# Software History under the Lens: A Study on Why and How Developers Examine It

Mihai Codoban*, Sruti Srinivasa Ragavan*, Danny Dig*, Brian Bailey†
Oregon State University*, University of Illinois at Urbana-Champaign†
{codobanm, srinivas, digd}@eecs.oregonstate.edu, bpbailey@illinois.edu

***Abstract*—Despite software history being indispensable for developers, there is little empirical knowledge about how they examine software history. Without such knowledge, researchers and tool builders are in danger of making wrong assumptions and building inadequate tools.**

**In this paper we present an in-depth empirical study about the motivations developers have for examining software history, the strategies they use, and the challenges they encounter. To learn these, we interviewed 14 experienced developers from industry, and then extended our findings by surveying 217 developers. We found that history does not begin with the latest commit but with uncommitted changes. Moreover, we found that developers had different motivations for examining recent and old history. Based on these findings we propose 3-LENS HISTORY, a novel unified model for reasoning about software history.**

***Index Terms*—software history, software change management, human aspects of software evolution, user studies**

## I. INTRODUCTION

Software history is indispensable for developers. Of the 217 developers surveyed in this work, 85% find software history important to their development activities and 61% need to refer to history at least several times a day. However, there exists little knowledge on why and how developers examine software history. Without such knowledge, tool builders might not address the needs of developers, and researchers working on software history might make wrong assumptions about developers' true needs.

Version Control System (VCS) tools assume many things regarding developer workflow [1]. For example, they assume that developers (i) plan in advance and produce systemic, cohesive commits, and (ii) remember all the changes that need to be committed when context switching to the VCS tool [2]–[4]. In addition, when developers need to search and understand past changes, current tools assume that all history is equally important, homogenous, and with a flat, non-hierarchical structure. If these assumptions are false, then the tools relying on them risk wasting developer effort.

In this paper we report the reasons why developers examine software history. We document the strategies developers use to examine history and the challenges they encounter in doing so. We then expose the inadequacies in current tool support for software history, and uncover the workarounds and practices that partially help developers in examining history.

To motivate our study, we build upon seminal research [5]–[14] on developers' informational needs. We found that there exists a knowledge gap about why and how developers examine software history. In this paper, we address this critical gap by conducting an in-depth study of developer's usage of software history.

We interviewed 14 software developers with an average of 13 years of professional experience. We then deployed a survey, with 217 respondents from the industry, to quantify and extend the interview findings. We targeted the following four research questions:

**RQ1**: *Why do developers examine software history?* We found that history does not begin with the latest commit but with uncommitted changes. Additionally, we found that participants had different motivations for examining recent and old history (Section III).

**RQ2**: *How do developers use software history?* Developers search through history for three main items: particular commits, recent commits (that may impact their work) and proper commits to revert to. They use different strategies for finding each item (Section IV).

**RQ3**: *What challenges do developers face when examining history?* We found that despite fulfilling many needs, software history tools remain incomplete. More critically, we found that developers have history related needs from uncommitted changes but today's tools largely ignore them (Section V).

**RQ4**: *What helps developers and what do they wish they had?* We found that developers combine several tools and practices that partially help them overcome the challenges they expressed in RQ3. However, developers still express a wide variety of unsatisfied needs that current tools do not adequately support, such as understanding the context of changes and selective change notifications.

Based on the analysis of the data, we found that VCS tools provide a single lens on history. They only provide a *temporal* view of committed changes [15]. However, we found that developers have many needs which are not covered by this single lens. For example, (i) uncommitted changes are not represented, thus forcing developers to keep a mental list of pending changes, (ii) the high volume of recent changes results in information overload, thus causing bugs to go unnoticed, (iii) past related changes are not grouped, making it very tedious to trace them.

Thus, we propose 3-LENS HISTORY, a novel, developer-centric, motivation-based model for software history. Each lens focuses on a different aspect of history based on developers' needs for that specific region: the IMMEDIATE lens manages and structures uncommitted changes, the AWARENESS lens focuses on keeping up with and understanding the latest

changes, and the ARCHAEOLOGY lens recovers knowledge that is buried in software history.

Our results open new research and tool design avenues. Tool builders can make each lens a first class entity and provide explicit support for it while researchers can focus on analyses and visualizations to better augment each lens. For example, the (i) IMMEDIATE lens could automatically group related uncommitted changes, (ii) the AWARENESS lens could selectively notify developers about changes that can affect them, and (iii) the ARCHAEOLOGY lens could present a past version of a code snippet in the context of the broader change.

This paper makes the following contributions:

**1**) We provide an empirically justified set of developers' motivations for examining software history.
**2**) We expose a gap between developers' needs and existing tools for software history.
**3**) We define 3-LENS HISTORY, a novel unified model for reasoning about software history from the developers' perspective.
**4**) We present actionable implications for each lens that researchers and tool builders can build upon.

## II. METHODOLOGY

In this study we aim to understand the motivations that developers have for examining software history. To do this, we employ established methodologies from prior research [8], [16]–[19].

The primary research methodology that we use in this study is interviews with software developers. We then supplement the interview results with a survey. Interviews are a qualitative method: they are used to elicit people's knowledge and experience but often draw from a limited sample size [19]. Surveys on the other hand are a quantitative method: they summarize information over a larger sample and therefore provide increased generality [19]. Thus, we used interviews to elicit developers' experiences with examining software history and built a taxonomy of motivations, strategies, challenges, enabling activities and unfulfilled needs. We then deployed a survey to quantify and further refine the taxonomy.

Buse et al. [5] used a temporal categorization for developers' information needs. We hypothesized that such differences might also exist in developers' usage of software history. To verify this hypothesis, we asked interview and survey participants about their motivations for examining recent and old history. We defined recent history as commits that are contemporaneous, i.e., current iteration or up to about 2 weeks [20], and old history as commits that dated further back.

The interview script, survey questions and raw data can be found on this study's *companion website* [21].

