Developers' Need for the Rationale of Code Commits: An In-breadth and In-depth Study*

Khadijah Al Safwan[a,*], Mohammed Elarnaoty[a,c] and Francisco Servant[b,a,*,1]

[a]Department of Computer Science, Virginia Tech, United States of America
[b]Departamento de Teoría de la Señal y las Comunicaciones y Sistemas Telemáticos y Computación, Universidad Rey Juan Carlos, Madrid, Spain
[c]Department of computer science, Faculty of computers and artificial intelligence, Cairo university, Giza, Egypt

ARTICLE INFO

Keywords:
Software Changes Rationale
Software Evolution and Maintenance

ABSTRACT

Communicating the rationale behind decisions is essential for the success of software engineering projects. In particular, understanding the rationale of code commits is an important and often difficult task. Although the software engineering community recognizes rationale need and importance, there is a lack of in-depth study of rationale for commits. To bridge this gap, we apply a mixed-methods approach, interviewing software developers and distributing two surveys, to study their perspective of rationale for code commits. We found that software developers need to investigate code commits to understand their rationale when working on diverse tasks. We also found that developers decompose the rationale of code commits into 15 components, each is differently needed, found, and recorded. Furthermore, we explored software developers' experiences with rationale need, finding, and recording. We discovered factors leading software developers to give up their search for rationale of code commits. Our findings provide a better understanding of the need for rationale of code commits. In light of our findings, we discuss and present our vision about rationale of code commits practitioners' documentation, tools support, and documentation automation. In addition, we discuss the benefits of analyses that could arise from good documentation of rationale for code commits.

1. Introduction

The rationale of code commits is informally defined as the answer to the question: *"why was this code implemented this way?"* [18, 55]. However, software developers could easily interpret this informal question in many different ways, potentially as disparate as: *"what is the purpose of this code?"* [49]; *"why where [these changes] introduced?"* [27]; or *"why was it done this way?"* [55] — all of which request different answers. In past studies, software developers mentioned all these different interpretations when asked about rationale. Thus, we formed our intuition that it could be decomposed into multiple components, each addressing different aspects of the question.

In the scope of code changes, *i.e., code commits*, rationale is a major information need. Many research studies convey the importance of understanding the rationale of code commits. Rationale is the most common [18] and important [79] information need to understand from code-change history, and it is very frequently sought in developer tasks [25, 59, 62, 26]. Unfortunately, it can also be quite difficult to find an answer for it [79, 55]. We posit that a fundamental step to supporting developers in the difficult and important task of managing rationale of code commits is to understand their need for rationale of code changes.

Efforts to study rationale in-depth have been carried out in the context of design, decomposing it into various more-specific components [77]. In the context of software maintenance, *Burge et al. prescriptively* propose some questions that may answer rationale [16]. We, instead, take a *descriptive* approach, *i.e.,* we aim to discover *how developers decompose* the rationale of code changes — as opposed to conceptually and rigorously decomposing the concept.

With our developer-centric approach, our goal is to understand *what developers mean* by the rationale of code changes, and their *need* for it in practice. First, we study the context in which software developers need the rationale for code changes. We set out to discover the *tasks* (RQ1) and the *target code* (RQ2) for which developers need rationale of code changes. Second, we study the specific *pieces of information* (RQ3) in which developers decompose the rationale of code changes. Third, we study software developers' *experiences* (RQ4-7) when seeking rationale for code changes. To understand these aspects of the rationale of code changes in depth, we used developer interviews and surveys. Other methods (*e.g.,* mining software repositories) could be applied to study different aspects (*e.g.,* the extent to which code commits fulfill the need for their rationale), but our goal was to understand the developers' need in depth.

We used a mixed-methods approach in our study, involving interviews and surveys of software developers. This strategy allowed us to *qualitatively* study in detail various aspects about the rationale of code commits (through rich one-on-one conversations). Additionally, this strategy allowed us to reach a larger number of participants once we defined a more specific set of *quantitative* questions.

We found that software developers need the rationale for code commits for various software development tasks (and subtasks): *programming*, *working on bugs*, *communication*, *tools*, *documentation*, *project management*, *testing*,

*This paper is an extension of a conference research paper [66].
*Corresponding author
**Principal corresponding author
khsaf@vt.edu (K. Al Safwan); marnaoty@vt.edu (M. Elarnaoty); francisco.servant@urjc.es (F. Servant)
ORCID(s):
[1]Some work performed while at Virginia Tech.

and *specifications*.. Because our information-needs study is information-based (*i.e.,* rationale), we discovered a much more exhaustive list of tasks for which rationale for code commits is needed than task-based (*e.g.,* code review) information-needs studies. To the best of our knowledge, the need for rationale for code commits while working on the subtasks *postmortem bug analysis* and *deployment* was not known before our study. With our discovered tasks, efforts to support rationale for code commits need can now be targeted to improve rationale documentation in the proper context.

We discovered that software developers decompose the rationale of code commits into 15 separate components that they could seek when searching for rationale: *goal*, *need*, *benefits*, *constraints*, *alternatives*, *selected alternative*, *dependencies*, *committer*, *time*, *location*, *modifications*, *explanation of modifications*, *validation*, *maturity stage*, and *side effects*. Some of these reported components, *e.g., committer* and *time*, were not previously mentioned in studies of rationale in other contexts, *e.g., Tang et al.* [77], and were instead specific to the context of software maintenance. Understanding which components developers seek in rationale is an important problem, since most developers reported seeking it multiple times a week or more often, and spending more than 20 minutes on finding it in hard cases.

Furthermore, we highlight software developers' experiences with rationale. We present *human*, *team*, and *project* factors leading software developers to give up their search for rationale. For example, *changes in project personnel* make it hard for software developers to find the rationale behind code changes. We also present components of rationale that are needed but not frequently found or rarely recorded. Developers most struggled to find *side effects* and *alternatives*, and they need to find them on average multiple times per month and per year, respectively. Additionally, the developers least often record: *alternatives*, *selected alternative*, *constraints*, and *maturity stage*, even if they need to find them on average multiple times per year (*alternatives*) and per month (remaining ones). These findings of developers' experiences revealed areas of improvement in developers' practices regarding the rationale of code commits.

Our findings have multiple *implications for practitioners*. Our decomposition of the rationale of code commits provides: (1) a *common language* to use when discussing it, which practitioners can use to (2) assess and (3) strengthen the quality of their rationale sharing and documentation processes. While we do not expect practitioners to document all components in all situations, they now have an extensive list of components to judge which ones are relevant for each situation. Our findings also facilitate multiple *lines of research*. Given the decomposition of rationale, the set of tasks for which rationale is needed, and developers' experiences with rationale efforts could now be targeted to: (1) better document, (2) develop support tools, and (3) automate documenting rationale for code changes.

This paper extends our previously published paper [66], adding three new research questions (RQ1, RQ2, and RQ6) for which we collected new data and analyzed data that was not analyzed in the conference paper. This paper provides more breadth and depth. We cover more breadth by exploring the context (RQ1 and RQ2) for which the rationale of code commits is needed. For more depth, we extend our study of developers' experience with rationale of code commits by investigating the factors (RQ6) that make them give up their search for it. The remaining research questions in this paper remain the same as in the conference paper. They analyze the same data and provide the same results. They correspond as follows: RQ3 corresponds to previous RQ2, RQ4 corresponds to previous RQ1, RQ5 corresponds to previous RQ3, and RQ7 corresponds to previous RQ4. We also illustrate this in Figure 1: we represent the new research questions in blue and the old ones in grey.

2. Related Work

Existing work highlights the importance of rationale management throughout the software development life-cycle [23, 16]. Thus, multiple approaches and systems have been proposed to integrate rationale management in the process of software requirements engineering, software design, and software architecture [54, 30].

We focus on the rationale for code commits in the context of software evolution and maintenance. In this context, existing work studied part of the experience of developers seeking the rationale of code changes, finding that it was considered important and sometimes hard to find [79]. In contrast, this manuscript presents the first in-depth study of the **decomposition** of the rationale of code changes into multiple components, and the **experience** of developers seeking **it and its individual components**. We also identified an extensive list of the **tasks** in which developers need to learn the rationale of code commits.

In the following sections, we discuss the related work in several areas:

2.1. Rationale Management in Software Requirements, Design, and Architecture

Multiple extensions of requirements models were proposed to encourage the capture of rationale within them [7, 45]. In addition to these models, tools have been proposed to manage rationale of software requirements [54, 39, 6].

Many schemes have also been proposed to capture design and architecture rationale. The schemes can be divided into two categories: decision-centric *e.g., Lee and Lai* [56] and usage-centric approaches *e.g., Burge et al.* [16]. The decision-centric approaches [61, 77] focus on capturing the rationale as a decision-making process utilizing Toulmin's model of argumentation [80] and Rittel's Issue-Based Information System (IBIS) [51]. The usage-centric approaches focus on capturing rationale without representing the decision-making process [30, 81, 33, 77].

The usage-centric approaches "recognize that organizing rationale around decisions is not the best way to elicit and characterize some of the rationale needed for making appropriate design decisions" [16]. *Jarczyk et al.* provided a survey of the systems developed to support design rationale,

all of which were based on Toulmin's model or IBIS [40]. Design rationale has also been supported by multiple tools [34, 15], which can help detect inconsistencies, omissions, and conflicts [78]. Other more recent tools focus on capturing design rationale from software artifacts like IRC discussions [4, 3, 5], or user reviews [52]. Our study, in turn, focuses on the rationale in software maintenance.

2.2. Rationale in Software Evolution and Maintenance

This section discusses the related work in which *rationale of code changes* were prescriptively decomposed and discovered to be a major information need.

2.2.1. Components of the Rationale of Code Changes

Burge et al. prescriptively enumerate a few questions that may answer rationale in software maintenance [16]. We, in turn, provide a *descriptive* model of rationale in the context of software maintenance, from the perspective of what developers need to find when they seek it. Our descriptive approach extends *Burge et al.* 's prescriptive approach as we discovered more questions that may answer rationale in software maintenance. For example, our change objective questions (*What did you want to achieve?*, *Why did you need to achieve that?*, and *What is the benefit of what you want to achieve?*) were not mentioned in *Burge et al.* study. Our extended set of questions cover all of *Burge et al.* prescriptively enumerated questions. Because our set of questions is descriptive of the developers' needs while seeking and finding rationale for code changes, we believe these questions will help the finding and recording activities.

2.2.2. Tasks that need the Rationale of Code Changes

Past research focused on discovering various developer information needs within a given developer task (*e.g.,* collaborating [49], understanding code [57], understanding bug reports [14], understanding the life of bugs [8], or reviewing code [59]), and uncovered a breadth of aspects about them, such as how early [14] or frequently [57] each piece of information was needed. Our goal is instead to discover the tasks in which this particularly important information need (the rationale for code commits) exists (*i.e.,* our study takes the opposite direction), and we study various aspects about it. Our results in this theme validate the findings of some previous studies, since we found that the rationale of code changes was needed in *learning* [16], *code review* [59, 62, 26], and *mentoring* [20].

Tao et al. prescriptively proposed a list of seven "development scenarios" (*e.g.,* refactoring, developing new features, and fixing bugs) in which they expected developers to need to understand code changes. They asked developers to choose which ones they encountered most often, and found that the most often encountered one was "reviewing others' changes" [79]. Our work had a different goal: to *discover* the tasks for which developers need to understand the rationale of code changes, as stated by developers, *i.e.,* in a *descriptive* fashion. As a result, we discovered a much more exhaustive list: we observed that developers needed to understand the rationale of code changes in eight tasks and 25 sub-tasks (as opposed to *Tao et al.* 's seven scenarios).

2.2.3. Experience with the Rationale of Code Changes

Studies involving software history and developers' information needs in the last decade establish a strong demand for rationale [16, 18, 27, 79, 49, 55, 64, 59, 25, 57, 76, 62, 26, 20]. Our work is motivated by these empirical studies highlighting the importance of rationale for code commits.

The most closely related study to ours is *Tao et al.* 's [79]. *Tao et al.* found that the most important information need for understanding code changes is rationale, which is sometimes easy and sometimes difficult. We replicated these two questions (importance and difficulty) of their study. We also studied additional questions in three additional contexts: needing, finding, and recording rationale. Our study validates their results since our participants reported similar ratings for the importance and difficulty of finding the rationale of code changes. This similarity of results also shows that we studied a similar population of developers.

Our study extends *Tao et al.* 's by finding the individual pieces of information that compose rationale; the experiences of developers needing, finding, and recording those individual pieces; and recommendations to improve their documentation. In particular, this finer level of granularity (*i.e.,* rationale of code commits) enables us to provide a possible explanation for one of the main phenomena observed by *Tao et al.* : that rationale is easy to find when it is *well documented*. We posit that rationale is deemed well documented when it contains the specific components that the developer is seeking at that moment.

3. Research Questions

Our empirical study answers seven research questions on three themes.

3.1. Theme 1: Tasks with Rationale Need

We aim to understand some of the contexts under which the rationale of code commits is needed.

RQ1: What are the tasks in which software developers need to find the rationale for code commits? Discovering the larger set of *tasks* in which *rationale for code commits is needed* will enable researchers to tailor their support to the context of each task. While the rationale for code commits has been identified as a need in previous task-specific studies (*e.g.,* during code review [62]), an exhaustive list of of tasks in which developers needed it is still unknown.

Finding: Our study participants need to find the rationale for code commits while working on *diverse software development tasks* (Table 2 and Fig. 3A).

RQ2: How often do developers seek rationale of their team's internal code vs. others' code? Our intuition is that developers may seek the rationale for code commits of internal and external code outside their team. If our intuition is validated, future research efforts (e.g., rationale retrieval support) should accommodate for the fact that developers may or may not own the code for which they seek rationale.

Finding: Our study participants seek rationale of code commits for both internal and external code (Fig. 3B). However, we observed tasks where only internal/external code commits rationale is sought (Fig. 3A).

3.2. Theme 2: Components of Rationale

We aim to discover an extensive set of rationale for code commits components that developers believe would compose a high-quality detailed description.

RQ3: Which components do software developers decompose the rationale of code commits into? A *model of rationale for code commits* will inform developers wanting to improve their documentation of rationale of code commits — whether they aim to document it fully or more thoroughly.

Finding: Our study participants decomposed the rationale of code commits into *15 different components* (Table 3). We categorized the components into four themes; the change *objective*, *design*, *execution*, and *evaluation*.

3.3. Theme 3: Experience with Rationale

We aim to explore software developers' experience needing, finding, and recording the rationale of code commits.

RQ4: What is the experience of developers needing, finding, and recording the rationale of code commits? We investigated this research question to understand the effort developers dedicate to seek and document the rationale of code commits. *Tao et al.* discovered that finding the rationale of code commits is *very important*, and it is *easy* or *hard to find* depending on how well-documented it is [79]. We replicate those two questions of their study, and extend it by asking developers five additional questions in three different contexts: needing, finding, and recording rationale.

Finding: The majority of our study participants reported *needing* the rationale of code commits *relatively frequently*: multiple times per week or more often (Fig. 4A).

Finding: Our study participants reported that rationale of code commits is *found often or almost always*, with *neutral difficulty of finding*, and *relatively low average search time*. However, in the *hard search cases*, our participants could *spend more than 30 minutes* searching for the rationale of code commits (Fig. 4B).

Finding: Our participants reported *similar recording and finding frequencies* of rationale for code commits (Fig. 4C), which suggests that documentation efforts generally help others in finding rationale.

RQ5: What is the experience of developers needing, finding, and recording the individual components of the rationale of code commits? We studied how developers need, find, and record different components differently. This study will enable developers to improve their documentation of rationale efforts, *e.g.,* by focusing on documenting the most needed or most hard-to-find components.

Finding: Our study participants' most needed and important components of rationale for code commits are the change *modifications, goal, location, need, committer, and time*. (Fig. 5).

Finding: Our study participants' most hard-to-find components of rationale for code commits are the change *alternatives, side effects, constraints, dependency, selected alternative, benefits, and validation*. (Fig. 5).

