversion;
12) search: circles and angles overlap playground;
13) `example.py`: Defining a subcircle to be used to represent part of the larger radius more finely;
14) search: equal circles with 60 degree overlap;
15) search: read from file in javascript;
16) `example.py`: Convert meters to latitude and longitude differences;
17) `example.py`: adjust latitude and longitude to be placed north and east of starter circle, instead of south and west;
18) `example.py`: shift every other column of subcircles horizontally to cover area more efficiently;
19) search: 30 60 90 triangle;
20) revisit: Algorithm - Circle Overlapping - Mathematics Stack Exchange;
21) `example.py`: adjust distance between subcircles based on hand calculations;
22) search: how many degrees in circle;
23) revisit: https://chat.openai.com/;
24) `example.py`: Loop through 5X5 grid and print where centroids of subcircles should be;
25) revisit: Google Maps;
26) `visualizeData.py`: Plot found restaurants by `user_ratings_total` and average rating;
27) search: numpy log;
28) `visualizeData.py`: attempt to make scatterplot interactive, to see the label associated with each data point;
29) revisit: Google Maps;
30) `visualizeData.py`: filter restaurants by `min_rating` and `min_reviews`, and annotate the datapoints with labels;
31) `visualizeData.py`: calculate z-score for rating and num_reviews;
32) `visualizeData.py`: only display 90th percentile outliers for rating and num reviews;
33) `example.py`: adjust search radius and number of subcircles;
34) revisit: Google Cloud Platform;
35) `example.py`: adjust centerpoint of search radius;
36) revisit: Google Cloud Platform;
37) search: google maps types of places;
38) `visualizeData.py`: fixed typo in csv file name;

C. Post-Generation Edits

The research team manually edited GPT-4's output to ensure the protagonist of the story remained ungendered, to avoid gender bias in the study participants. Additionally, when specific actions were omitted from the story-view, the research team manually added them into the story. For example, the story-view only references 38 actions, whereas the input included 40 actions. Afterwards, the research team converted the story into a simple website to give programmers access to the story. In doing so, the team removed the superscripts and instead added links for sentences which referenced developer actions. While this represents a manual bottleneck for the creation process, these edits are much quicker to make than crafting a story from scratch. Further, experimenting with GPT-4o highlighted its significantly improved ability of incorporating all developer actions into the generated story.

D. Activity Segmentation Heuristic Creation

Testing segmentation requires a set of code histories that was not readily available. In order to create these heuristics, we first created a validation set of 6 programs. For each program, we observed its creation in its entirety, and recorded when the developer tested their code. At each test, we recorded the program state through either a code recording software we developed or from a process of OCR and manual edits to capture the code from livestreamers' videos. Upon observing that developers commonly went through iterations of rapid changing of a given section of code, and then testing between each change, we decided to define a heuristic which groups these localized changes together. Through several iterations of defining heuristics, we were satisfied with the resulting grouping for the six validation programs. The team underwent a similar process in grouping web searches together; we found that developers often made multiple, similar searches before finding a web resource. Through several rounds of heuristic testing, our process of grouping web searches together based on common keywords appeared to appropriately address this phenomenon.

1) Final Heuristic Segmentation Algorithm: The final heuristic segmentation algorithm iterates through the set of code states at each test, and analyzes the changes made between states, appending consecutive code changes to the same segment of activity if they were made in the same file and any of the following conditions applied:

- One or more code lines are modified, but there are no additions of new lines or deletions of lines
- Up to three lines are added or deleted from the code state
- No changes or only white space changes are made

If none of these conditions were met, or the code change happened in a different file than the previous change, a new cluster of code changes would begin. We noted that developers often sketched out a new section of functionality with the insertion of new lines of code and then modified and debugged those lines. Thus, the limitation on the number of new lines helps to capture the beginning of new functionality.

When grouping sequential web searches together, we relied on a process of comparing sequential searches as bags of words. If two consecutive searches had an overlap of keywords, then we included the sequence of events into one search cluster. We excluded traditional stop words as well as common programming words like "javascript", "python", and "documentation". This simple approach quickly allowed us to group semantically similar searches together.

E. Controlling for Covariates: Supplemental Interpretation

Additionally, Table I shows statistical comparisons for the various metrics using both raw data, and residuals from a linear regression controlling for the covariates. Average residuals of each linear regression are equal and opposite between story and list view groups. This is because we balanced our data to have equal counts of task order and task type for each history view condition. Since the line of best fit will always have an average residual of 0 (all residuals cancel out), when we look at the average residual for the two conditions, each having the same count of covariate values, their averages are equally far from zero in opposite magnitudes. Positive average magnitudes in Table I indicate measurements that were higher than the estimations given the covariates, and negative average magnitudes indicates measurements lower than the regression predictions.