### A. Interviews

We conducted semi-structured interviews [19] because this method's open-ended questions keep the discussion focused while allowing interesting tangents to be followed. We interviewed 14 software developers from 11 companies, with an average experience of 13 years. They use diverse VCS (Git, SVN, TFS, Bazaar) and clients (terminal, Github, Stash, etc.). Each interview lasted between 40 and 90 minutes and the participant was paid $50 at the end of the interview.

The interviews were based on the research questions presented in Section I. The following were some questions we asked during the interviews:

- Describe an instance when you had to examine recent (or old) history (how did you do it, difficulties faced, etc.)
- What is the most important reason for which you examine software history?
- What is the biggest difficulty you encounter when working with software history?
- How do you gain awareness of the code changes happening on your project? (when you come in for work, when you come back from a vacation, etc.)

We coded the transcripts using established guidelines in literature [22] and referred to Campbell et al. [23] for handling specific issues with coding open-ended interviews such as segmentation, codebook evolution and achieving coder agreement.

For each interview, the first author segmented the transcript based on the topics from our research questions. The first and second authors then independently coded the transcripts using the open coding methodology [24]. The isolated coding sessions resulted in agreement levels (Jaccard coefficient) of 65%, levels which are consistent with coding open-ended interviews [23]. The authors then used the negotiated agreement technique to reach agreement [23]: resolving all coding disagreements by refining the codes through the addition, deletion, merger or better description of codes. Each following interview was coded using the codebook from the previous session. It took 10 interviews to stabilize the codes using this process. The previously coded interviews were then re-coded using the improved codes. After coding was complete, the authors grouped the codes into larger emerging themes.

### B. Surveys

We designed a survey with 16 questions to quantify our findings from the interviews. The questions were directly derived from the interview results. To ensure completeness, we also included "other" fields for all questions in case survey respondents had additional insights beyond what we derived from the interviews.

We recruited 217 survey respondents by advertising our survey on social media channels[1] frequently accessed by software developers. Of the respondents, 84% were from industry. Also, 80% of all respondents had more than 5 years of professional experience.

## III. MOTIVATIONS FOR EXAMINING HISTORY

From our interviews we found that software history starts with uncommitted changes. Hence, we define software history as the set of all code changes performed for a project. It consists of all the committed and uncommitted changes made for a project.

[1] reddit/r/programming and Twitter

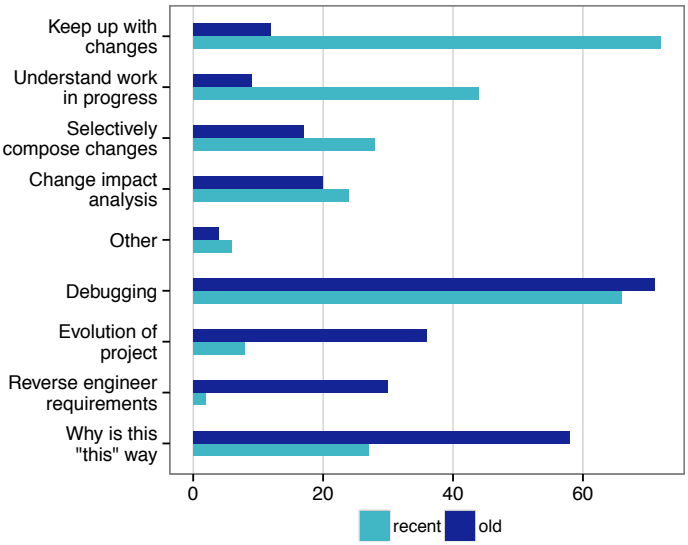


**Fig. 1:** Survey respondents' motivations for examining history (%). The first 3 motivations are predominantly accomplished via recent history while the last 3 via old history.

We noticed that interviewees use recent and old history with different motivations. Figure 1 shows that the survey respondents confirm this hypothesis. For example, they primarily used old history to reverse engineer requirements and recent history to understand their work in progress. We categorize the motivations based on the age of history.

### A. *Motivations for recent history*

We found that the primary motivation for examining recent history is to become aware of the changes that are happening to the project. There are two main categories of changes that they need to keep up with.

*1)* ***What is going on that I need to know about:*** Participants said they need to become aware of the recent code changes that other people are making on the project to keep up with how the code state evolves.

Figure 2 shows that survey participants did not want to be aware of all changes. Instead, they were only interested in specific changes. Some of them were especially interested in breaking changes, changes that might overlap with or affect their work in progress. P5 stated: *"I inspect the current history of the commits that are being made and see if any of them are related to what I am changing."*

From the survey, we additionally found that respondents examine recent history for code reviews and to summarize commits. Summarization involves activities such as writing public changelogs, reporting work at the end of the week or squashing commits down to a single one for code review.

*2)* ***Where do I come from and where am I going:*** When developers work on a task, they sometimes commit their subtasks as intermediary commits. Some of our interviewees said that they often need to read their own commits for their current task in order to understand their work in progress. Participant P5 said: *"..helps to jog my memory as to what are the things there and what were the things that I needed to change."* We also found that sometimes work in progress is split across multiple branches and our participants had to compare branches to see what remained to be done.

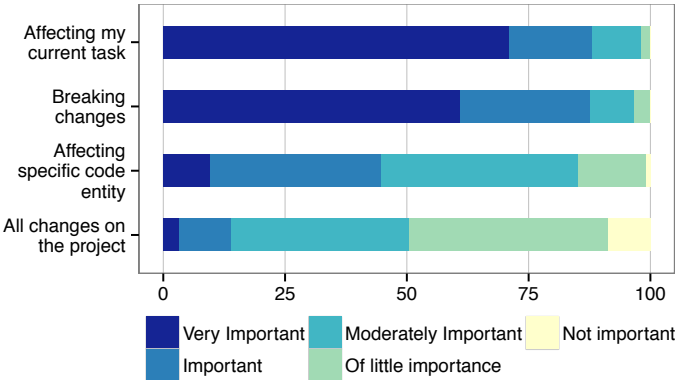


**Fig. 2:** Which changes did survey respondents considered more important to learn about (%).

Interviewees reported that they sometimes reach a dead end in implementation when doing trial and error programming, for example P2 said *"... got about half way down, and I realized it would not work."* When this happens, they undo and revert their changes to a previous working state. This behaviour is known in literature as backtracking [25].