RQ6: What makes software developers give up their search for rationale of code commits? We wanted to discover issues that may limit software developers from effectively finding rationale of code commits. Identifying these issues will inform future research efforts that are targeted to support information needs' finding

Finding: Our study participants give up their search of rationale for code changes due to diverse eight reasons (Table 4 and Fig. 6), which we categorize into *project-centric*, *human-centric*, and *team-centric* factors.

RQ7: Would comparing the experience of developers needing, finding, and recording the individual components of the rationale of code commits with each other reveal areas for improvement? We performed a cross-dimensional study (*i.e.,* comparing need vs. finding vs. recording components) to investigate areas for improvement in current recording and finding practices of rationale of code commits. Identifying gaps, *e.g.,* between needed and recorded components, will provide valuable recommendations for developers who want to improve their documentation of rationale for code commits.

Finding: Most rationale for code commits' components are not too frequently needed, but when they are needed they are really hard to find. Our analysis shows that our participants struggle most to find the *side effects* and *alternatives* components (Fig. 7C).

4. Research Method

Our study uses a mixed-methods approach, combining developer interviews and a survey. Mixed-methods have been successfully employed in other studies of software developers, *e.g.,* [18, 79, 38, 73]. The developer interviews allowed us to *qualitatively* discover details about multiple aspects of the rationale of code commits, through rich one-on-one conversations with developers. The surveys enabled us to reach more participants and extend our *quantitative* findings. Figure 1 outlines the method that we followed in this study. In the following subsections, we present the design of our interview, surveys, and participant recruitment.

4.1. Developer Interviews

We designed and refined our interview script through five pilot sessions. We ran a first pilot interview at the early stages of designing our interview script, in which we asked general open questions about the rationale of code changes. After the first pilot, we improved the interview

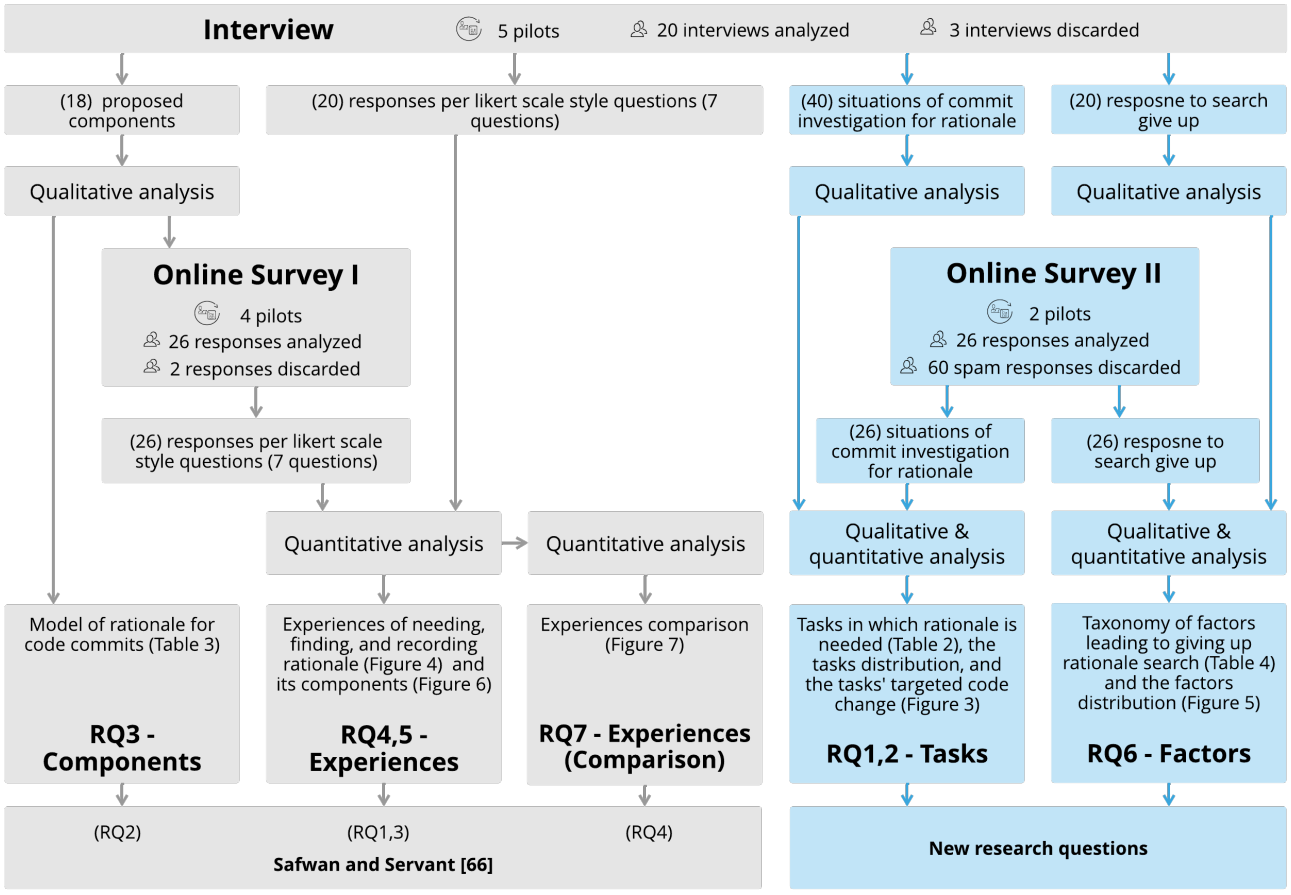


Figure 1: Research method summary

script, making it more structured, making the questions more specific, and adding a preliminary model of rationale (Table 1) to the script. Then, we ran a second pilot to test these improvements. After it, we changed some questions' wording to make them clearer. We ran the third and fourth pilots with experienced and beginner developers respectfully. Our goal was to test the new improvements and check the time required for our interview. After the fourth pilot, we finalized the interview kit (available in the research artifacts package) by adding introductions to the different sections of the interview script. Finally, we ran a fifth pilot to check for the entire interview process. The process included: advertisement, screening survey, scheduling, interview session, and analysis of responses. After this pilot, we were ready to recruit and interview participants.

Our interview consisted of three parts, one for each research theme. The first part focused on finding the tasks in which rationale of code commits is needed (RQ1). The second part focused on discovering the components that form the rationale of code commits (RQ3). The third part aimed to understand the experiences of developers needing, finding, and recording rationale of code commits (RQ4) and its components (RQ5), and to understand when developers give up searching it (RQ6). We study RQ7 by comparing participants' responses in different dimensions.

Theme 1: Tasks with Rationale Need. We started our interviews by giving our participants the definition of *rationale of code* that is most common in the research literature, *i.e., the answer to "why is the code this way?"* [18, 55]. We did this to ensure that all participants had a uniform definition of the concept we discussed. Next, we asked them to describe real situations in which they investigated a code commit to understand its rationale. This way, they grounded their answers in real experiences. We used their answers to identify the tasks (RQ1-2) in which they needed rationale. We provide more details about this analysis in Section 5.1.2.

Theme 2: Components of Rationale. Right after giving our participants the definition of rationale of code commits and asking them to describe real situations in which they needed it, we asked them to decompose the rationale of code commits into components. By asking this question after they had been thinking about their own experiences searching for the rationale of code commits, we intended to stimulate the participants' memories and set them in the right context, as well as to maximize the number of components that they would report. Then, we showed them a *preliminary model* (see Table 1) of components of the rationale of code commits that we created by studying the research literature — including components to which researchers have referred as *rationale* [16, 49, 27, 74, 55]. We used this preliminary model as a probe to prime our participants and get them in the right

frame of reference. We asked participants to critique and extend the preliminary model — considering their previous decomposition — to the extent that they believed necessary to build a *final model* of all the components of the rationale of code commits. For any component added by participants, we asked them to describe it with a name, question, and example answer. We presented the same preliminary model to all participants — *i.e.,* we did not show the modified models to other interview participants.

Using the preliminary model as a probe served multiple purposes. The preliminary model clarified the scope of our study. It also allowed developers to discuss an extensive set of components. We also believe that it allowed us to reach saturation of answers much faster (interviewing fewer participants) than if we had only relied on our participants' experiences and decompositions — since situations in which many components are needed simultaneously may be rare, or because people's memory is generally unreliable.

However, by using a preliminary model, we had the risk of introducing confirmation bias [60]. We took multiple measures to reduce this potential bias. First, we presented the preliminary model neutrally, as *"this model"* — avoiding potentially-biasing adjectives, such as "ours" or "preliminary". Second, we built it from the research literature, reducing the risk of inserting our own opinions. Third, we presented the preliminary model to participants only after they had produced their own decomposition, without having seen it. Fourth, we asked participants to consider their own decomposition when critiquing and extending the preliminary model. We believe that we were successful with these efforts since our final model of rationale of code commits (see Table 3) is much more extensive than the preliminary model (see Table 1). The preliminary model had only nine components, whereas the final model has 15.

Theme 3: Experience with Rationale. Next, we asked participants to rate their experiences needing, finding, and recording rationale of code commits (RQ4) and its components (RQ5) in Likert-scale-style questions. We also asked an open question about what makes developers give up their search for rationale (RQ6).

4.2. Survey I

Once we had identified the components of the rationale of code commits through our interviews, we used a survey to obtain more answers about developers' experiences needing, finding, and recording rationale of code commits and its components. We refined our survey through four pilot versions, improving its clarity and the time required to complete it. Our survey included the same Likert-scale-style questions that we asked our interview participants for RQ4 and RQ5, but the reference model of rationale of code commits that we gave survey respondents was the final model resulting from our analysis for RQ3. Our results for RQ4 and RQ5 include the answers that we obtained both from our interviews and the survey. We answer RQ7 by comparing the responses obtained from RQ4 and RQ5 (from both interviews and surveys) across multiple dimensions.

Table 1
Preliminary model of rationale of code commits

Component	Component Expressed as a Question	Literature References
Goal	What do you want to achieve?	[16, 49]
Need	Why do you need to achieve that?	[27, 74]
Location	What artifacts were changed?	[16]
Modifications	What specific changes were performed in the artifacts?	[74, 49]
Alternatives	What other alternatives did you have?	[55]
Selected alternative	Why did you make those specific changes and not others?	[49, 55]
Validation	How do those specific changes achieve the goal?	[55, 16]
Benefits	What is the benefit of what you want to achieve?	[49, 16]
Costs	What risks could come from these changes?	[49, 16]

4.3. Survey II

Once we had analyzed our interviews to identify the tasks for which rationale is needed and the factors leading our participants to give up their search of the rationale of code commits, we created a short survey to obtain more responses. We ran two pilots to test the new survey. The pilots provided us with positive feedback and an understanding of the time required to complete the survey. Our survey included an introduction and the same open questions we asked our interview participants for RQ1 and RQ6. We obtained results for RQ1 and RQ6 by analyzing the answers that we obtained both from our interviews and Survey II.

4.4. Participants' Recruitment

We used snowball sampling [12] to recruit participants for our study, *i.e.,* we asked them to refer our study to their contacts. We advertised our study in mailing lists in our university that covered software developers of diverse experience, *e.g.,* developing various university software systems, and graduate students with professional software development experience. We also advertised it through public channels and social media, *e.g.,* developers' communities on Slack. We compensated interview participants with a $20 Amazon gift card and encouraged surveys participation by raffling a $50 gift card. Figure 2 represents the demographics of our interview, survey I, and survey II participants.

We interviewed 20 participants, after having discarded three other interviews for various reasons: one participant could not describe an example of seeking rationale of code commits, another voluntarily expressed lack of experience throughout the interview, and we found out that the last one knew our interview materials.

We analyzed 26 survey I responses, after having discarded two responses. We discarded two surveys that we deemed as having been done carelessly, taking less than 10 minutes. We determined this cut-off point through our pilot

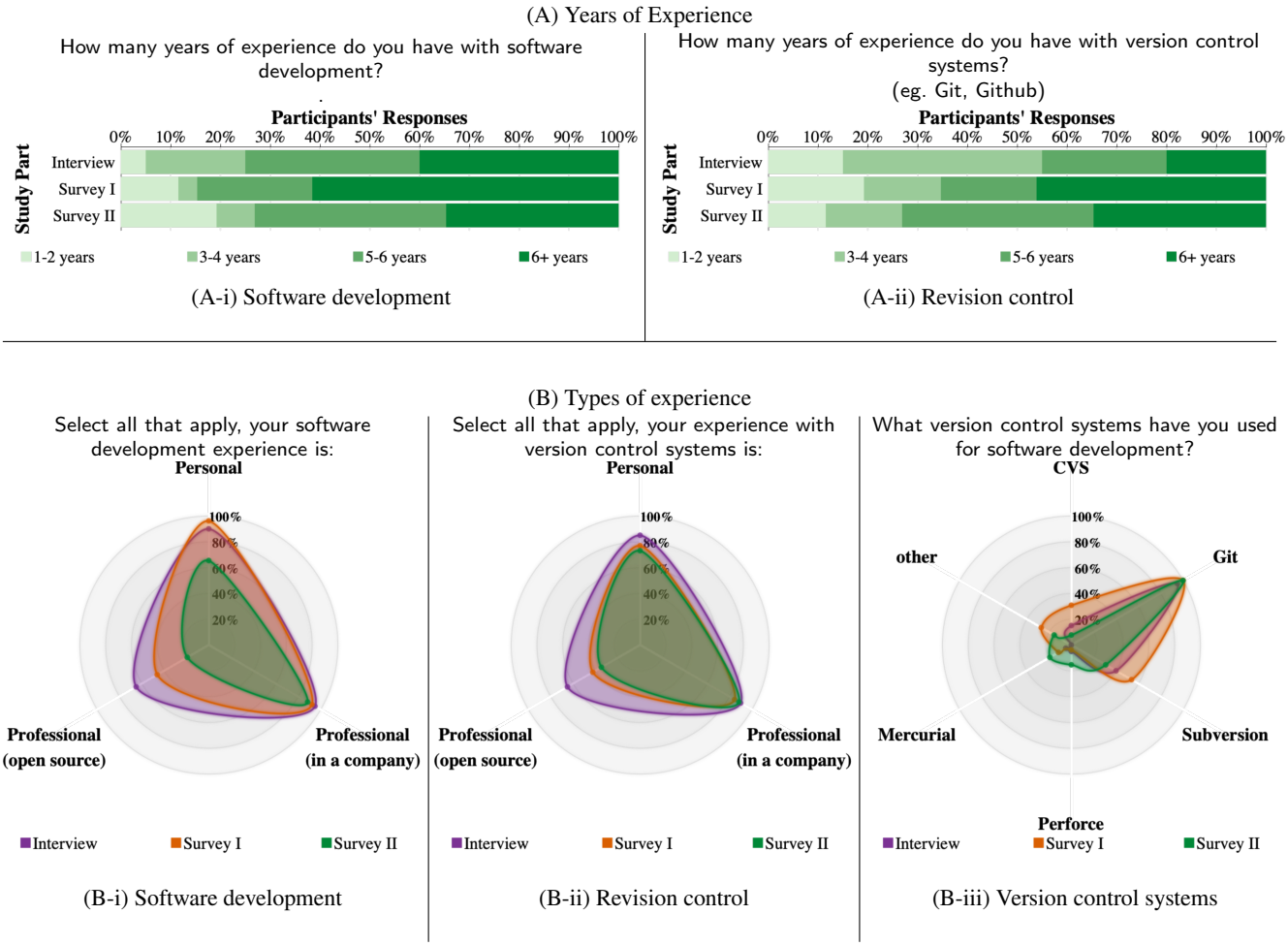


Figure 2: Demographics of our interview and survey participants

surveys; we asked one pilot participant to fill the survey carelessly and it took 10 minutes. We specifically asked the interview participants not to fill the survey, eliminating the chance of including duplicates in our results. For this survey, we did not have a target number of responses. We kept the survey open for a month, which resulted in 28 total responses (before any filtration).