### B. *Motivations for old history*

We found that participants examined old history to regain lost knowledge that is buried deep inside software history.

*1)* ***Why is this "this way":*** Our participants reported that they often needed to examine old history in order to recover the rationale behind a snippet of code. They do this in order to verify that their present changes won't affect existing functionality, to study how a bug came into existence, etc. They also do this to reverse engineer requirements from code.

For each change that affected the code snippet, they also try to understand the change rationale, what broader modification that change was part of, and how the code snippet interacted with the surrounding code at that point in time.

*2)* ***Understand the evolution of the project:*** Some of our participants occasionally use software history to learn how the project evolved. They look for who are the knowledgeable peers on certain modules and patterns such as architectural decisions, requirements decisions, frequent bugs, etc.

### C. *Age-independent motivations*

We found that in some motivations developers make no distinction between old and recent changes.

*1)* ***Debugging:*** Developers make errors while changing code: they introduce bugs, unexpectedly alter the behaviour of the program, or make bad merges. Therefore developers periodically examine history in order to find the problematic commits that introduce issues. They use that specific commit to understand the change, learn how the requirements changed and find the author of the commit. While interviewees mentioned performing this activity for recent history, the survey participants marked it for both old and recent.

*2) **Change impact analysis**:* Interviewees perform change impact analysis on commits to learn what high level modules are affected. They use this information to gain a better mental model of the changes introduced by a commit. They also use this to define what regression tests to prioritize or what modules to rebuild or to manage deployments. Our interviewees mentioned performing this activity for recent history, but the survey participants marked it for both old and recent.

*3) **Selectively compose changes**:* Some of our participants also use software history to revert specific commits or selectively compose changes by applying commits from one branch into another. For example, they cherry pick specific commits from several branches into one branch.

The magnitude of this activity varies from recovering small snippets of code (e.g., hash map initialization values) to entire feature or bug fix commits and vary from recent branches to commits that are several months old.

**Observation 1:** Developers have different use cases for old and recent history. They use: (i) recent history to keep up with other people's changes and to manage their work in progress, (ii) old history to recover program intent and rationale, and (iii) both old and recent history to find problematic commits and to selectively compose changes.

## IV. Strategies

To accomplish the motivations from Section III, participants looked for certain information fragments. We grouped these into three categories: (i) finding specific commits, (ii) change-awareness information, and (iii) safe points for backtracking.

In this section we present the strategies that developers use to gather these artifacts.

### A. *Finding specific commits*

The motivations for old and age-independent history primarily involve finding a specific commit. Participants used three progressive strategies to find a commit.

*1) **Reducing the search space**:* Participants seeked to reduce the number of commits they have to analyze before searching for a specific commit.

In many cases participants searched from known code by browsing through the history of a particular snippet of code. P13 said: *"we had to go back through each version [of a file] and look it over to see when the caching was actually implemented ... Really what we were looking for is a particular line of code and exactly what point in time did it change."*

Participants reported that sometimes they remember the time period in which a commit was made and thus retrieve only the commits performed in a specific date range. P13 stated that: *"I find it much more useful to go by date. Because I know when I was working with this on Monday it was fine but now it's Friday and it's broken. Give me the changes that occurred between these dates."* Participants also filter commits that contain certain keywords or tagged commits.

The common pattern is that participants were only interested in the commits that were related to their current task.

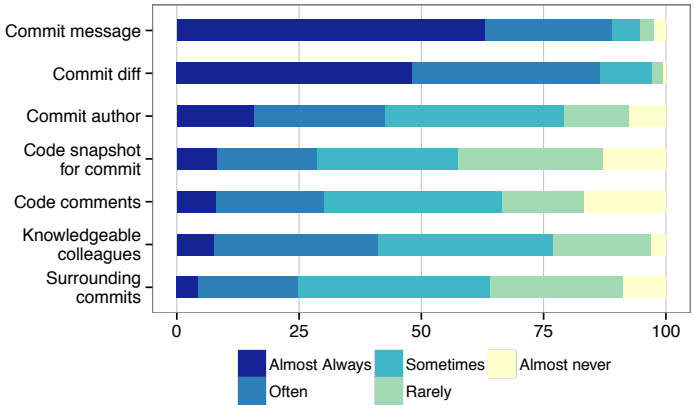


**Fig. 3:** Survey respondents' strategies for understanding commits (%)

*2) **Traversing commits**:* Participants traversed history by quickly skimming over commits in search for the ones with a high probability of containing the information they need.

When identifying a problematic commit that caused a test to fail, participants located the commit by reverting at specific commits and ran the test to check the behaviour. They either reverted commits one by one or employed binary search. Some VCS offer explicit support for binary search (e.g., git bisect).

*3) **Understanding commits**:* Interviewees iterated through several strategies in order to learn the intent of a commit. Figure 3 shows these strategies quantified by the survey in terms of how frequently they are used.

From the interviews we found that commit messages and commit diffs are used while skimming commits whereas the commit author or other knowledgeable colleagues represent the most successful strategy for understanding the deep rationale of a commit . The survey confirms these findings (Fig. 3).

**Observation 2:** Developers find a specific commit in three steps: (i) reduce commit space based on current task, (ii) skim to identify interesting commits and (iii) understand commit intent. They use different strategies for each step.

### B. *Change awareness information*

Interviewees employed different information sources as a strategy for keeping up with changes. Table I shows these sources, quantified by the percentage of survey respondents who used them. Even though history and face-to-face communication are the top sources, survey respondents use multiple sources and there is no clear top choice.

Some participants report a tendency for communication preference, i.e., they do not use history to gain awareness but prefer to ask their peers. For example P2 said *"I can't think of a time I ever had to look through the repository to just see what happened. I would go talk to people."*

When using history participants used a similar strategy to that of finding a specific commit (Sec. IV-A). However they used different methods to reduce the search space: they only look at recent commits or they view grouped related commits from branch diffs, from commits between tags or from platform specific representations (e.g., Github pull requests).

**Table I:** Survey respondents' strategies for keeping up with changes

| | |
|---|---|
| Sync with VCS and skim changes | 68% |
| Face-to-face communication | 67% |
| Code reviews | 60% |
| Emails, documents, etc. | 48% |
| Task log | 48% |
| Automatic notifications and feeds | 41% |
| Other | 2% |

We also observed two particular behaviours. One participant (P12) mentioned doing a periodic digest by reading change emails only at certain moments during a day. Other participants employed a selective change awareness strategy. They read and received updates only about changes they express interest in: when skimming commits, they only looked at *"interesting changes"*, they created email filters for different repositories, and asked peers about changes on specific modules. P12 mentioned: *"I have a couple of folders of email that if I get any email I'll look at it fairly quickly, within like 30 minutes or so. They are different because those are changes that are introduced to an important repo and I want to know fairly quickly if something happened."*

**Observation 3:** Developers don't have a top strategy for keeping up with changes. They use multiple strategies and face-to-face communication is among the top choices.

### C. *Safe points for backtracking*

Developers use history to find good commits to revert to. When doing trial and error development they need to partially or completely undo their uncommitted changes. An important concern that helps developers backtrack is ensuring a steady stream of good commits where they can revert to. P2 gives a detailed account of this strategy: *"I had committed and pushed something that I felt confident in. I finished a feature and I was exploring how to start the next one. I knew when I started that there was a possibility that this would end in disaster so I started writing it thinking that that was a possibility and it turned out that I was correct. So I rolled back to the point where I made the decision that this could possibility go wrong from here. I specifically committed at that point to give myself a return spot so that I would be able to return to it."*

## V. Challenges, solutions, and needs

Along with motivations and strategies, our interview participants reported difficulties in examining software history. They also reported techniques for mitigating some of these challenges, but also wished that tools provided more capabilities. Some of these challenges and mitigation techniques mentioned in this section are specific to certain VCS. We used the survey to quantify both the challenges (Tab. IIa) and the wishes (Tab. IIb).

We categorized the results into four themes: information mess, knowledge fragmentation, understanding history, and tool limitations. We define information mess as the challenges due to the structure and organization of information and knowledge fragmentation as the challenges in retrieving knowledge from multiple systems. In this section we present each theme in detail.

### A. *Information mess*

We found that many interviewees encountered challenges related to the information in VCS and other related systems. These challenges were found in managing both committed and uncommitted changes.

*1) **Committed changes**:* Many projects have a large VCS history and are still actively under development. This large volume of information caused information overload: *"The density of the commits is too high. You have a lot of files to go through."* About 47% of our survey respondents considered this a challenge. Our interviewees reported the following consequences of information overload:

- hard to locate a particular commit among sometimes noisy commits, e.g., actual change among several merge commits
- difficult to keep up with code changes, emails and also in remembering things
- undesirable changes go unnoticed, e.g., developers don't read every single commit that gets made

Such challenges were also reported with diffs. Participants often reported that diffs are hard to read: it's hard to get any insights or intent of the change from a diff, and one has to go through too many diffs to really understand a change. There is also noise added that makes diffs hard to understand, e.g., white spaces, line-endings, etc.

Even though using diffs is the second most popular strategy among our survey respondents to understand a commit (Fig.3), 31% of them reported that understanding diffs is difficult.

Participant P14 reported that diffs are overwhelming especially with large commits: *"The amount of signal - noise ratio you have to deal with sometimes. If there is a single file you have to look at, it is usually easier."*

The issue becomes more prominent due to tangled changes: having large commits with more than one unrelated changes. 54% of our survey respondents also reported tangled changes as a problem.

Some ways that help developers currently overcome these challenges include:

- *Grouping related commits* in version control history, e.g., pull requests, merge commits

**Table II: a)** Top challenges survey respondents faced when examining history, and **b)** Top tool capabilities survey respondents desired the most

**(a)**

| | |
|---|---|
| Non informative commit messages | 66% |
| Tangled Changes | 54% |
| Information overload | 47% |
| Traceability to versions | 31% |
| Interpreting diffs | 31% |
| Traceability to requirements | 20% |
| Tool limitations | 20% |
| Traceability to architecture | 17% |
| Other | 17% |

**(b)**

| | |
|---|---|
| Traceability to versions | 58% |
| Informative commit messages | 48% |
| Aggregate commits into groups | 46% |
| Ability to filter changes | 40% |
| Support for managing uncommitted changes | 28% |
| Traceability to architecture | 26% |
| Selective change notifications | 20% |
| Other | 3% |

- *Filtering commits*, primarily used to reduce their search space, e.g., search for a word in commit messages
- *Structured history*, which includes small, frequent commits without tangled changes.

However, in spite of the mitigation strategies developers employ, they still face challenges in dealing with information mess. Our survey participants reported that they wish they had better filtering and grouping capabilities (Tab.IIb). Some of our findings from interviews and surveys include:

- *Better grouping* capabilities for commits, e.g., group related changes across branches and repositories
- *Better filtering* of commits, e.g., only show non-merge commits
- *Providing filtering capabilities for diffs*, e.g., filter white space changes
- *Querying*, e.g., when was a line added

*2)* ***Uncommitted changes****:* With uncommitted changes, i.e., changes not yet included for versioning, participants reported two major challenges:

- the changes were tangled and it was hard to separate changes to good fine grained commits
- the changes were unorganized and it was hard to navigate *"the jungle"* of changes (P2)

Currently there are not many tools that support this activity: 28% of our survey participants wished they had better support for managing uncommitted changes. We found that developers need some structure for viewing and grouping uncommitted changes. This can help them pick changes for small, fine grained commits and also in backtracking during their development activities. Participant P2 said *"If I had a list of all the code changes ... in a way that I could identify the good parts and throw away the bad ... it would have to be incredibly easy for me to find what are the good parts that I want to keep."*

**Observation 4:** Developers have challenges with information overload because history is cluttered and unstructured. Similar issues affect uncommitted changes as well.