We analyzed 26 survey II responses after having discarded 60 spam responses. We identified those spam responses since the spammers did not address the asked open questions. Instead, for each response, the spammers submitted the same spam text answer with different contact information. We believe a large number of spam responses resulted from the compensation's raffle odds of winning (1 in 50 chance). For this second survey, the advertisement emphasized: "not to fill the survey if you have participated in the study before". Our target was to reach a similar number to the previous survey I. We also kept the survey open for a month, which resulted in 86 total responses (before any filtration). Although we received/analyzed the same number of responses as survey I, the participants are different, which can be observed from the demographics figure (Figure 2).

5. Theme 1: Tasks with Rationale Need

5.1. RQ1: What are the tasks in which software developers need to find the rationale for code commits?

5.1.1. Research Method

We asked our (20) interview and (26) survey participants to describe a situation in which they needed to understand the rationale of a code commit. We analyzed their responses qualitatively, using closed coding to extract the task they were performing when they needed rationale of code changes. All the authors of the paper were involved in the coding. We reached saturation in our observed tasks in interview 6 (of 20), and in our observed subtasks in survey 14 (of 26).

Analysis data: To base our observations on real developer experiences, we asked our participants (with an open-ended question) to describe a real situation in which they needed to understand the rationale of code commits. Then, we analyzed their descriptions of these real-world experiences to identify the tasks they were performing, in which they experienced the need to know the rationale of code commits. We specifically asked our participants:

Table 2
Tasks for which rationale for code commits is needed

Task	Subtask	Description	Example (abstracted from participants' responses)
Programming	Reading	Reading source code to understand various aspects about it, like design and features.	A Developer navigates through the commits of a project to understand how a specific feature was implemented. The feature spans multiple classes, and the developer finds the commit that introduced the feature. The developer reads and investigates the rationale of the changes.
	Writing	Writing source code to implement, refactor, improve, and maintain the codebase.	A developer tries to break down a big function in a previous commit to improve code reuse and testing. This leads the developer to ask questions about the rationale of the changes.
	Proofreading	Reading source code to look for and solve issues before submitting the code for review or boarding.	A developer proofreads code to make sure that variable names are clear and informative before committing. This verification of variable names leads the developer to ask questions about the rationale of the changes.
	Code Reviewing	Reviewing source code as part of the code review process.	A reviewer assesses the correctness and quality of the commit under review, which requires understanding the rationale of the changes.
Working on Bugs	Reproducing	Reproducing the situation in which a previously reported bug was observed.	A developer tries to reproduce a race condition that happens occasionally. The developer investigates the edge cases introduced in a previous commit to guess when the race condition emerges, which leads the developer to ask questions about the rationale of the changes.
	Reporting	Reporting the existence of a bug through formal/informal communication methods to inform the team members.	A developer discovers a bug in the codebase. To write a bug report for it, the developer wants to refer to a particular commit as a suspect of introducing the bug. In the effort to find the suspect commit, the developer needs to understand the rationale of the investigated commits.
	Triaging	Evaluating a reported bug in terms of validity, severity, urgency, and needed work, before assigning a developer and a due fix time.	A Project Manager (PM) uses git blame to figure out who introduced code associated with a newly reported bug. In this effort, the PM needs to understand the rationale of the multiple code changes that introduced the buggy code.
	Debugging	Finding the source code that contains the bug and fixing it.	A developer investigates his/her assigned bug, returns to the previous commits, reads their code, reflects on their rationale, understands how the bug was introduced, and writes a fix.
	Postmortem Analysis *	Exploring and analyzing bugs, which might have been already resolved, for research or to improve productivity.	A PM studies the history of bugs to take preventive measures in the future. In this effort, the PM needs to understand how these bugs were introduced and the rationale behind the commits introducing them.
Communication	Learning	Learning software best practices, conventions, technologies, skills, and tools.	A researcher looks at the commits in the repository of an open-source ML library to learn the undocumented mathematics of the ML approach. This leads the developer to ask questions about the rationale of the changes in the commits..
	Coordinating	Coordinating with members of the same or another team to achieve a smooth integration.	Team A and team B are developing intersecting functionality. A developer in team A looks at the commits of team B to avoid redundancy, sees how team A can leverage parts of team B's code, and builds a common vision of the two teams. In this process, developer A requires understanding the rationale for some of the code changes by team B.
	Mentoring	Advising, guiding, and one-to-one teaching another software developer.	A mentor looks at the mentee's code commits to check how well the mentee practiced the design process, which leads the mentor to ask questions about the rationale of the mentee's changes.
Tools	Discovering	Discovering various aspects about a tool or library before adoption, such as its supported features, popularity, and version history.	A software team is assessing whether to replace an existing component with a new tool that is not fully documented yet. A developer goes through the commits in the repository of the new tool to discover its capabilities and to look for what the team needed. In this effort, the developer needs to understand the rationale of the historical changes to the tool.
	Installing	Installing tools for developers to use for their tasks.	A developer tries to install Docker Composer, and finds it incompatible with other installed tools. The developer finds another repository that previously encountered and solved the same problem. The developer then tries to understand the rationale behind the changes that solved the compatibility issue.
	Using	Using external tools that are not part of the default API libraries, either by calling from code or by impacting the execution environment.	A developer wants to use some GUI controls for the website they are building. The developer looks at a similar code repository that uses these controls and goes through its commits to learn how to use the GUI controls. In this effort, the developer needs to understand the rationale of code commits.
	Building	Compiling the source code, for the software to launch and execute correctly.	A developer's code is not building successfully. The developer goes back to an older version of the code to figure out why the build was successful at that time. The developer reads commits related to the problematic code and reflects on the rationale behind the changes before the build started failing.
Specifications	Writing	Writing specifications for functional or non-functional requirements.	A development team wants to write a new specification document to improve the functionality of a system. The team decides to assign the task to a new team member, to obtain a fresh perspective. While thinking of possible improvements, the developer also studies old code commits to understand how the system evolved. This effort takes the developer to ask questions about the rationale of the changes.

Table 2
Tasks for which rationale for code commits is needed (cont.)

Task	Subtask	Description	Example (abstracted from participants' responses)
Documentation	Searching	Searching for reference documentation, tutorials, commit messages, or any other sources of knowledge that can help accomplish a task.	A developer looks for the documentation of a particular piece of code. Because inline comments in the code are not clear, the developer runs git blame to look for related commits. The developer reads the related commit messages, and asks about the rationale of the code changes.
	Writing	Writing documentation of software artifacts (*e.g.,* tutorials, inline code comments, or user-generated JavaDoc) or the software process (*e.g.,* commit messages, process documents, or meeting minutes).	A developer works on writing a tutorial on how to use a library that the team newly migrated to. Since the library documentation is not complete, the developer looks at commits from the library repository, to understand the rationale behind some of its functionality.
	Reading	Reading documentation of software artifacts or software processes.	A developer reads the project changelog and figures out where the support for a library was added. The developer reads the commit to understand the rationale of this change.
Project Management	Checking out	Checking out a software artifact to a separate machine to work on it independently.	A developer pulls the changes made to an existing branch. The developer looks the pull commits to understand their impact on his/her tasks and eventually needs to understand the rationale behind these changes.
	Reverting	Reverting to an older version of the software repository.	A developer inserts an incorrect commit on the project. The team does not know exactly which commit is the problematic one. A developer goes through the changed files for each commit to backtrack the problem. The developer needs to understand the rationale of various code changes in the history to revert those inserted after the incorrect one.
	Deploying *	Moving software from one controlled environment to another, *e.g.,* merging the development branch into the production branch, releasing the software to users, moving the software to a different environment on the customer end, or staging to test the software using real data.	Some tests fail after moving the project from the development branch to the production branch. A developer investigates the commits in the repository and finds a change in configuration that might affect the deployment. The developer reads the commit thoughtfully to find out the goal of each configuration flag, and why they were introduced.
Testing	Writing	Writing test code that checks the behavior of the software against its specifications.	A developer wants to write several tests for new functionality. The developer studies the latest code commits to design his/her test strategy, which also involves understanding the rationale of those changes.
	Running	Running tests to compare the behavior of the software with its specifications.	Before a developer runs a test suite for a system, the developer looks at the code commits to design his/her test strategy. In this effort, the developer tries to understand the rationale behind the changes performed to the code under test.

* Subtasks observed in this study that were not originally in the codebook (*i.e.,* that were not observed in Begel and Simon's study of developer tasks [10]).

"Tell me about one time in which you investigated a code commit to understand its rationale. Why did you need to find the rationale for that specific code commit? What was the rationale for that specific code commit?"

Analysis method: We analyzed our participants' answers using closed coding [53] (also used in *grounded theory* [1]) — *i.e.,* we coded our participants answers, labeling them with categories according to a pre-existing set of codes (a *codebook*). Studies of the tasks developers perform at their job already exist in the research literature. So, we used a closed codebook containing the list of developer tasks that *Begel and Simon* captured when observing software developers at work [10]. We analyzed our participants' responses and labeled them with the task(s) from our codebook that we identified they were performing. We allowed multiple labels for each response, since our participants sometimes mentioned being involved in multiple tasks when needing the rationale of code changes.

First, the first two authors of the paper held a code discussion session, to reach a common understanding of the scope of each developer task and subtask in the codebook. In this session, the first two authors produced a detailed description for each task to delineate their difference better. Then, the first two authors performed their own individual coding. They labeled participant responses with multiple labels in two scenarios: if multiple tasks were mentioned (*e.g.,* "debugging" during "code review"), or when the response could fit multiple tasks (*i.e.,* if it was not clear which one of the many were being described). In both situations, they had discussions to agree on the set of labels that better fit each response, *i.e.,* the superset of tasks that were either mentioned or would easily fit the description in the text. They also allowed the addition of new tasks and subtasks if they were not already in the codebook. The two authors coded about 52% of the tasks similarly, and they coded about 59% of the subtasks similarly. Next, the two authors held a joint focused-coding session and resolved disagreements. After this joint coding session, the two authors resolved most disagreements, but they still disagreed on coding two participants' responses. Therefore, the paper's third author provided additional independent coding for these two responses. Finally, after a final discussion that reviewed all three codings, all authors agreed on how to label these two responses.

Analysis evaluation: We reached saturation [67, 35] in our identified set of tasks; our participants mentioned no new tasks after the sixth interview (out of 20) and in our subtasks after the 14th survey (out of 26).

5.1.2. Results

We show the list of tasks (and subtasks) during which our participants needed to understand the rationale of code commits in Table 2. Table 2 also contains a description and example (abstracted from our participants' responses) that we created to clarify the scope of each task during our coding sessions. Finally, we report in Figure 3A the relative frequency with which each task and subtask was mentioned.

Our participants reported needing to understand the rationale of code changes in a wide diversity of software development tasks: *programming*, *working on bugs*, *communication*, *tools*, *documentation*, *project management*, *testing*, and *specifications*. Their responses covered all (8) tasks in the codebook, and the majority of subtasks in it (23 out of 34 subtasks). Another signal of the wide diversity of tasks requiring understanding the rationale of code changes is the fact that a good portion of our participant responses (35%) described multiple tasks.

For many of the reported tasks and subtasks, it is intuitive that developers would need to understand the rationale of code changes for them, *e.g.,* in *code review*, or when *reading* documentation. Others are initially surprising (*e.g.,* when *installing* or *discovering* tools) but become less surprising after hearing our participants' examples (column 4 in Table 2). For example, when a tool is poorly documented, some of our practitioners resort to understanding its code changes to understand its behavior.

Our main takeaway from these observations is that when the rationale of code changes is well documented, it could help software developers in many different situations. The help could even be in less intuitive situations, such as when they need to understand the rationale of code changes in other codebases that they do not own (*e.g.,* to understand the behavior of external tools). Good documentation of the rationale of code changes should consider the fact that developers seeking it may be involved in a wide diversity of tasks, which could cause them to search for it in different places and search for different dimensions of it under different contexts. Furthermore, it would also be beneficial if the documentation of the rationale of code changes can be understood even by people outside the development team of the software project (24% of responses mentioned seeking the rationale of code changes outside their project).

We hope that this new understanding of how many tasks can benefit from a well-documented rationale of code changes encourages developers to document it well, and doing so appropriately for a wide diversity of tasks (*i.e.,* in adequate locations, and with adequate levels of detail suitable for many contexts).

We also learned from the tasks and subtasks that our participants reported but were not originally included in the codebook: *post-mortem analysis* of bugs, and software *deployment*. We believe that these tasks were not originally observed by *Begel and Simon* because they are intuitively performed less regularly, and are therefore harder to observe. However, our participants reported these tasks in their experiences of needing to understand the rationale of code changes: they needed it to understand why an old bug was introduced in the first place (to avoid similar ones in the future), and they needed it to have a stronger understanding of the new changes that they were deploying. We learn from these observations that the rationale of code changes is needed for both frequent and infrequent tasks.

Similarly, our participants did not mention some developer tasks from the codebook. Some of them were tasks that do not necessarily involve code changes: tools (*finding*), specifications (*reading*), and communication (*finding people*). Others are tasks that are more focused on *providing* information than *requesting* it: communication (*persuasion*, *meeting prep*, *interacting with managers*, and *teaching*). The remaining ones are tasks that we believe practitioners may be more likely to think of them as part of a broader task (and thus may not mention them specifically): programming (*commenting*), project management (*check in*), communication (*asking questions*, *email*, and *meetings*). Overall, there are more tasks and subtasks that require understanding the rationale of code changes than those that may not.

Finally, we also measured the ratio of times that our participants mentioned any task or subtask (Figure 3A). The tasks that were most often reported by our participants were *programming* (32% of mentions), followed by *working on bugs* (22%), and *communication* (14%). Within them, the most popular subtasks were *debugging*, code *reading*, *code review*, *learning*, and *documentation search*.

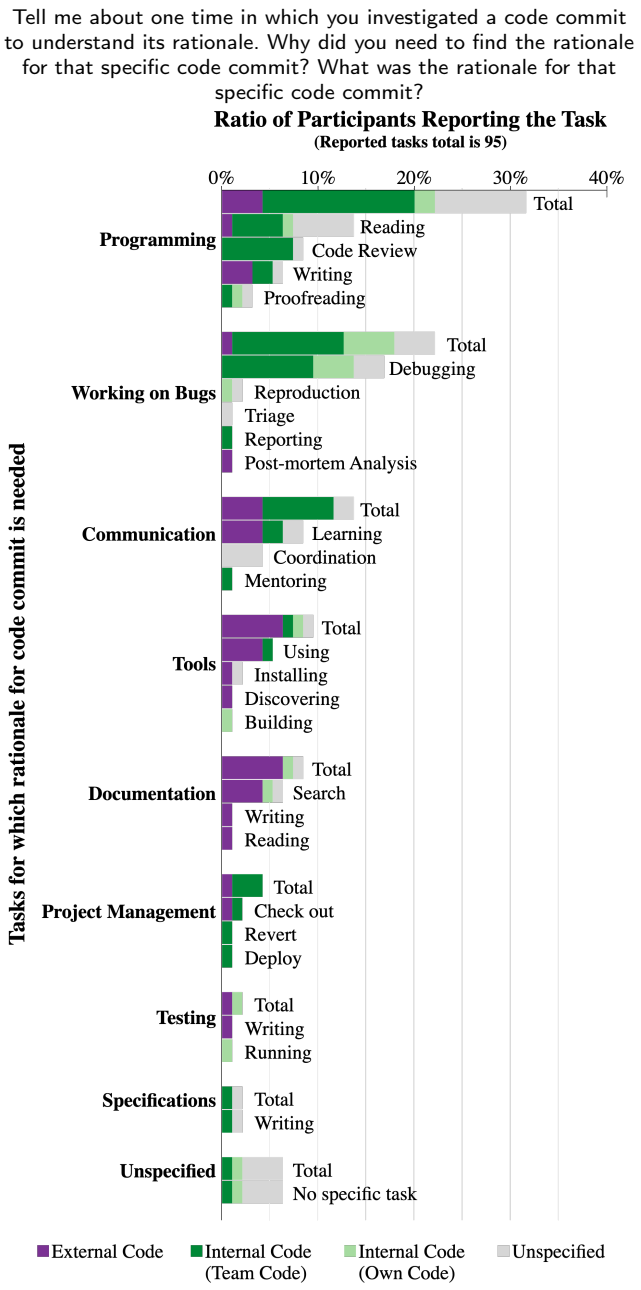
Most often reported tasks teaches us that the most typical scenarios for practitioners needing to understand the rationale of code changes are those that involve *debugging* code or *reading* it to *learn* something about it, often during *code review*, and by also *searching* for its documentation. Therefore, developers documenting the rationale of code commits to help other developers in the most typical scenarios should document the aspects of the code change that could be informative for these tasks.