### B. *Knowledge fragmentation*

As developers navigate and examine history to find information, they also ask high level questions regarding requirements [26], architecture, related changes, etc. They need to understand not just the change but also the context. Interviewees reported challenges in retrieving high level information from history.

*1)* ***Traceability to versions****:* Participants wanted to see the entire history of a code snippet of a file or a document.

Code snippet history is lost and fragmented when files get moved, renamed or code gets moved within a file. Participant P4 mentioned *"this came into being here [version 721] because it was moved from somewhere else...and the original line came into being at version 507."* Even though Git has options like "git mv" or "git log –follow" to handle file moves, these are not popular nor the defaults and the UI clients for VCS tools often do not support any of these.

They also reported that history got lost when VCS repositories got truncated during updates or files were moved between repositories. P11 mentioned *"I had to go find backups of files to find the older versions of stuff."*

Some suggestions for tools from our interviews are:

- Look at a code snippet change in the context of the entire change it was a part of, *"... I look at the rest of the changes for that particular commit. Just to see the context of why that particular line was changed"*(P6)
- History should be grouped and overlapped on code: *"instead of being, like 'this line is red and this line is green because it got deleted' there would be special colors like 'this one came first, this one came second ...' so you can say 'ok, these changes all came together and these changes all came together' "*(P9)

*2)* ***Traceability to requirements****:* Our participants reported the need to know the requirements that drove a change. However, some of our interviewees and 20% of our survey participants reported that it was hard. Some also reported that it was not trivial to find all changes related to a requirement. P2 said *"finding all the bug fixes and applying them to the other one [specific client] was tedious and horrendous."*

We found that some participants mitigate this by referencing issue tracking system IDs in commit messages and using knowledge aggregators (e.g., Atlassian Stash) that connect code changes to requirements.Without such techniques, it was tedious and difficult to use software history. For example, P2 reported *"In retrospect if we would have had a good bug tracking system, and if we could have tied the bug reports to specific code patches, and then if we could cherry pick those code patches and apply them to the other code base, then that would have made the process trivial."*

*3)* ***Traceability to architecture****:* Our participants reported that it is difficult to tell which components (module, subsystem, etc.) are affected by a commit. While 30% of our survey participants analyze the impact of change in recent code (Fig. 1), 17% reported it as a challenge (Tab. IIa) and 26% wished this was easier(Tab. IIb). This especially seemed useful when one uses history for managing deployments, planning regression testing, etc.

For example, P10 wished a UML-like diagram with modified components highlighted: *"I can see a model [UML model]...I could get a picture of my architecture, and it could show me 'here are the red components that have changed' "*. D'ambros et al. [27] present a prototype for projecting changes on high level software visualizations.

Participants stated that good software modularity [28] enabled assessing the architectural impact of a commit. This in turn helps with understanding commits and selectively composing past changes. P8 said that: *"The project is very modular. A change does not affect many things. So that helps."*

**Observation 5:** Developers face issues in finding the history of a code snippet, tracing that history to requirements and assessing the architectural impact of changes.

### C. Understanding history

*1)* ***Hard to understand****:* Participants also reported that old history is hard to understand. P2 reports that: *"We lost context on the original code. Going back and understanding what you had done 6 months ago was challenging."* This is because there is a lot of context that is not obvious or documented, buried only in old history. However, participants also reported that a good knowledge of code made it easier to understand history, e.g., P8 said *"I know the system very well, know the hierarchy, the control flow. It takes a quick visual inspection of what was modified to know if it is related."*

*2)* ***Commit messages****:* Commit messages are the most popular strategy to understand a commit among our survey respondents (Fig. 3). Interestingly, over 66% of them reported that commit messages are non-informative (Tab. IIa) and 48% wished commit messages were more informative (Tab. IIb). One participant said *"some are very helpful, some useless."* We also found that developers have different preferences for the level of detail they need in commit messages.

A good commit message expresses the rationale of the change and provides a link to requirements. Participant P9 said *"Since our commit messages are so explicit, it has been really nice to take a look at a line in the history where we can't figure out why it's there, check when it was committed, check the commit message and see what the intent was."*

*3)* ***Software engineering practices****:* Interviewees noted that good software engineering practices such as continuous integration, frequently syncing with VCS, unit tests and code reviews facilitated working effectively with history.

Participants also stated that the challenges in examining history were also due to a lack of rigor in practices:

- not following commit message guidelines
- avoiding code reviews
- not following some branching strategies
- missing tests
- no use of issue/bug tracking systems
- committing breaking changes

**Observation 6:** Developers encounter challenges even with the most popular strategies for understanding commits: commit messages and diffs.

### D. Tool limitations

Interviewees explicitly reported limitations with VCS tools and suggested several improvements.

*1)* ***Visualizing history****:* Participants reported challenges when comparing multiple parts of history at once and wished they were able to better compare branches or commits. Moreover, they wish they were able to track the movement of a commit between branches or to better visualize the history of a file, e.g., P10 wished *".. a way for me to sort of move the file back through time, just right there."*

*2)* ***Organization of history****:* Participants faced challenges with the linear organization of history, some participants mentioning challenges with not knowing where to start. P5 remarked *"wrong assumptions lead to wrong searches."* They wished there was a hierarchy to software history. For example, they wanted to view Github pull requests inside history, or the ability to group history by features or change similarity.

*3)* ***Usability****:* Participants also expressed challenges with the usability of existing tools. For example, they expressed having a hard time setting up binary search environments (e.g., for Git bisect), lack of support for detecting file moves and renames and with selectively picking commits from one branch to another. They also expressed challenges due the fact that VCS tools have heterogenous vocabularies, making it hard to map functionality across tools.

*4)* ***Change notifications****:* Both interview participants and survey respondents expressed a wish for receiving selective change notifications. Figure 2 shows that survey respondents were only interested in keeping up with some specific changes rather than all changes on the project while Table IIb shows that 20% of survey respondents have a need to be notified of some changes to the code repository. P8 wished that he received automated notifications when a change affected either his work in progress or a specific code entity: *"Get a text message whenever a change affects me. Have a predetermined set of files and the VCS tell if any of these change."*

## VI. 3-LENS HISTORY

Section III shows that developers have different needs for history based on the age of the changes. We hence propose a 3-LENS HISTORY model for version control systems. Each lens is unique in its usages and nature of information. A developer can use these lenses to focus on any part of history to accomplish specific goals.

**The IMMEDIATE lens**. Our results suggest that software history does not begin with commits, but with uncommitted changes. Even though in Section III we classified uncommitted changes as recent, they deserve special treatment. Developers backtrack and do trial and error programming, hence the changes at this level are more fine grained than a commit. They also group related uncommitted changes when splitting them in commits.

**The AWARENESS lens**. Developers use recent history to keep track of what changes are happening, to understand their own work in progress and to integrate their work with other people (Sec. III-A). Active collaboration with the others in the team happens in this zone and changes are being integrated with others'. The strategies developers use with recent history are described in Section IV-B.

**The ARCHAEOLOGY lens**. Old history is primarily used to retrieve lost knowledge (Section III-B). This part of history is passive and often only serves as a historical reference. With old history, developers understand the rationale behind a change, reverse engineer requirements or understand the project evolution. Often, this involves gathering more information like requirements, related changes, etc (Section V-B).

## VII. IMPLICATIONS

As the previous section shows, developers use each history lens for different activities. Therefore, we postulate that devel-

opment tools should make each lens a first class citizen and provide specialized support for each one.

**The IMMEDIATE lens**. Participants had challenges with managing uncommitted changes due to tangled changes and information overload (Sec. V-A). Table IIb shows that 28% of survey respondents wish they had better support for managing uncommitted changes such as grouping related changes and choosing which groups to commit.

We conjecture that keeping track of *uncommitted changes* could be the tool's responsibility: they could automatically group related changes and allow the developer to further refine the groupings. Tools could also aid the developer in transforming uncommitted changes into commits by providing a bread-crumb trail of transient commits that the developer could later toggle into real commits.

**The AWARENESS lens**. Table I shows that there is no single dominant change awareness source. It may be that no single source is good enough by itself to keep developers up to date with recent changes. Interviewees had a hard time keeping up with all the changes due to information overload (Sec. V-A) and 20% of survey respondents wish they could receive notifications when a piece of code of interest is changed (Tab. IIb). We foresee two venues for recent history tool support.

First, tools could aid developers in keeping track of the latest changes. They could give personalized updates based on the developer's current task and focus in the code, and allow him to resolve conflicts before conflicts get committed. Survey participants preferred to be notified only of changes that affect their current tasks or a specific code region, or fault introducing changes (Fig. 2). Second, tools could help the developer maintain a mental model of the changes for his work in progress. Current VCS mix the developer's commits with those of others thus obfuscating his own progress.

**The ARCHAEOLOGY lens**. 58% of survey respondents considered that understanding the evolution of a code snippet was a top reason for referring to old history while interviewees wished they could be able to understand how a previous version of the code snippet integrated with the surrounding code (Section V-B). However, participants face challenges when trying to do so: the code snippet moves inside files, between files or even between repositories. Moreover, participants faced challenges in recovering requirement or architectural intents.

This suggests that IDEs should be able to show all the past changes that a snippet of code went through and the changes should be placed in context of: (i) the rationale of the broader change it is part of and (ii) the surrounding code at that moment in time.

**Visually representing history**. The work done in this study represents the domain characterization phase in Munzner's nested model [29] for visualization design. Therefore tool builders who want to design visualizations for software history can use this study to learn about the domain of software developers' needs for history.

Interviewees mentioned that old history is hard to understand (Section V-C). We hypothesize this is due to the information overload caused by current VCS that represent history as a reverse chronological list of commits and to the fact that all the commits from all developers, tasks, and requirements, are amalgamated together.

On the other hand, developers need recent software history to understand how the latest commits affect their work in progress and how they affect the program logic.

We therefore conjecture that software history should have different representations, based on the age of commits. Recent software changes could be represented in detail, showing granular changes in program logic. Old history could be represented in an abstracted way: instead of a list of low level commits, old history could be narrated as a list of high level changes such as features, bugfixes, etc. Our previous work [30] that found differences between the usage of centralized and distributed VCS further supports this conjecture.

## VIII. RELATED WORK

We group the related work in two parts: (i) prior studies on software history and (ii) research tools that enable developers when accessing software history.

### A. *Prior studies on software history*

While previous work analyzed isolated needs for software history, we take a holistic approach. In this section we position prior studies under our unified 3-LENS HISTORY model.

Yoon et al. [25] studied developers' backtracking strategies and found several actions that developers have to perform with uncommitted changes, such as selectively undoing changes. Thus, we place their research in the IMMEDIATE lens. We found that developers accumulate large quantities of uncommitted changes. This leads to the concern of picking which changes to commit and which changes to abandon.

De Souza et al. [31] explore two change awareness activities: in forward impact management developers notify their peers about changes while in backward impact management developers seek to find the changes that affect them. Binkley et al [32] studied the effectiveness of automatic summarization of commits. We place their research in the AWARENESS lens.

Tao et al. [17] explored how developers understand code changes (i.e. understand a commit). They found that understanding changes is a frequent but difficult activity. In our study, understanding a commit is catalogued as a strategy used by developers in the AWARENESS and ARCHAEOLOGY lens.

Previous work in end user programming [33], [34] explored how end users used Yahoo! Pipes versioning. They found that end users use history to learn about an artifact's logic which is similar to the ARCHAEOLOGY lens or to revert changes which is similar to the IMMEDIATE lens.