However, in addition to those scenarios, there is a long tail of tasks that were much less often mentioned in our practitioners' scenarios, but still required understanding the rationale of code commits. Therefore, developers aiming to provide extensive documentation of the rationale of code changes should document all the aspects of rationale that could be relevant for all those tasks and subtasks.

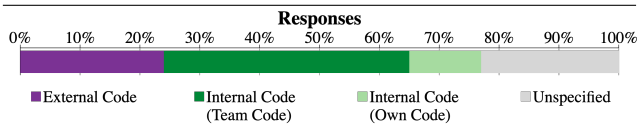
5.2. RQ2: How often do developers seek the rationale of code commits within their team's internal code vs. external code?

5.2.1. Research Method

We used the same method to answer RQ2 as we did for RQ1 (see Section 5.1.1). We again analyzed our participants' descriptions of their experiences investigating the rationale of code commits, this time looking for whether they were investigating internal or external code. As with RQ1, the first and second authors of the paper first performed open coding individually and resolved disagreements afterward in a joint focused-coding session.



(A) Targeted code change per task



(B) Targeted code change

Figure 3: Tasks for which rationale for code commits is needed

5.2.2. Results

We display in Figure 3B the breakdown of how often our participants reported investigating internal code vs. external code. Internal code refers to code changes owned by our participants or developers in the development team for which our participants are contributors to the team's project. External code refers to code changes owned by developers from external projects' teams to our participants. As we did for RQ1, when it was unclear which source code was being investigated, we coded the answer as "unspecified".

Our participants mostly investigated the rationale of internal code changes (53%), but they also investigated it for external code changes (24%). The external code could be code changes to open-source libraries or external projects in the same organization related to the participant's current project. An example of external code changes, as one of our participants mentioned, is *"I was trying to look at forks of that repository and how other people solved those issues."* In most cases, our participants were investigating code changes that were written by other members of their team— internal code (team code) (41%); However, in other situations, the investigated code change was written by the same person investigating it — internal code (own code) (12%). Some of our participants had forgotten the rationale of their own code changes after some time had passed. One participant said, *"It was just a decision made by the person who wrote that who also happened to be me."* We also connected our participants' targeted code change to the software development task they mentioned performing when seeking the rationale. We report targeted code change per task in Figure 3A.

In Figure 3A, we observe three types of tasks. The first type includes tasks for which our participants reported needing to understand only the rationale of internal code. Examples of these are: *code review* and *debugging*. This observation supports an intuitive guess that this type of tasks would only need rationale of code commits in internal code, since they often only involve internal code.

Other types of tasks are those for which our participants reported needing to understand only the rationale of external code. Examples of these are several tasks under the *tools* and the *documentation* categories. We were less surprised that our participants would need to understand the rationale of code changes to external code when the developers investigate tools since tools can often be developed externally. One participant was reading the changelog of a tool he/she was using.

> *" ...one of the software that I work with is called MXNet. It is a machine learning, deep learning library, and I saw in the changelog for MXNet that they added support for one of the libraries it depends on, open CV, which is a computer vision library. It changed how they loaded JPEG and they included support in MXNet ... In the MXNet changelog, they made reference to the commit in open CV where this occurred. So I looked at that commit and in the detailed commit message... "*

We were more surprised that they also needed to understand the rationale of code changes to external code when writing documentation. However, our participants often were creating internal documentation about external code to better keep track of the external code's behavior. For example, one of the participants was trying to write a tutorial for his team about a specific tool, and so he had to discover the tool

in various ways, which led to multiple questions about the rationale of code changes to the tool.

Finally, for other tasks, our participants mentioned the need to understand the rationale of both internal and external code changes. Examples of these tasks are *learning* and *code writing*. It is more intuitive to understand why *learning* could require understanding the rationale of external code changes —to learn from external projects. However, we learned that our participants also often have that need when writing code. For example, when they are trying to port or incorporate external code into their internal project. One participant mentioned:

> *" I worked on a derivative of the bitcoin codebase for two years (named LBRYcrd). There were many occasions where I had to trace the history of some line of suspicious code. Often I would find bugs in the LBRY-specific portion, but rarely in bitcoin... Just recently, as I was working on the Golang version of that, I had to understand where the UPnP code portion came from to see if the author had created a newer version (that might have fixed some bugs I was seeing with it). "*

These observations may explain why other studies observed that finding the rationale of code changes was sometimes easy, sometimes hard [18, 79]. It would be harder to find when the developer is seeking rationale for *external* code changes, since they would have access to fewer documentation resources for external code changes than for internal code changes. The fact that our participants need rationale of external code could explain the challenge of finding rationale and its consistent occurrence as information need in information needs' studies like [18, 79].

More generally, our participants needed to understand the rationale of external code changes most of our studied development tasks (all but "specification design" in Table 2). This observation motivates the creation of external-facing tools and practices to improve the documentation and explanation of the rationale of code changes, not only to internal stakeholders but also to external ones. We discuss ideas for how to achieve this in Section 8.4.

6. Theme 2: Components of Rationale

6.1. RQ3: Which components do software developers decompose the rationale of code commits into?

6.1.1. Research Method

Data: The data for this research question is a large set of rationale for code commits components. We proposed nine components based on the literature references to rationale. In addition, our participants proposed 18 different components of rationale for code commits.

Analysis method: Our goal is to create a mental model and derive a taxonomy of the components. Therefore, we used card sorting [75] to discover the components of rationale for code commits. Card sorting is a widely used inexpensive method with three phases: preparation (participants selection and cards creation), execution (cards sorting into meaningful groups), and analysis (hierarchies formation). For the card sorting preparation phase, we prepared the cards of all the components from the preliminary model and the participants proposed final model. Then, for the execution phase, we (the first and last authors) performed individual open card sorting. We individually sorted the cards without using predefined groups. During the individual sorting, we aggregated those cards that described similar components. For example, we aggregated into *"Side Effects"*: the preliminary component *"Costs"*, and the *"Merge Conflict/Success"*, *"Limitation"*, and *"Impact"* components that were mentioned by different participants. After that, both authors collaboratively consolidated the two sets of individually-aggregated components, comparing them and deciding on disagreements. Then, we characterized each of the resulting aggregated components with a name, a question, and an example answer to the question based on a hypothetical commit. Finally, we categorized the resulting components into themes for the analysis phase.

Analysis evaluation: Our discovered model of rationale for code commits (Table 3) is of 15 different components. The fact that many participants added and some removed components suggest that our participants were not strongly biased towards simply agreeing with the preliminary model (Table 1). Also, although we interviewed 20 software developers, we reached saturation in the 15th interview.

6.1.2. Results

We display in Table 3 the model of rationale of code commits that we discovered. It represents the union of all the models that our participants reported. As we discussed in Section 4.1, we obtained this model aggregating all the components that were mentioned by at least one participant in their final interview model of rationale of code commits. Each participant built their final model by adding and removing components to the preliminary model, while also considering their own rationale decomposition. Altogether, our participants reported 27 components of the rationale of code commits, adding 18 components to the nine components in the preliminary model. Since many of those components reported very similar concepts, we aggregated them using card sorting to obtain the final model that we show in Table 3. This resulting model of rationale of code commits includes 15 components into which developers decompose it. We categorized the resulting components into four themes.

Our goal with this rationale model was to gather an extensive set of specific components of the rationale of code commits, which developers may be looking for when they need it. For that reason, when a participant removed a component from the preliminary model, we still kept it in our resulting final model (in Table 3). Besides, only a few participants removed components.

When participants decided to remove components from the preliminary model, they mentioned two main reasons: overlap with other components, and the component being out of scope. In terms of overlap among components, one

Table 3
Resulting model of the rationale of code commits

Theme	Component	Component Expressed as Question	Example Answer
Change Objective	*Goal	What did the developer want to achieve?	The code is this way because the developer *wants to modify* the usage of try/catch blocks to account for unexpected Exceptions.
	*Need	Why did the developer need to achieve that?	The code is this way because the developer *needs* to improve Exception handling by June 1st as per the new *company demand* to eliminate exceptions before release.
	*Benefits	What is the benefit of what the developer wants to achieve?	The code is this way because handling exceptions that were not considered before will benefit in increasing the system's quality.
Change Design (pre-implementation assessment)	Constraints	What were the constraints limiting the developer implementation choice?	The code is this way because the developer choices are *limited* by the team development *guidelines* that *prohibit hard-coded Sring use*.
	Alternatives	What other alternatives did the developer have?	The code is this way because the *alternative bucket sort implementation* option is not feasible since the maximum value is unknown.
	*Selected Alternative	Why did the developer make those specific changes and not others?	The code is this way because heap sort has the *advantage* of being space *efficient* and has a *predictable speed*. Other sorting algorithms options are not as efficient and predictable.
	Dependency	What other changes does this change depend on?	The code is this way because it *depends* on the *API response format*, which needs to be updated to provide JSON format.
Change Execution	Committer	Who changed the code?	The code is this way because it was introduced by *Developer X*, who was our *short-term consultant hired to improve the security* of our software system.
	Time	Why were the changes made at that time?	The code is this way because it was introduced *four months ago* to meet *3.0 release cycle*.
	*Location	What artifacts were changed?	The code is this way because, in our MVC architecture, the *model*, *view*, and *controller* are updated together when introducing a new data field.
	*Modifications	What specific changes were performed in the artifacts?	The code is this way because the developer *altered* the user interfaces' look and feel, including *color* and *layout*.
	Explanation of Modifications	What are the details of the implementation?	The code is this way because look and feel are *altered by*: 1- Changing all interfaces colors to match color palettes provided by web accessibility guidelines. 2- Changing all interfaces layouts to be responsive to screen size.
Change Evaluation (post-implementation assessment)	*Validation	How did those specific changes achieve the goal?	The code is this way because the goal is to account for an edge case, which newly added *test cases* show the edge cases examples and verify code success in handling the edge case.
	Maturity Stage	How mature is this code?	The code is this way because it is an *experimental hack* created to explore ways to fix a persistent bug.
	*Side Effects	What are the side effects of the change?	The code is this way because of *side effect* mitigations; Temporary control statements are added to *avoid integration test failure* until the API is updated.

* Components that were included in the preliminary model of rationale of code commits. We extended the preliminary component *Costs* to *Side Effects* to include other side effects mentioned by participants *e.g., Impact*.

participant thought that *goal* and *need* can be the same most of the time and preferred to merge them, deleting the *goal* component. Another thought that *need* is included in *benefits* and *cost*, deleting the *need* component. Another participant deleted *benefits* because it is included in *goal*. Another one considered *location* as part of *modification*.

We believe it is possible that different components' answers can be the same in some cases. For a single code commit, components of the same theme (see Table 3) may have very similar answers to their expressive question. However, they will be different in many other cases, making it worthwhile to separate those components. We illustrate the differences between components in Table 3 by including the components expressed as questions and different example answers for different ones.

Other participants removed components that they considered out of scope of rationale. From our 20 interview participants: two participants removed *modifications* because they considered it too low-level; three participants removed *location* because it would not tell why the changes were made; three participants removed *alternatives*, *e.g., "alternatives is not something that you actually implement!"*; and one participant deleted *validation*, saying that *"validation answers why the code is correct, not the rationale"*. Despite these disagreements, the majority of our interview participants (18, 17, 17, and 19, respectively) considered that these components do belong in the rationale of code commits.

Furthermore, our participants generally provided positive comments about the preliminary model — describing it as *e.g., "a good model," "detailed," "thorough," "comprehensive," "holistic,"* or *"exhaustive."* They thought

that the model *"formally define[s] rationale"* and that *"the components seem to be related to each other, but classified differently to each other."* One participant said that the model is a

> *"logical framework for thinking through rationale because [it is] a sort of wide-open concept. It is a little bit hard to know how to think about [rationale]. [The model] makes sense as a directed way to understand a specific commit or a series of commits. Why they are the way they are."*

Many participants added components to the preliminary model. As we mentioned earlier, we used card sorting to aggregate them to the preliminary model and with each other. The 18 components proposed by participants were: *technical requirement*, *timeliness*, *documentation*, *guidelines*, *non-feasible alternative*, *opinion selected alternative*, *constraints*, *dependency*, *committer*, *time/date*, *explanation of modifications*, *result*, *environment*, *scope for future development*, *quality*, *merge conflict/success*, *limitation*, and *impact*. For each added component, we also asked participants to describe them with a name, expressive question, and example answer to the question.

The fact that many participants added and some removed components suggest that our participants were not strongly biased towards simply agreeing with the preliminary model. More importantly, it also suggests that different developers seek different components at different times. Our study throws light into this phenomenon. Thus, we posit that the rationale of code commits would be much easier to comprehend, search for, and document when expressed as its components– not necessarily all of them at all times, but the ones relevant to each situation.

7. Theme 3: Experience with Rationale

7.1. RQ4: What is the experience of developers needing, finding, and recording the rationale of code commits?

7.1.1. Research Method

The *data* for this research question is quantitative. We asked both the interview and survey participants the likert-scale-style question shown in Figure 4. To *analyse* the participants experiences' responses, we compare and report the responses' statistics.

7.1.2. Results

Figure 4 shows the interview/survey questions which we asked about the developers' needing, finding, and recording experiences along with the distribution of the participants' responses for each question.

Need: The participants of our study reported needing to seek rationale with diverse frequencies (see Figure 4A-i): multiple times per day (27%), multiple times per week (29%), multiple times per month (27%), multiple times per year (13%), and a few times per year (2%). Overall, the majority (56%) of our study participants need rationale relatively frequently: multiple times per week or more often. The inconsistency of the need for rationale for code commits is because of the diversity of the software developers' roles and their work activities. One participant said

> *"I do a lot more than just software engineering on a yearly basis. And so there are periods of time when I am doing primarily software engineering, and there are large periods of my work time that I am not."*

When asked about how important it is to understand the rationale of code commits, 86% of our participants reported needing the rationale of code commits (see Figure 4A-ii), from which: 7% cannot complete their work without understanding it, 30% struggle to complete their work without it, and 46% can complete their work without it but still need it. The remaining 17% do not need the rationale of code commits, but 15% of the 17% report that it still helps them complete their work. A very similar question was studied by *Tao et al.* [79], whose participants *"generally considered knowing the rationale of a change as the top priority in change-understanding tasks"*. Our finding is aligned with theirs since a majority of our participants reported needing the rationale of code commits, which validates that we are studying a similar population of developers.

Finding: Our participants' responses in Figure 4B-i indicate that the difficulty of finding the rationale of code commits, in general, is not easy nor difficult. Software developers (on average) selected neutral difficulty of finding the rationale of code commits. This finding also generally agrees with *Tao et al.* 's, since their participants reported that the rationale of code commits was generally easy to find, but sometimes hard, depending on *"the availability and quality of the change description"* [79].

Regardless of how hard it is, we were also interested in how often developers end up finding the rationale of code commits altogether. For this aspect, our study participants responses are positive (see Figure 4B-ii). Most software developers find the rationale of code commits often or almost always. Only a few participants (11%) rarely or almost never find the rationale of code commits.