### B. *Software history research tools*

While researchers have been partially addressing needs for each lens in the 3-LENS HISTORY model, we are the first to propose a unified model for examining history. In this section we position each tool under one of our lenses.

**The IMMEDIATE lens**. Azurite [35] is a visualization for uncommitted changes that supports selective undo. Our

study suggests that IMMEDIATE lens tools should also mitigate information overload by providing support for grouping uncommitted changes and transforming them into commits. Recent work [36]–[38] studies how to group related changes.

**The AWARENESS lens.** ARENA [39] automatically summarizes commits into release notes. Several research tools [40]–[44], display changes and detect conflicts as they happen while Mylyn [45] helps developers understand their work in progress. We found that developers would benefit from selective change notifications to avoid information overload.

**The ARCHAEOLOGY lens.** Chronos [46] and LHDiff [47] provide the entire history for a code snippet while DeepIntellisense [48] and Rationalizer [49] extract past requirement changes for a commit. Coxr [50] also mitigates some of the knowledge fragmentation challenges. From our study we conjecture that developers would benefit if these two functionalities were combined. In addition, our participants also wished that the context of the entire change along with the surrounding code at that point in time was available for each past version of a code snippet.

Several tools provide various kinds of visualizations for the evolution of source code [51], [52]. LSDiff [53] provides a high level narration of a commit together with possible inconsistencies. These approaches may aid activities in both the AWARENESS and ARCHAEOLOGY lens.

### C. Other research on software history

Researchers have mined software repositories to understand project evolution [54]–[59]. Other uses of history in mining repositories are listed in the seminal survey by Kagdi et al. [60]. Additionally, researchers are proposing new ways of representing software history, such as making software change a first class citizen and recording history at a fine-grained detail [61]–[63]. In this study we focus on developers and explore their usage of history.

## IX. Threats to validity

**Construct**. *Are we asking the right questions?* We performed an extensive literature review [5]–[14] to learn the gaps in understanding how developers access history. From this review we created the research questions in Section I.

We defined the threshold between old and recent history based on the Agile Manifesto [20]. A different threshold might lead to more precise results: the age independent motivations (Sec. III-C) might have an affinity for either old or recent history, but that affinity may not have appeared in the survey because of the time threshold we used.

**Internal**. *Is there something inherent to how we collect and analyze information that could skew the accuracy of our results?* Interviews and surveys risk biases and inaccurate responses. To mitigate these issues, we used guidelines in literature for designing and deploying our studies. [19], [31], [64]. For example, we ran pilots to iteratively evaluate and improve both studies.

Interviewees described strategies for examining history. These are conscious strategies they remember and they might be doing other things as well. Further research is needed with a complementary methodology (direct observation) to confirm and further clarify the findings of this study.

Interview and survey participants used different VCS tools. While we expect motivations, strategies and challenges to be largely tool independent, some mitigation techniques are VCS specific or the participants were unaware of existing solutions.

**External**. *Are our results generalizable for the general software history usage practice?* The responses that our interviewees gave might not be representative of the entire developer population. We mitigated this issue by interviewing at 11 companies working in different domains (e.g., graphics, entertainment, navigation, devops tools, etc.). We then validated the interview results with a survey taken by 217 respondents. Since the survey was advertised online, it may suffer from non-response and self-selection bias.

**Reliability**. *Can others replicate our results?* The interview script, survey template and raw data can be found on our website [21] so that others can replicate our research.

## X. Conclusions

Examining software history is an important activity in software development. Developers frequently refer to history to gain knowledge and make choices while writing code. Up to 61% of our survey respondents need to examine history up to a few times a day.

However, this is an under evaluated topic in research. In this study we shed light onto how developers examine history. We found distinct motivations for which developers examine history. We found that they use different strategies to examine different parts of history for different intents.

Unfortunately, we also found that software history tools are ill suited and do not provide explicit support for the needs that developers have from history. Furthermore, we found that developers have history related needs from uncommitted changes but today's tools largely ignore this fact.

We defined 3-LENS HISTORY, a unified software history model that provides a foundation for future work to give identity to history. Tool builders and researchers can use this model to make each lens a first class citizen in software development. They can focus and improve either each lens individually or the transition and interaction between lenses.

We hope that this study inspires others to improve how developers examine history, via education and tool support.

## XI. Acknowledgement

We would like to thank Margaret Burnett, Chris Scaffidi, Ron Metoyer, Carlos Jensen, Michael Hilton, Kendall Bailey, Shane McKee, Nicholas Nelson, Sean McDonald, Hugh McDonald, Sandeep Kuttal, David Piorkowski, Will Jernigan, Charles Hill, and the anonymous reviewers for feedback on earlier drafts of this paper. We would like to thank Joel Spolsky and Robert Martin for helping us promote the survey.

This research is partly funded through NSF CCF-1439957 and CCF-1442157 grants and a SEIF award from Microsoft.

## REFERENCES


[1] T. Barik, K. Lubick, and E. Murphy-Hill, "Commit bubbles," in *ICSE*, 2015.

[2] S. Perez De Rosso and D. Jackson, "What's wrong with git?: a conceptual design analysis," in *Onward!*, 2013.

[3] C. Parnin and S. Rugaber, "Programmer information needs after memory failure," in *ICPC*, 2012.

[4] M. S. Ackerman, "The intellectual challenge of cscw: the gap between social requirements and technical feasibility," *HCI*, 2000.

[5] R. P. Buse and T. Zimmermann, "Information needs for software development analytics," in *ICSE*, 2012.

[6] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *PLATEAU*, 2010.

[7] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *ICSE*, 2007.

[8] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *ICSE*, 2006.

[9] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *ICSE*, 2010.

[10] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *TSE*, 2008.

[11] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "A quantitative analysis of developer information needs in software ecosystems," in *ECSAW*, 2014.

[12] H. Li, Z. Xing, X. Peng, and W. Zhao, "What help do developers seek, when and how?" in *WCRE*, 2013.