In addition to studying whether software developers find the rationale of code commits, we also studied how much time they spend searching for it. Figure 4B-iii and 4B-iv shows the times that our participants reported spending when searching for rationale. In the usual cases, slightly more than half of our participants (55%) spend less than 10 minutes. However, in the hard cases of searching for rationale, only slightly less than half of our participants (46%) spend more than 30 minutes searching for the rationale of code commits. One participant said about the time they spend searching for rationale in the hard cases that it *"depends how responsive the other person is."* A considerable amount of time, 68% of the participants spend more than 20 minutes, is spent by software developers when it is hard to search for the rationale of code commits. When considering the relatively high frequency with which developers search for rationale of code commits, it can be a rather time-consuming task.

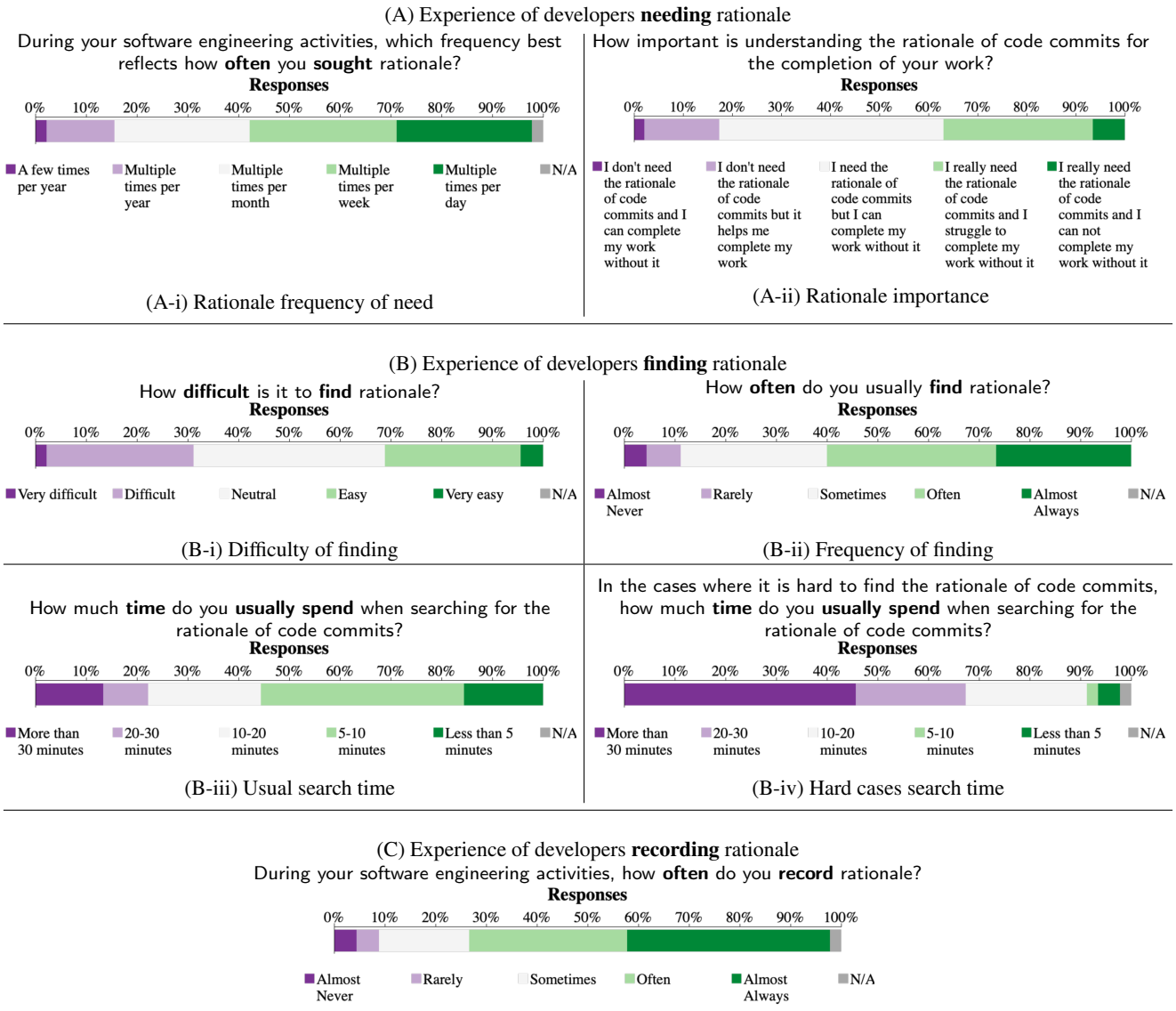


Figure 4: Experience of developers with the rationale for code commits

Recording: Regarding the frequency of recording rationale in general, Figure 4C shows our participants' responses. Our participants reported high involvement in recording the rationale for their code changes.

The majority of them (71%) reported recording the rationale of code commits often (31%) or almost always (40%). These ratios are very similar to the frequencies with which they report needing it and finding it. However, there may be multiple explanations for why these two ratios are similar. It could be that the teams to which our participants belong are generally diligent about documenting the rationale for their code changes, and that is why they report finding it at similar frequencies when they need it. However, there could be more nuance to this observation: our participants may sometimes find the rationale that was documented, and other times find it after asking their colleagues (because it was not documented). Also, the found rationale by our participants is a reflection of the documentation habits of their teammates than theirs. Our observation motivates further study of the extent to which the specific rationale that developers document is what ends up helping their teammates later find it (*e.g.,* with observational studies or mining of software repositories). We take one further step in understanding the similarity between the frequency of recording and finding the rationale of code changes in more depth in RQ7 (see Section 7.4), in which we observed that such similarity is generally preserved for individual components of it.

7.2. RQ5: What is the experience of developers needing, finding, and recording the individual components of the rationale of code commits?

7.2.1. Research Method

Just like RQ4, the *data* for this research question is quantitative. We asked likert-scale-style question shown in Figure 5. To *analyze* the participants experiences' responses, we compare and report the responses' statistics. For the

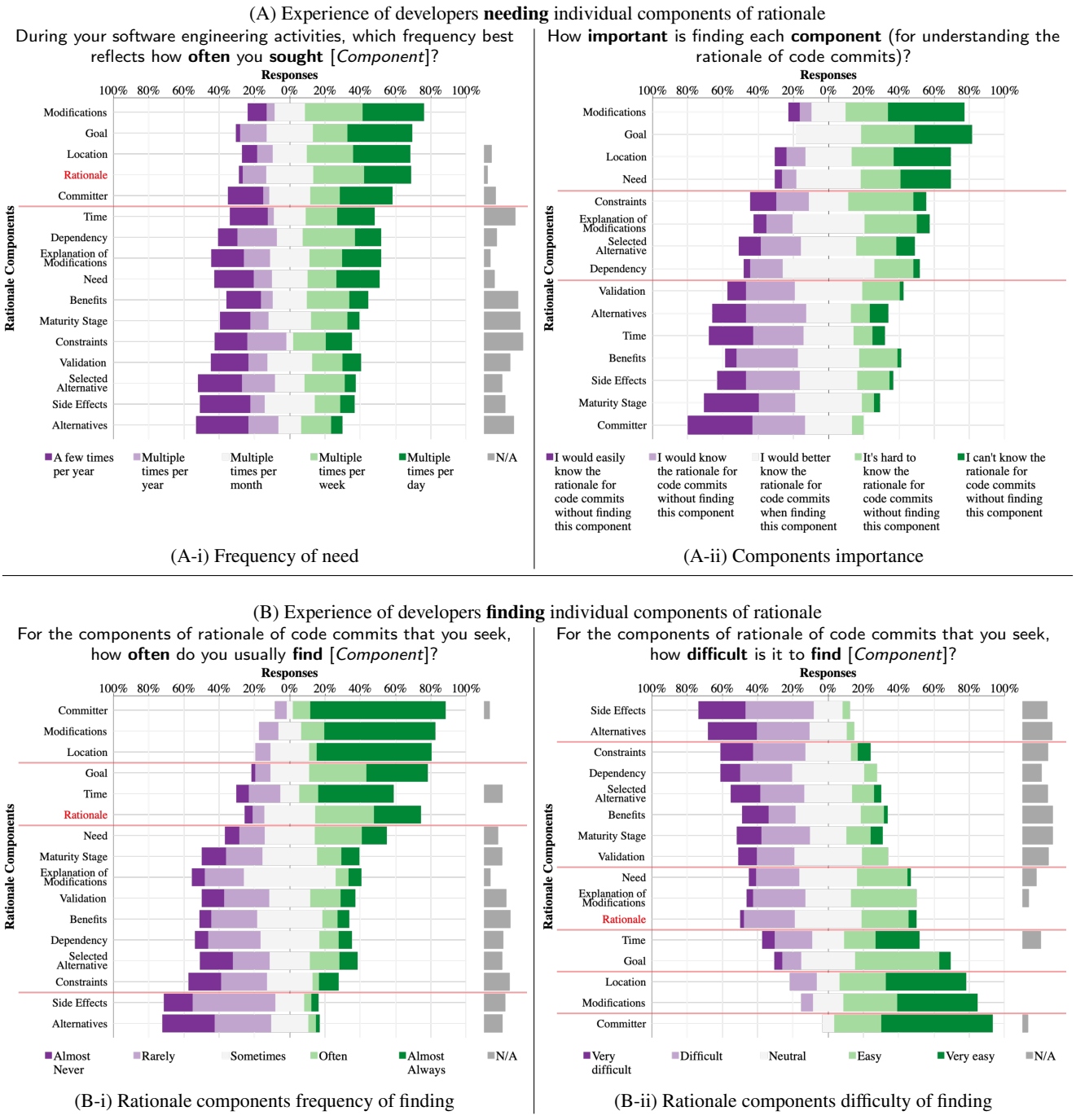


Figure 5: Experience of developers with individual components of the rationale

individual rationale components, the answers include only the interview participants that included them in their final model. Whenever we aggregated components through card-sorting, we also aggregated the responses about the experience. The reported experiences with rationale components combine the aggregated interview and survey responses.

In addition to the responses' statistics, we wanted to cluster components with similar experiences. Clustering the components made it simpler for us to compare the experiences. We used Scott-Knott [28] clustering algorithm to group the components which have similar software developers' experiences. Scott-Knott is a hierarchical clustering algorithm that serves in the Analysis of Variance (ANOVA) contexts. For the individual experiences (need, finding, and recording), the algorithm compares the means of all component's responses. This comparison results in a non-overlapping grouping of rationale components. We present the algorithm outputted groups by the red borderlines in Figure 5.

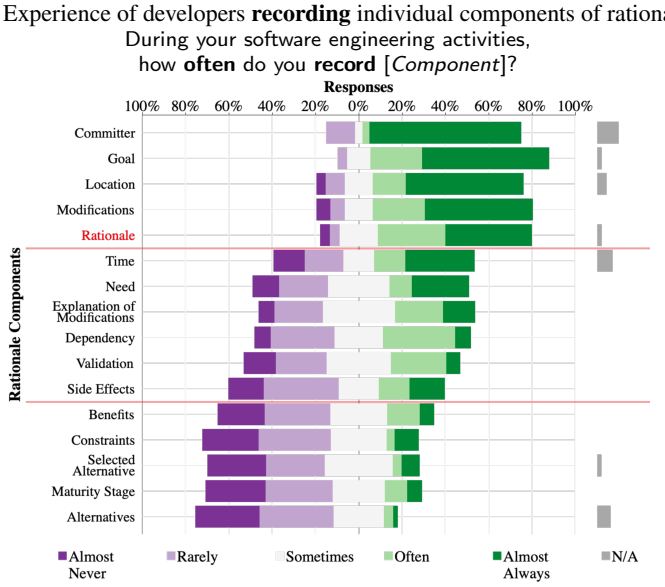


Figure 5: Experience of developers with individual components of the rationale (cont.)

7.2.2. Results

We wanted to expand our knowledge about software developers' experiences with rationale components and how it differs from their general experiences with rationale.

We plot the distribution of answers to our questions about individual components of the rationale for code commits in Figures 5A–5C. We cluster components into similar groups according to the mean value of their responses using the Scott-Knott [28] algorithm. We sort the components in our figures by the mean value of their responses, and we use red horizontal lines to separate clusters. As a reference point, we also include in each figure the responses that our participants gave for rationale in general.

Need: Figure 5A-i shows the distribution of responses for how frequently developers need each component of the rationale of code commits. Overall, the frequency with which developers need different components of the rationale of code commits is highly similar for all components and rationale itself in general. In this case, the Scott-Knott algorithm returns only two very-similar clusters. While some of the most often needed components (like *modifications*, *location* or *committer*) are normally automatically recorded by revision control systems, many other components are similarly often needed and are not recorded automatically (like *need* or *dependency*, or *constraints*). These results show that practitioners would benefit from regularly recording these frequently-needed components.

Figure 5A-ii shows the relative importance of each component to understand the rationale of code commits reported by developers. These results show that most developers mentioned that most components are important enough to understand the rationale of code commits better if they knew that component. We also observe that developers wanting to document the most important component of rationale should focus on documenting the *goal* of their changes, since the other most-important components (*modifications*, *location*) are already recorded by revision control.

Finding: Figure 5B shows the relative frequency and difficulty of finding reported by developers for each component. Unsurprisingly, the most frequently found components (and also the easiest to find) are those automatically tracked by revision control (*committer*, *modifications*, and *location*), followed by *goal* and *time*. However, the frequency (and easiness) of finding drops quickly for all other components, bringing our attention to a clear problem in finding the remaining components. These results highlight the need to improve documentation for the other components since they are hard to find. This clear divide could also explain why developers talking about rationale (in general) say that sometimes it is much harder to find rationale than other times [79] and it takes longer (Figure 4B-iv).

Recording: Figure 5C shows the relative frequency with which developers reported to record components of the rationale of code commits. Again unsurprisingly, the most frequently recorded components are those recorded automatically by revision control, but again the frequency of recording drops dramatically for the remaining components (which probably explains why they are hard to find). These results show that, even if developers claim to frequently record rationale in general, there are many components that they are not recording frequently (even if they are relatively often needed).

7.3. RQ6: What makes software developers give up their search for rationale of code commits?

7.3.1. Research Method

We asked our (20) interview and (26) survey participants an open question about when they give up their search for rationale of code changes. We analyzed their responses qualitatively, using open coding to extract the factors that our

Table 4
Factors leading software developers to give up the search for rationale of code changes

Factor Category	Give Up Factor	Give Up Factor Description
Project-centric Factors	Codebase state	The codebase state includes (1) the repository state (*e.g.,* the number of commits in a pull requests under review) and (2) the code aspects (*e.g.,* the code is/is not running, the code has design/readability issues, the code is reviewed).
	Documentation	The documentation factor includes (1) artifacts documentation (*e.g.,* Javadocs, inline comments) and (2) development process documentation (*e.g.,* commit messages, log messages).
Human-centric Factors	Effort management	Effort management involves the developers' assessment of their work activities in terms of the required (1) effort and (2) time. This assessment is typically followed by a decision to possibly give more priorities to some of these activities, discard or replace some others.
	Developer knowledge	Developer knowledge involves the developer possessing/awareness of (1) background knowledge to comprehend the code change under inspection (*e.g.,* programming language, mathematics, technical knowledge) or (2) project knowledge (*e.g.,* project design aspects, implementation aspects, and testing aspects).
	Interpersonal (Emotions)	Interpersonal factor involves human sentiment or attitude in the situation (*e.g.,* frustration, confusion, fear of obtrusion to the code review).
Team-centric Factors	Impact on productivity	Impact on productivity includes the developers' awareness of the impact of their actions on the project progress (*e.g.,* work progress stalls).
	Personnel	Personnel includes aspects related to the development team personnel (*e.g.,* developer is not around, project manager changed).
	Time management	Time management includes the team time-related aspects (*e.g.,* time crisis (deadline), team time-frame for the project).

participants reported. All the authors of the paper were involved in the coding. We reached saturation in our observed tasks in response 20 of 46.

Data: For this research question, we asked our participants an open question: *"When do you give up the search for rationale of code changes?"*. We asked this question during our interviews and survey. We used a survey in addition to the interviews to reach saturation in our observations — an initial analysis of the interview responses showed that we were still making new observations in the last responses.

Analysis method: We analyzed our participants' answers using open coding [53] (also used in *grounded theory* [1]). We decided to apply open coding for this research question (as opposed to closed coding, as in RQ1) because it was not clear that any pre-existing list of "reasons to give up tasks" would fit this context well. We decided that it would be more appropriate to have our codebook emerge from our participants' responses in this case. We labeled each participant's answer with one or multiple codes that best described the reported factor(s).

First, the first two authors of the paper individually coded the interview responses. Then, they had a discussion to merge their codebooks and reach a common understanding of the scope of each code. During this meeting, they also categorized the resulting codes under three broader themes: *project-centric*, *human-centric*, and *team-centric* factors. Then, they individually reviewed and updated the labels of each participant's response, according to the merged codebook. After this step, the two authors reached an agreement ratio of about 85%. Then, they held a joint focused-coding session where they resolved all disagreements.

This first effort did not allow us to reach saturation — the last response mentioned a new factor that was not previously observed. Thus, we decided to obtain more responses to the same question, running an additional survey that we described in Section 4.3.

The first two authors of the paper individually coded the survey responses, using the codebook obtained from their analysis of the interview responses. They allowed the addition of new codes to the codebook if new factors were observed. The two authors coded about 46% of the survey responses similarly. Then, they held a joint focused-coding session to compare their coding and resolved all disagreements. After coding all the survey responses, no new codes were added to the codebook.

Analysis evaluation: After coding all the interview and survey responses, we reached saturation in our observed give-up factors —we observed no new factors after the 20th response out of 46 (the last interview response).

7.3.2. Results

We present the results of this research question in Table 4 and Figure 6. Table 4 shows the factors that our participants reported for giving up the search for the rationale of code changes. It also contains a description of each factor. Figure 6 shows the ratio of how many times our participants reported each factor. Our eight observed factors were mentioned 69 times by our 46 participants since some of them (16) mentioned multiple factors.

Figure 6 shows our three observed categories of factors being reported relatively similar rates: the most often reported factors were project-centric factors (39%), followed by human-centric (30%) and team-centric (19%) factors. The remaining 12% of responses form the *unspecified* category: these either did not provide a valid specific answer to the question (6%) or reported never giving up the search (6%). Next, we describe each category in more detail.

Unspecified factors: An example of an invalid response stated why it is important to understand rationale without mentioning giving up or giving up actions. Examples of participants reporting not giving up either stated it explicitly

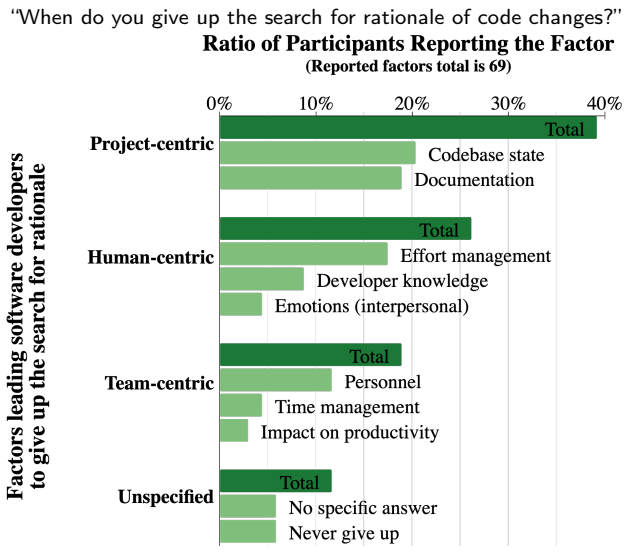


Figure 6: Give up factor

or reported giving up only temporarily. Sometimes, giving up temporarily meant *switching to another task*, leaving the rationale-demanding task for a later time:

> *" Essentially, if I cannot talk to the person who committed it, I will usually just postpone until they are back online. "*

Other times, our participants paused to rest and returned to the task later with a fresh mind or after acquiring more background knowledge.

> *"I usually do not give up. I mean, I just go run or sleep, and then I try again the next day. With a clear mind, or something like that."*

> *"Well, sometimes, if I want to do something and implement something math-intense, perhaps I will go first and revise my math knowledge behind this. "*

Project-centric factors: Our participants most reported give up factor is *codebase state* (20%). When the code base state is problematic (*e.g.,* "commit is quite large", "changes are too much to be able to track them", "variables are not properly named"), our participants may give up their search for rationale of code changes. This observation shows that past efforts to aid in the comprehension of source code, *e.g.,* efforts to untangle large code changes [36, 37, 22], have additional benefits. Any effort that helps improve the comprehension of source code would also help developers not give up their search for the rationale of changes to that code.

Our participants also reported giving up due to poor/lack of *documentation* (19%). Lack of good documentation here refers to both the source code documentation, *e.g.,* JavaDocs, and the documentation of the development process, *e.g.,* commit messages.

Poor documentation in source code is a well-known problem [2], and we found that it affects the search for rationale too. One participant mentioned:

> *"In well-documented code it is very rare that doing this is necessary."*

Our participants also highlighted the importance of documenting the change itself and not only the code.

> *" Committing with a summary might not be enough sometimes. It is better to add a broader description to give a more detailed idea about the changes. "*

In general, our participants communicated giving up the search for rationale of code commits when the "commit lacks description", "old documentation is not found on GitHub". They also give up when the commit message "is vague", "not illustrative enough", or "not descriptive enough to help me understand what is going on".

Even when best practices are specified to document code well, they can be followed inconsistently or the granularity of the documentation can be too coarse.

> *" I wish the commit messages were more granular so that they exist on each file level instead of the whole commit event. "*

The automatic generation of commit messages has been proposed in multiple research efforts [41, 42]. Our observed responses call for attention from the researchers working in the commit message auto-generation area. Our observed responses call for designing techniques that generate more granular commit messages, potentially documenting some of the rationale components presented in RQ3.

Human-centric factors: These factors refer to the internal state of the developer, more specifically to the developers sense of effort (17%), knowledge (9%), and interpersonal emotions (4%).

Some of our participants evaluated the *effort* required to find the rationale of code commits. We believe that effort management is an intuitive give-up factor since developers can spend long hours or even days attempting to find the rationale of code changes. Developers often have several tasks to do, and they want to move on. They can get frustrated if they spend long time and effort being blocked by a single task. Our participants commented on the time sent search for rationale of code changes:

> *"If it takes half an hour, it is not worth spending more time [on the search]. Then, I will ask others and interrupt their work"*

> *"I would say at the hour mark, because at that point I would be like, "I just need to do something, and I will make a best-case judgment call.""*

> *"If it is taking me more than 30 minutes [to find the rationale of a code commit], I will start trying and get in touch with the person who wrote the code, at which point it becomes an asynchronous process. So I am no longer sitting there trying to figure out the rationale; I will email/message the person, then I move on with whatever I was doing."*

Other participants mentioned *lack of knowledge* as a reason to give up searching for rationale. Their responses

referred to knowledge to background and skills, or knowledge about the project. Examples of the first case are when a developer is not fluent in a particular programming language or not familiar with the underlying mathematical foundations of the code. Examples of the second case are when a developer is responsible for certain project areas (*e.g.,* back/front-end) and lacks knowledge of other project areas. As a result, we see the existence of this factor as a normal part of software development. Educating developers could mitigate this factor, but in many cases, it is not an affordable option in terms of cost and developer time. In other words, this specific give-up factor might be one of the main prompts for documenting the code change rationale since it is inevitable on many occasions. Furthermore, this factor, perhaps, encourages documenting rationale in a way that other developers can understand regardless of their detailed knowledge of the project.

Finally, other participants expressed that *interpersonal (emotions)* play a role in giving up their search for rationale of code changes. One participant answered:

> *" No specific time cut-off, I would say. I would say it is more…how frustrated I am "*

Other emotions that we observed were confusion, fear of obtrusiveness, and anger. Software developers' sentiments and emotions are important aspects that could impact their work [31, 58, 32]. Our findings suggest that negative emotions like frustration impact the software developers while searching for rationale of code commits. Therefore, we encourage future studies of the relationship between sentiments and rationale. We also encourage future studies of how to mitigate the impact of these sentiments while still satisfying developer information needs.

Team-centric factors: Our participants expressed that changes in the project *personnel* (12%) introduce obstacles to communication and could prevent them from finding the change owners' rationale. One participant described a give-up situation caused by personnel that left as

> *"[The change] was a decision made by [the previous project manager] that is not even here anymore. We do not really understand the reason, but we make a new decision from this point forward."*

Personnel changes are common in software development teams. In such situations, we think that standard documentation guidelines could make the search of rationale for code changes easier. One participant described his/her company guidelines:

> *"At my company, typically we link each commit, branch, and PR to a ticket number, which should include details and discussion about the change and why it is being made."*

Our participants also give up their search for rationale of code changes to accommodate project-related *time management*. For example, in a rush to fix a bug before approaching a software release, one participant said that he/she came up with some rationale for a buggy code change to provide a temporary fix of the bug. One participant answered:

> *"When I have to, it could be when I am not getting other things done because I am searching for this rationale. Working intimately with another partner, our progress was halted until I completed or gave up. In other contexts that I have somewhat been associated with a large team, I do not need to understand it as long as other people on the team do."*

Composite factors: Sixteen participants (34%) mentioned more than one give-up factors. Twelve of them mentioned two factors in conjunction. For example:

> *"I will look at the commit (code and description) as well as the pull request, including the commit and the ticket relating to that pull request. If the implementer can still be contacted, I will do so. If none of this leads to any results, I will usually give up. "*

The remaining 4 participants mentioned two or more factors in disjunction. For example:

> *" Usually, there are certain scenarios where I give up finding rationale, if I cannot run the code, or if it is a language that I cannot understand, programming language. Or the code base is too big. "*

When participants report factors in conjunction, they show that many factors need to co-exist for them to give up. Most of the time, they mention the lack of documentation in addition to another factor. The second factor is sometimes the codebase state *(3 times)*, showing that the commit is large or complicated, making the search for rationale a tedious process. Some other times, the second factor besides the absence of documentation is the absence of the author *(3 times)*, which deprives the developer seeking the rationale from, possibly, the two most helpful information sources.

When our participants report factors in disjunction, they show multiple individual reasons why they may give up (one of them is enough). For example, they may give up to save effort (effort management factor) given the large size of the commit (Codebase state), or because they were not sure they had the required project knowledge to understand the code change (Developer knowledge) given the poor code readability state (Codebase state).

Another case that we found interesting is when the participants use a combination of factors mixing negative and positive instances of them *(3 times)*. For example, a participant's decision to give up seeking rationale was based on a close project deadline, but the decision was eased by the fact that the code was running correctly, in which case seeking rationale was not a pressing need.

> *" ...Then, the main concern was to finish the project before the deadline. Sometimes in such a situation, if the codes work, then I did not try to check the rationale. "*

We believe that the search time required to find the rationale of code commits is likely connected with the difficulty of finding it (Section 7.1.2), *i.e.,* the rationale for code commits may be easy to find when it is available and well documented. When the components sought by the

developer are documented where the developer is expecting, the rationale for code commit could be easy and fast to find.

The problems revealed by the team and project factors are: the reliance on the availability of the change owner, the trade-off between searching for rationale and productivity/time, and the quality of a project's codebase and documentation. We discuss in Section 8.4 how future tools could support developers in their search for the rationale of code commits, informed by our observations.

7.4. RQ7: Would comparing the experience of developers needing, finding, and recording the individual components of the rationale of code commits with each other reveal areas for improvement?

7.4.1. Research Method

The data for this research question is the same as RQ5. We performed a cross-experiences analysis to further investigate the developers' experiences with rationale components' need, finding, and recording. We paired every two experiences (need-finding, need-recording, and finding-recording) then used the median responses of each component to discover needed areas of improvement.

7.4.2. Results

Figures 7A and 7B show that software developers are most frequently finding and recording the most frequently needed components of rationale. Most of the components are in the middle frequency of need and finding. However, this result reveals that many components are not too frequently needed, but when they are needed, they are really hard to find. Developers most struggled to find *side effects*, *alternatives*, and *constraints*, even if they need to find them on average multiple times per month and per year. In these cases, the difficulty of finding these components may overcome their limited frequency of need. Thus, practitioners may want to pay more attention to documenting these not-so-frequently-needed components.

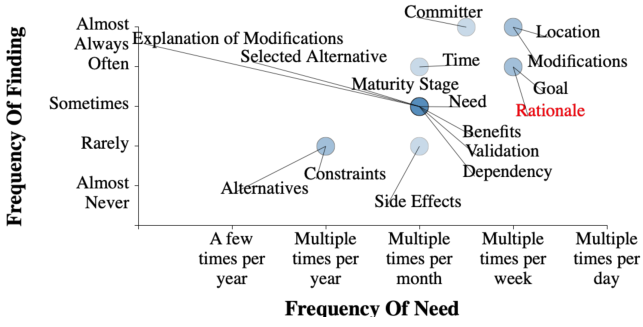
The difficulty of finding rationale depends on many factors, *e.g.,* the complexity of code commits, the developers' documentation of code changes, and the need for discovering the rationale. One of the participants said about the giving up of searching rationale

> *"I would completely give up if I could not find any record in our system and the author was someone who either is no longer at our company or is somebody who just does not write code anymore. Yeah. I give up when I have exhausted all the possibilities, but if I really need to know, I would keep trying until I figured it out."*

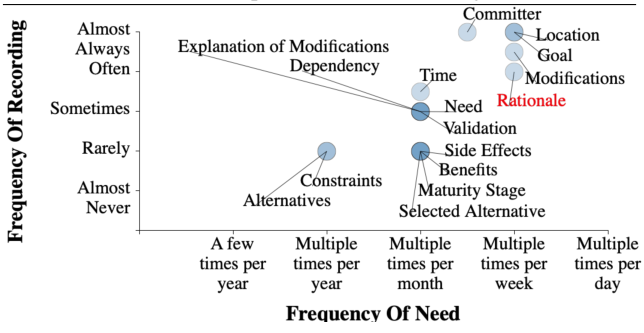
For the components of rationale that are not easy to find, guidelines could be established, and tools could be developed to simplify finding these components. One participant said about finding rationale:

> *"From my experience, the rationale, it is easier to figure out once your team kind of has standards or guidelines."*

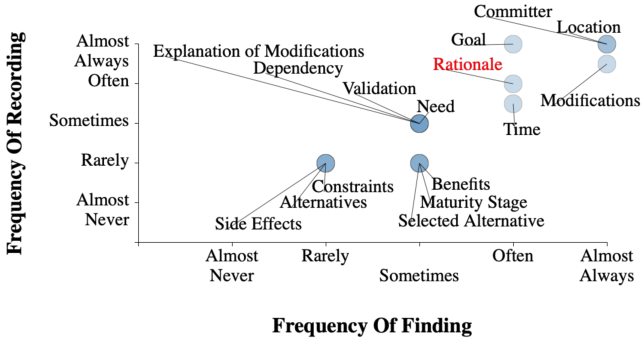
The recording of rationale goes hand in hand with the finding of rationale (see Figure 7C). Unsurprisingly, not recording some components makes it hard to find them later. The rarely recorded components were: *side effects*, *alternatives*, *constraints*, *selected alternative*, *maturity stage*, and *benefits* — even when developers need to find them on average multiple times per year (*alternatives* and *constraints*) and per month (remaining ones). Identifying this group of rarely recorded components should encourage researchers to develop tools specifically focused on recording or answering them. For example, a technique to evaluate the maturity stage of a commit will aid developers in seeking this component without the need for other developers to document it manually.



(A) Components need vs. finding



(B) Components need vs. recording



(C) Components finding vs. recording

Figure 7: Cross-dimensional analysis of developers experience with the individual components of rationale of code commits

8. Discussion and Implications

This section discuss our findings and their implications.