[13] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *FSE*, ser. FSE 2014, 2014.

[14] A. Guzzi, A. Bacchelli, Y. Riche, and A. van Deursen, "Supporting developers' coordination in the ide," in *CSCW*, 2015.

[15] M. Biazzini, M. Monperrus, and B. Baudry, "On analyzing the topology of commit histories in decentralized version control systems," in *ICSME*, 2014.

[16] K. Muşlu, C. Bird, N. Nagappan, and J. Czerwonka, "Transition from centralized to decentralized version control systems: a case study on reasons, barriers, and outcomes," in *ICSE*, 2014.

[17] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *FSE*, 2012.

[18] S. Phillips, T. Zimmermann, and C. Bird, "Understanding and improving software build teams," in *ICSE*, 2014.

[19] F. Shull, J. Singer, and D. I. Sjøberg, *Guide to advanced empirical software engineering*. Springer, 2008.

[20] "Agile manifesto," http://agilemanifesto.org/principles.html.

[21] "Study website companion," http://cope.eecs.oregonstate.edu/HistoryStudy/.

[22] J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2012.

[23] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews problems of unitization and intercoder reliability and agreement," *Sociological Methods & Research*, 2013.

[24] S. Adolph, W. Hall, and P. Kruchten, "Using grounded theory to study the experience of software development," *ESE*, 2011.

[25] Y. Yoon and B. A. Myers, "An exploratory study of backtracking strategies used by developers," in *CHASE*, 2012.

[26] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *TSE*, 2002.

[27] M. D'Ambros, M. Lanza, and R. Robbes, "Commit 2.0," in *Web2SE*, 2010.

[28] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[29] T. Munzner, "A nested model for visualization design and validation," *TVCG*, 2009.

[30] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *ICSE*, 2014.

[31] C. R. de Souza and D. F. Redmiles, "An empirical study of software developers' management of dependencies and changes," in *ICSE*.

[32] D. Binkley, D. Lawrie, E. Hill, J. Burge, I. Harris, R. Hebig, O. Keszocze, K. Reed, and J. Slankas, "Task-driven software summarization," in *ICSM*, 2013.

[33] S. K. Kuttal, A. Sarma, and G. Rothermel, "On the benefits of providing versioning support for end users: an empirical study," *TOCHI*, 2014.

[34] K. T. Stolee, S. Elbaum, and A. Sarma, "Discovering how end-user programmers and their communities use public repositories: A study on yahoo! pipes," *IST*.

[35] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of fine-grained code change history," in *VL/HCC*, 2013.

[36] K. Herzig and A. Zeller, "The impact of tangled code changes," in *MSR*, 2013.

[37] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse, "Untangling fine-grained code changes," in *SANER*, 2015.

[38] M. Barnett, C. Bird, J. Brunet, and S. Lahiri, "Helping developers help themselves: Automatic decomposition of code review changesets," in *ICSE*, 2015.

[39] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *FSE*, 2014.

[40] A. Sarma, D. F. Redmiles, and A. Van Der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *TSE*, 2012.

[41] F. Servant, J. A. Jones, and A. Van Der Hoek, "Casi: preventing indirect conflicts through a live visualization," in *CHASE*, 2010.

[42] L. Hattori and M. Lanza, "Syde: a tool for collaborative software development," in *ICSE*, 2010.

[43] I. A. Da Silva, P. H. Chen, C. Van der Westhuizen, R. M. Ripley, and A. Van Der Hoek, "Lighthouse: coordination through emerging design," in *OOPSLA*, 2006.

[44] C. Treude and M. Storey, "Awareness 2.0: staying aware of projects, developers and tasks using dashboards and feeds," in *ICSE*, 2010.

[45] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *AOSD*, 2005.

[46] F. Servant and J. A. Jones, "History slicing: assisting code-evolution tasks," in *FSE*, 2012.

[47] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta, "Lhdiff: a language-independent hybrid approach for tracking source code lines," in *ICSM*, 2013.

[48] R. Holmes and A. Begel, "Deep intellisense: a tool for rehydrating evaporated information," in *MSR*, 2008.

[49] A. W. Bradley and G. C. Murphy, "Supporting software history exploration," in *MSR*, 2011.

[50] M. Matsushita, K. Sasaki, and K. Inoue, "Coxr: Open source development history search system," in *APSEC*, 2005.

[51] Q. Tu and M. W. Godfrey, "An integrated approach for studying architectural evolution," in *ICPC*, 2002.

[52] M. Burch, S. Diehl, and P. Weißgerber, "Visual data mining in software archives," in *SOFTVIS*, 2005.

[53] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE*, 2009.

[54] T. Girba, S. Ducasse, and M. Lanza, "Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes," in *ICSM*, 2004.

[55] G. Scanniello, "Source code survival with the kaplan meier," in *ICSM*, 2011.

[56] A. Serebrenik, W. Poncin, and M. van den Brand, "Process mining software repositories: Do developers work as expected?." *ERCIM News*, 2012.

[57] E. Tempero, H. Y. Yang, and J. Noble, "What programmers do with inheritance in java," in *ECOOP*, 2013.

[58] A. Gonzalez-Torres, R. Theron, F. J. Garcia-Penalvo, M. Wermelinger, and Y. Yu, "Maleku: An evolutionary visual software analysis tool for providing insights into software evolution," in *ICSM*, 2011.

[59] S. Wang, D. Lo, and X. Jiang, "Understanding widespread changes: A taxonomic study," in *CSMR*, 2013.

[60] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *JSME*, 2007.

[61] T. Gîrba and S. Ducasse, "Modeling history to analyze software evolution," *SME*, 2006.

[62] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *ECOOP*, 2013.

[63] R. Robbes and M. Lanza, "A change-based approach to software evolution," *Electronic Notes in Theoretical Computer Science*, 2007.

[64] I. Seidman, *Interviewing as qualitative research: A guide for researchers in education and the social sciences*. Teachers college press, 2013.