8.1. Is the need for rationale of code commits different than the need for rationale in other contexts?

Some of the components of the rationale of code commits that we discovered are also relevant for rationale in software requirements, design, and architecture. These are: *constraints* [34, 77, 30, 81, 33], *alternatives* [51, 56, 33, 34], and *validation* [34, 30, 33]. However, we also discovered components that are specific to the scope of code commits: *committer*, *time*, *location*, and *modifications*. They generally refer to performing the code change. Our participants indicated that these components are more needed and important than those that also show in rationale in other contexts (see Figures 5A and 7A). Similarly, some components were not mentioned in our study and were reported by previous work as part of the design rationale. Examples are: design *assumptions* [77, 30, 81] and *weaknesses* [77, 30].

8.2. Providing a better understanding of the need for rationale of code commits

The observations in this study illustrate the importance of supporting developers in documenting and finding the rationale of code commits. Software developers regularly need the rationale of code commits (RQ4, see Figure 4A-ii) and spend a significant amount of time searching for it (see RQ4, Figures 4B-iii and 4B-iv).

Our observations also allow us to speculate about the root cause of this problem. We observed that most components of the rationale of code commits are frequently not recorded (RQ5, see Figure 5C and RQ7, see Figure 7B), not found (RQ5, see Figures 5B-i and RQ7, see Figure 7A), or difficult to find (RQ5, see Figure 5B-ii). This may be because finding rationale for code commits could be very easy when the rationale is well documented, or the change owner is easily available to provide the rationale.

We also observed that finding the rationale of code commits may cause productivity loss and sometimes without gain. Furthermore, even after investing high effort searching for it, a variety of factors may lead them to abandon the search (RQ6, see Table 4), making their productivity loss more serious, since they did not get any value from the time spent. Furthermore, after giving up their search for rationale, developers may resort to speculating their own understanding, which may lead to introducing code errors.

> *"At one point, I gave up understanding why they did what they did. I was confused as to why [input check was done] redundantly and gave up trying to figure out. I removed the [redundant check] later on as it made more sense [to me]."*

8.3. Our vision: how should practitioners document the rationale of code commits?

We do not believe that developers should necessarily document every component of rationale all of the time. In fact, our participants mentioned concern about such an approach.

> *"I know it might not be doable or possible because no one will ever answer all these in a commit. However, it is a good model."*

Instead, the goal of our model is to provide a superset of the possible components that could answer a question about the rationale of code changes. Developers would then choose which components are relevant for which code change. We believe that developers may seek different components of rationale at different times, but not necessarily all of them every time. In fact, we observed that they seek different components with different frequencies (see Figure 5A-i).

We see our decomposition of the rationale of code commits as an artifact to support developers in documenting it (by reminding them of all the components they may want to document), not as a template that they would always be forced to fill completely. We believe that different components will be relevant for different types of code changes, and developers should judge which ones are worth documenting on each occasion. However, having our model as a *checklist* to review while documenting the rationale of code changes can be very useful for developers to ensure that all the relevant ones are documented. A similar approach was successfully applied in the area of bug reports [21, 11], not only to assist the documentation of various components but also to measure their quality. Checklists are a known powerful mechanism to ensure processes are performed correctly [24].

Another measure that would benefit developers is the *lazy* documentation of the rationale of code changes — *i.e.,* to document it opportunistically after they have needed it. Our observations in RQ6 show that developers often give up seeking the rationale of code changes due to not finding it in the code or documentation. Other times, they do not find it in code or documentation, but end up finding it in different ways, *e.g.,* asking a colleague:

> *"If I cannot figure out, I ask someone to help me out to understand the code because I need to work."*

We believe that this signals an opportunity: if developers document the rationale of code changes in a centralized documentation (maybe in the code comments themselves) after identifying it by other means, the next developer will easily find it when searching. This incremental approach to documenting may be preferred by many developers since the workload may feel more manageable that way, *e.g.,* as is the case with other practices like incremental testing [47, 46].

Developer teams are diverse [29], and some developers may want to improve the documentation of rationale of their past code commits in a more *exhaustive* fashion, as opposed to opportunistically. Some developers may also want to improve the current documentation of rationale of their past code commits, in a more *exhaustive* fashion —as opposed to opportunistically. Our observations also provide feedback on how to prioritize such efforts. A starting point for improving the rationale documentation of existing commits would be tackling the areas of improvement that we identified studying RQ5 and RQ7. We observed which components of the rationale of code commits are frequently not recorded (RQ5, see Figure 5C and RQ7, see Figure 7B), not found (RQ5, see

Figures 5B-i and RQ7, see Figure 7A), or difficult to find (RQ5, see Figure 5B-ii).

8.4. Our vision: how could tools support developers in documenting the rationale of code commits?

Our observations provide some advice on designing future support for developers to find the rationale of code commits. We observed that the need for the rationale of code commits has a wide reach: it affects most software development tasks and subtasks (RQ1, Figure 3A). Therefore, any future support should be accessible and convenient, independent of the developer's task.

We also observed that the rationale of code changes is needed for code changes in software projects both within the developer's company and external to it (RQ2, Figure 3). This teaches us that the rationale of code changes should be documented by also keeping in mind developers that do not necessarily belong to the software development team. Some examples of additional considerations that developers could have to be inclusive of external developers in their documentation efforts are: including clarifications about vocabulary that is specific to the software project (or adding links to a centralized legend); including assumptions that may be clear to the development team, but not to outsiders (*e.g.,* assumptions about the pre-set configuration of the OS or underlying libraries); documenting best practices that are common within the development team, but that may not be intuitive to outsiders (*e.g.,* unconventional code writing habits, such as special casing, indentation, exception handling, logging, etc.).

Our observations of why developers give up their search for rationale also inform how future techniques can be designed. We observed that the major factors why developers gave up were: (1) the poor state of the code and documentation, or (2) to avoid impacting the productivity of their team, or (3) their own. Therefore, it would be beneficial if future efforts to support the documentation of the rationale of code commits address these issues.

In particular, we believe that future efforts should support (1) recording the rationale of code commits once it is found. For example, allowing the edit of code commits (or other artifacts where rationale could be recorded) after they are created, so that if their rationale was missing, it could be added after it is sought and identified. We also believe that future efforts should support (2 & 3) asynchronous communication mechanisms, so that the productivity of the seeker and the people eventually providing the rationale can be impacted as little as possible.

A future centralized tool that supports asynchronous discussion of rationale for a code change (similar to the asynchronous discussion provided by code review tools for assessing code changes) would enable efficient communication among developers, while also helping rationale documentation. As code-review tools have shown, channeling and centralizing common conversations about code changes (in this case, it would be about rationale) can positively impact developer teams. Our participants also echoed the benefit of connecting software development artifacts that are related, so that their later comprehension is made easier:

> *"At my company, typically we link each commit, branch, and pull request (PR) to a ticket number, which should include details and discussion about the change and why it is being made."*

They also commented on the opposite: the difficulties they face when these centralized connections break.

> *"What usually makes the search more difficult is when code has been moved around, and the version control system lost track of its origin."*

Such a centralized system to support discussions about the rationale of code commits could provide additional benefits. First, it could be useful for developers to signal and keep track of code changes that need better documentation — which could be better documented at any pace that suits practitioners (supporting the *lazy* documentation that we discussed in Section 8.3). This practice of signaling improvement needed can be especially useful before somebody leaves the company — they could be requested to document all the code changes that they performed and need better documentation. This could contribute to reducing the problem of not being able to find rationale that can only be provided by somebody who has already left the team (as we observed in RQ6).

Having a tool to support conversations about the rationale of code commits would make it easier to document rationale after the discussions have happened — because the conversations themselves could help and simplify the process of producing structured documentation (as in bug-tracking systems). Such a system could also make it easier to save the documentation in the right place, connected to the code change discussed.

Finally, such a tool could also make it easier to find the rationale of code changes that have already been documented. If we had a system that kept track of the conversations about and documentation of the rationale of code changes, it would be easy to offer that information in easy-to-access ways in software development environments. The retrieved rationale of code changes could also be represented in different ways to fit the different tasks in which developers need to find it (RQ1) — similar to how the Whyline system provides already-available program-slicing information in a much more user-friendly format [48]. Such varied representations may also require varied analyses, *e.g.,* rationale finding triggered by a debugging task might require in-depth navigation of the history of relevant code changes (*e.g.,* [69, 70, 71, 72, 68]), whereas rationale finding triggered by learning about an external tool might require a broad search for similar codes across the external codebase. Finally, these efforts should provide the right amount of information to developers, since giving too many recommendations can have hidden costs [44].

An example of such an enhancement of an IDE would be to provide the rationale of code changes while the user is

actively debugging a feature. The IDE plugin could gather rationale components from previous commits, where the specific feature under debugging was changed. One of our participants commented on the importance of understanding the evolution of code (*i.e.,* "why the code is this way?") while debugging:

> *"I believe that understanding the evolution of the code is just as important as understanding the current code. If you know where the code has been, you can get a sense of where it needs to evolve for the next release and to be able to avoid the pitfalls of past bugs."*

8.5. Our vision: how could the documentation of the rationale of code commits be automated?

The previous subsection describes how future software tools could support developers in the process of documenting and finding the rationale of code commits. Next, we discuss how some parts of that process could be more strongly automated.

First, we see opportunities to automate the detection of insufficiently documented rationale of code changes. This could be either code changes completely lacking the documentation, or lacking relevant components. Automating this problem would be akin to predicting which code changes developers are likely to face the need to understand rationale components that are not currently documented. Such a tool could base its predictions on observing which characteristics are common among the code changes for which certain components are requested. We believe that this tool would be really valuable for developers to prioritize their efforts of documenting the rationale of code commits *i.e.,* starting with those for which their rationale is likely to be requested.

Next, we also see opportunities for automating the documentation of the rationale of code changes. Similarly, techniques could be developed to capture the common characteristics among the code changes with a specific answer to a given component of the rationale of code changes. If such common characteristics could be captured, future code changes with similar characteristics could be automatically labeled with the same answer to the rationale component. Similar mechanisms have been successfully applied to similar goals, *e.g.,* applying machine learning to detect design patterns [9]. Other approaches may also be promising, such as applying techniques for summarizing and documenting code changes like [19, 17], tools answering "what" and "why" questions about code changes like [41, 42, 63, 48, 13], and studies of impact and risk of changes like [82, 43, 65].

Any efforts to automate or support the process of documenting or finding the rationale of code commits will benefit from the rich understanding provided by our model (RQ2) of the specific pieces of information (components) that developers may seek when they need it. Our model now allows future research efforts to generate *targeted* pieces of information to generate to improve the documentation of rationale.

8.6. Our vision: what other benefits could arise from good documentation of the rationale of code commits?

In a longer timeline, we also anticipate further benefits from having a codebase in which the rationale of code commits is carefully documented. In projects with well-crafted documentation of the rationale of code commits, many future useful analyses may be possible. Next are some of the kinds of analyses that we anticipate.

First, we expect that the documentation of the rationale for code commits may be useful data to improve the traceability of software requirements to areas of the source code. This is a well-known problem and research area, and having richer data may help improve the accuracy of existing techniques to infer traceability or inspire future ones.

Second, the documentation of rationale for code commits may reveal hidden properties of the system, such as dependencies among different program parts that are otherwise not visible, assumptions that are not documented elsewhere, or development team habits that are otherwise not visible. Furthermore, the documentation of the rationale of code changes in highly successful software projects, *i.e.,* those with high quality, may reveal best practices that are not necessarily documented (are implicit in their decision-making), and from which other practitioners could learn.

The first step to studying the promise of these (or other) ideas for useful software analytics would start by identifying a current software project that already assigns high importance to documenting their code changes.

9. Threats to validity

9.1. Construct

To answer our research questions, we asked both open and quantitative questions. We scheduled the interview sessions to be relatively long (two hours), ensuring that we gave the participants enough time to express their ideas and share their thoughts. At the beginning of each interview section, we asked the participants to *"answer the questions in [their] own words and provide as much detail as [they] feel is relevant to address each question"*. We also placed an open question at the end of the interview to allow the participants to share any additional information about the topic.

9.2. Internal

The methods we used in our study, interviews and surveys, can be affected by bias and inaccurate responses. This effect could be intentional or unintentional. We gave gift cards to the interview participants and some survey participants, which could have biased our results. We indicated that the compensation is for the time spent and not the answers given to mitigate these concerns. We repeatedly and constantly used phrases to encourage the participants to provide their own honest opinions, using the phrase *"based on your experience"* in most of the questions. We also clearly indicated that the participants should *"feel free to change/add/delete components or not."* Sometimes, we also

indicated that *"there is no right or wrong answer; we are interested in what you think and your perspective."*

We also took multiple steps to reduce potential confirmation bias [60] resulting from using a preliminary model. We asked participants to describe their own examples and decomposition of rationale into components before they ever saw the preliminary model. We formed the preliminary model based on knowledge from the research literature and presented it neutrally. The fact that the preliminary was largely extended from 9 components into 15 validates that potential confirmation bias was minimal in our study.

Another threat to validity in our study is drawing conclusions based on recollected memories [50]. We are interested in capturing developers' opinions about what components constitute rationale, independently of how accurate their recollection is. We encouraged participants to take their time to recall situations and to report the components that mattered in their experience.

9.3. External

Our studied developers may not fully represent the whole developer population. To mitigate this threat, we recruited a diverse population with diverse types and amounts of experience (Figure 2). Our studied population was similar to the ones previously studied in the literature since we obtained similar answers for our two questions about rationale that were already studied by *Tao et al.* [79]. Furthermore, we have reached saturation for qualitative analysis of our open-ended interview questions (RQ1 and RQ6).

10. Conclusion

Developers invest valuable time and resources in the process of discovering the rationale of code commits, which they perform frequently and is difficult. However, any efforts aiming to improve this process will necessarily require a good understanding of the tasks for which rationale is needed and the specific pieces of information that developers seek when they search for rationale of code commits.

We applied a mixed-methods approach in this study. First, we performed a series of interviews with software developers to discover the components into which developers decompose the rationale of code commits, the tasks for which rationale for code commits needed, software developers' experiences (needing, finding, and recording) rationale of code commits and its components, and the factors leading developers to give up their search for rationale. Then, we ran a survey to better understand developers' experiences with rationale of code commits.

We found that software developers decompose rationale of code commits into 15 components along four themes. We also found that they need rationale for code commits to complete various software development tasks. We discovered that software developers have different experiences with different components. Developers need to find most components with similar frequency. However, they mostly only record and find those components automatically recorded by revision control systems. We also discovered the factors leading software developers to give up their search for rationale of code commits. This finding suggests that there is space for both researchers and practitioners to improve the practices of managing the rationale of code commits.

This work provides a detailed study of rationale for code commits. Our decomposition of rationale for code commits is a descriptive representation that practitioners can use to improve their documentation and communication of rationale. Additionally, researchers and tool builders can support the management of the rationale of code commits using our discovered components of rationale and the experiences of software developers with them.

11. Research Artifacts

An artifact containing our interview questions and survey instrument is available at https://doi.org/10.5281/zenodo.5941775.

12. Acknowledgements

This material is based upon work supported by Universidad Rey Juan Carlos under an International Distinguished Researcher award C01INVESDIST.

References

[1] Adolph, S., Hall, W., Kruchten, P., 2011. Using grounded theory to study the experience of software development. Empirical Software Engineering 16, 487–513. URL: https://doi.org/10.1007/s10664-010-9152-6, doi:10.1007/s10664-010-9152-6.

[2] Aghajani, E., Nagy, C., Vega-Márquez, O.L., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., 2019. Software documentation issues unveiled, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1199–1210. doi:10.1109/ICSE.2019.00122.

[3] Alkadhi, R., Johanssen, J.O., Guzman, E., Bruegge, B., 2017a. React: An approach for capturing rationale in chat messages, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 175–180. doi:10.1109/ESEM.2017.26.

[4] Alkadhi, R., Lata, T., Guzmany, E., Bruegge, B., 2017b. Rationale in development chat messages: An exploratory study, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 436–446. doi:10.1109/MSR.2017.43.

[5] Alkadhi, R., Nonnenmacher, M., Guzman, E., Bruegge, B., 2018. How do developers discuss rationale?, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 357–369. doi:10.1109/SANER.2018.8330223.

[6] Amyot, D., 2003. Introduction to the user requirements notation: learning by example. Computer Networks 42, 285 – 301. URL: http://www.sciencedirect.com/science/article/pii/S1389128603002445, doi:https://doi.org/10.1016/S1389-1286(03)00244-5. iTU-T System Design Languages (SDL).

[7] Anton, A.I., 1996. Goal-based requirements analysis, in: Proceedings of the Second International Conference on Requirements Engineering, pp. 136–144. doi:10.1109/ICRE.1996.491438.

[8] Aranda, J., Venolia, G., 2009. The secret life of bugs: Going past the errors and omissions in software repositories, in: 2009 IEEE 31st International Conference on Software Engineering, pp. 298–308. doi:10.1109/ICSE.2009.5070530.

[9] Barbudo, R., Ramírez, A., Servant, F., Romero, J.R., 2021. GEML: A grammar-based evolutionary machine learning approach for design-pattern detection. J. Syst. Softw. 175, 110919. URL: https://doi.org/10.1016/j.jss.2021.110919, doi:10.1016/j.jss.2021.110919.

[10] Begel, A., Simon, B., 2008. Novice software developers, all over again, in: Proceedings of the Fourth International Workshop on Computing Education Research, Association for Computing Machinery, New York, NY, USA. p. 3–14. URL: https://doi.org/10.1145/1404520.1404522, doi:10.1145/1404520.1404522.

[11] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., Zimmermann, T., 2008. What makes a good bug report?, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA. pp. 308–318. URL: http://doi.acm.org/10.1145/1453101.1453146, doi:10.1145/1453101.1453146.

[12] Biernacki, P., Waldorf, D., 1981. Snowball sampling: Problems and techniques of chain referral sampling. Sociological Methods & Research 10, 141–163. URL: https://doi.org/10.1177/004912418101000205, doi:10.1177/004912418101000205, arXiv:https://doi.org/10.1177/004912418101000205.

[13] Bradley, A.W., Murphy, G.C., 2011. Supporting software history exploration, in: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, New York, NY, USA. pp. 193–202. URL: http://doi.acm.org/10.1145/1985441.1985469, doi:10.1145/1985441.1985469.

[14] Breu, S., Premraj, R., Sillito, J., Zimmermann, T., 2010. Information needs in bug reports: Improving cooperation between developers and users, in: Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, Association for Computing Machinery, New York, NY, USA. p. 301–310. URL: https://doi.org/10.1145/1718918.1718973, doi:10.1145/1718918.1718973.

[15] Burge, J.E., Brown, D.C., 2008. Software engineering using rationale. Journal of Systems and Software 81, 395 – 413. URL: http://www.sciencedirect.com/science/article/pii/S0164121207001203, doi:https://doi.org/10.1016/j.jss.2007.05.004.

[16] Burge, J.E., Carroll, J.M., McCall, R., Mistrik, I., 2008. Rationale-Based Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg. URL: https://doi.org/10.1007/978-3-540-77583-6, doi:10.1007/978-3-540-77583-6.

[17] Buse, R.P., Weimer, W.R., 2010. Automatically documenting program changes, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, NY, USA. pp. 33–42. URL: http://doi.acm.org/10.1145/1858996.1859005, doi:10.1145/1858996.1859005.

[18] Codoban, M., Ragavan, S.S., Dig, D., Bailey, B., 2015. 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), pp. 1–10. doi:10.1109/ICSM.2015.7332446.

[19] Cortés-Coy, L.F., Linares-Vásquez, M., Aponte, J., Poshyvanyk, D., 2014. On automatically generating commit messages via summarization of source code changes, in: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp. 275–284. doi:10.1109/SCAM.2014.14.

[20] Dagenais, B., Ossher, H., Bellamy, R.K.E., Robillard, M.P., de Vries, J.P., 2010. Moving into a new software project landscape, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, Association for Computing Machinery, New York, NY, USA. p. 275–284. URL: https://doi.org/10.1145/1806799.1806842, doi:10.1145/1806799.1806842.

[21] Davies, S., Roper, M., 2014. What's in a bug report?, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ACM, New York, NY, USA. pp. 26:1–26:10. URL: http://doi.acm.org/10.1145/2652524.2652541, doi:10.1145/2652524.2652541.

[22] Dias, M., Bacchelli, A., Gousios, G., Cassou, D., Ducasse, S., 2015. Untangling fine-grained code changes, in: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 341–350. doi:10.1109/SANER.2015.7081844.

[23] Dutoit, A.H., McCall, R., Mistrík, I., Paech, B., 2006. Rationale Management in Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg. URL: https://doi.org/10.1007/978-3-540-30998-7, doi:10.1007/978-3-540-30998-7.

[24] Dziekan, G., 2008. Checklists save lives. Bulletin of the World Health Organization 86.

[25] Ebert, F., Castor, F., Novielli, N., Serebrenik, A., 2018. Communicative intention in code review questions, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 519–523. doi:10.1109/ICSME.2018.00061.

[26] Ebert, F., Castor, F., Novielli, N., Serebrenik, A., 2019. Confusion in code reviews: Reasons, impacts, and coping strategies, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 49–60. doi:10.1109/SANER.2019.8668024.

[27] Fritz, T., Murphy, G.C., 2010. Using information fragments to answer the questions developers ask, in: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM, New York, NY, USA. pp. 175–184. URL: http://doi.acm.org/10.1145/1806799.1806828, doi:10.1145/1806799.1806828.

[28] G Jelihovschi, E., Faria, J., 2014. Scottknott: A package for performing the scott-knott clustering algorithm in r. TEMA (São Carlos) 15. doi:10.5540/tema.2014.015.01.0003.

[29] Gautam, A., Vishwasrao, S., Servant, F., 2017. An empirical study of activity, popularity, size, testing, and stability in continuous integration, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 495–498. doi:10.1109/MSR.2017.38.

[30] Gilson, F., Englebert, V., 2011. Rationale, decisions and alternatives traceability for architecture design, in: Proceedings of the 5th European Conference on Software Architecture: Companion Volume, ACM, New York, NY, USA. pp. 4:1–4:9. URL: http://doi.acm.org/10.1145/2031759.2031764, doi:10.1145/2031759.2031764.

[31] Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P., 2017a. On the unhappiness of software developers, in: Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 324–333. URL: https://doi.org/10.1145/3084226.3084242, doi:10.1145/3084226.3084242.

[32] Graziotin, D., Fagerholm, F., Wang, X., Abrahamsson, P., 2017b. Unhappy developers: Bad for themselves, bad for process, and bad for software product, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 362–364. doi:10.1109/ICSE-C.2017.104.

[33] Gruber, T.R., Russell, D.M., 1991. Design knowledge and design rationale : A framework for representation , capture , and use.

[34] Gruber, T.R., Russell, D.M., 1996. Design rationale, L. Erlbaum Associates Inc., Hillsdale, NJ, USA. chapter Generative Design Rationale: Beyond the Record and Replay Paradigm, pp. 323–349. URL: http://dl.acm.org/citation.cfm?id=261685.261725.

[35] Hennink, M.M., Kaiser, B.N., 2019. Saturation in qualitative research. URL: http://dx.doi.org/10.4135/9781526421036822322.

[36] Herzig, K., Zeller, A., 2011. Untangling changes .

[37] Herzig, K., Zeller, A., 2013. The impact of tangled code changes, in: 2013 10th Working Conference on Mining Software Repositories (MSR), pp. 121–130. doi:10.1109/MSR.2013.6624018.

[38] Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D., 2017. Trade-offs in continuous integration: Assurance, security, and flexibility, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA. pp. 197–207. URL: http://doi.acm.org/10.1145/3106237.3106270, doi:10.1145/3106237.3106270.

[39] ITU-T, 2018. User requirements notation (urn) – language definition. URL: http://handle.itu.int/11.1002/1000/13711.

[40] Jarczyk, A.P.J., Loffler, P., Shipmann, F.M., 1992. Design rationale for software engineering: A survey, in: Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, pp. 577–586 vol.2. doi:10.1109/HICSS.1992.183309.

[41] Jiang, S., Armaly, A., McMillan, C., 2017a. Automatically generating commit messages from diffs using neural machine translation, in: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, Piscataway, NJ, USA. pp. 135–146. URL: http://dl.acm.org/citation.cfm?id=3155562.3155583.

[42] Jiang, S., McMillan, C., 2017. Towards automatic generation of short summaries of commits, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), pp. 320–323. doi:10.1109/ICPC.2017.12.

[43] Jiang, S., McMillan, C., Santelices, R., 2017b. Do programmers do change impact analysis in debugging? Empirical Software Engineering 22, 631–669. URL: https://doi.org/10.1007/s10664-016-9441-9, doi:10.1007/s10664-016-9441-9.

[44] Jin, X., Servant, F., 2018. The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency, in: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp. 70–73.

[45] Kaiya, H., Horai, H., Saeki, M., 2002. Agora: Attributed goal-oriented requirements analysis method, in: Proceedings IEEE Joint International Conference on Requirements Engineering, pp. 13–22. doi:10.1109/ICRE.2002.1048501.

[46] Kazerouni, A.M., Davis, J.C., Basak, A., Shaffer, C.A., Servant, F., Edwards, S.H., 2021. Fast and accurate incremental feedback for students' software tests using selective mutation analysis. J. Syst. Softw. 175, 110905. URL: https://doi.org/10.1016/j.jss.2021.110905, doi:10.1016/j.jss.2021.110905.

[47] Kazerouni, A.M., Shaffer, C.A., Edwards, S.H., Servant, F., 2019. Assessing incremental testing practices and their impact on project outcomes, in: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Association for Computing Machinery, New York, NY, USA. p. 407–413. URL: https://doi.org/10.1145/3287324.3287366, doi:10.1145/3287324.3287366.

[48] Ko, A., Myers, B., 2008. Debugging reinvented, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, pp. 301–310. doi:10.1145/1368088.1368130.

[49] Ko, A.J., DeLine, R., Venolia, G., 2007. Information needs in collocated software development teams, in: 29th International Conference on Software Engineering (ICSE'07), pp. 344–353. doi:10.1109/ICSE.2007.45.

[50] Koriat, A., Goldsmith, M., Pansky, A., 2000. Toward a psychology of memory accuracy. Annual Review of Psychology 51, 481–537. URL: https://doi.org/10.1146/annurev.psych.51.1.481, doi:10.1146/annurev.psych.51.1.481, arXiv:https://doi.org/10.1146/annurev.psych.51.1.481. pMID: 10751979.

[51] Kunz, W., Rittel, H., 1970. Issues as Elements of Information Systems. Number 131 in California. University. Center for Planning and Development Research. Working paper, no. 131, Institute of Urban and Regional Development, University of California. URL: https://books.google.com/books?id=B-MaAQAAMAAJ.

[52] Kurtanović, Z., Maalej, W., 2017. Mining user rationale from software reviews, in: 2017 IEEE 25th International Requirements Engineering Conference (RE), pp. 61–70. doi:10.1109/RE.2017.86.

[53] L BERG, B., 2001. Qualitative research methods for the social sciences .

[54] Lamsweerde, A.V., 2001. Goal-oriented requirements engineering: A guided tour, in: Proceedings Fifth IEEE International Symposium on Requirements Engineering, pp. 249–262. doi:10.1109/ISRE.2001.948567.

[55] LaToza, T.D., Myers, B.A., 2010. Hard-to-answer questions about code, in: Evaluation and Usability of Programming Languages and Tools, ACM, New York, NY, USA. pp. 8:1–8:6. URL: http://doi.acm.org/10.1145/1937117.1937125, doi:10.1145/1937117.1937125.

[56] Lee, J., Lai, K.Y., 1991. What's in design rationale? Hum.-Comput. Interact. 6, 251–280. URL: http://dx.doi.org/10.1207/s15327051hci0603&4_3, doi:10.1207/s15327051hci0603\&4_3.

[57] Maalej, W., Tiarks, R., Roehm, T., Koschke, R., 2014. On the comprehension of program comprehension. ACM Trans. Softw. Eng. Methodol. 23, 31:1–31:37. URL: http://doi.acm.org/10.1145/2622669, doi:10.1145/2622669.

[58] Murgia, A., Tourani, P., Adams, B., Ortu, M., 2014. Do developers feel emotions? an exploratory analysis of emotions in software artifacts, in: Proceedings of the 11th Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA. p. 262–271. URL: https://doi.org/10.1145/2597073.2597086, doi:10.1145/2597073.2597086.

[59] Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., Bacchelli, A., 2018. Information needs in contemporary code review. Proc. ACM Hum.-Comput. Interact. 2, 135:1–135:27. URL: http://doi.acm.org/10.1145/3274404, doi:10.1145/3274404.

[60] Pohl, R., Pohl, R., 2004. Confirmation bias, in: Cognitive Illusions: A Handbook on Fallacies and Biases in Thinking, Judgement and Memory. Psychology Press. chapter 4, pp. 79–96. URL: https://books.google.com/books?id=k5gTes7yyWEC.

[61] Potts, C., Bruns, G., 1988. Recording the reasons for design decisions, in: Proceedings of the 10th International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, USA. pp. 418–427. URL: http://dl.acm.org/citation.cfm?id=55823.55863.

[62] Ram, A., Sawant, A.A., Castelluccio, M., Bacchelli, A., 2018. What makes a code change easier to review: An empirical investigation on code change reviewability, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 201–212. URL: https://doi.org/10.1145/3236024.3236080, doi:10.1145/3236024.3236080.

[63] Rastkar, S., Murphy, G.C., 2013. Why did this code change?, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA. pp. 1193–1196. URL: http://dl.acm.org/citation.cfm?id=2486788.2486959.

[64] Roehm, T., Tiarks, R., Koschke, R., Maalej, W., 2012. How do professional developers comprehend software?, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA. pp. 255–265. URL: http://dl.acm.org/citation.cfm?id=2337223.2337254.

[65] Rosen, C., Grawi, B., Shihab, E., 2015. Commit guru: Analytics and risk prediction of software commits, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA. pp. 966–969. URL: http://doi.acm.org/10.1145/2786805.2803183, doi:10.1145/2786805.2803183.

[66] Safwan, K.A., Servant, F., 2019. Decomposing the rationale of code commits: The software developer's perspective, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 397–408. URL: https://doi.org/10.1145/3338906.3338979, doi:10.1145/3338906.3338979.

[67] Saunders, B., Sim, J., Kingstone, T., Baker, S., Waterfield, J., Bartlam, B., Burroughs, H., Jinks, C., 2018. Saturation in qualitative research: exploring its conceptualization and operationalization. Quality & Quantity 52, 1893–1907. URL: https://doi.org/10.1007/s11135-017-0574-8, doi:10.1007/s11135-017-0574-8.

[68] Servant, F., 2013. Supporting bug investigation using history analysis, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 754–757. doi:10.1109/ASE.2013.6693150.

[69] Servant, F., Jones, J.A., 2011. History slicing, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), pp. 452–455. doi:10.1109/ASE.2011.6100097.

[70] Servant, F., Jones, J.A., 2012. History slicing: Assisting code-evolution tasks, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/2393596.2393646, doi:10.1145/2393596.2393646.

[71] Servant, F., Jones, J.A., 2013. Chronos: Visualizing slices of source-code history, in: 2013 First IEEE Working Conference on Software Visualization (VISSOFT), pp. 1–4. doi:10.1109/VISSOFT.2013.6650547.

[72] Servant, F., Jones, J.A., 2017. Fuzzy fine-grained code-history analysis, in: Proceedings of the 39th International Conference on Software Engineering, IEEE Press. p. 746–757. URL: https://doi.

org/10.1109/ICSE.2017.74, doi:10.1109/ICSE.2017.74.

[73] Singer, J., Sim, S.E., Lethbridge, T.C., 2008. Guide to Advanced Empirical Software Engineering. Springer London, London. URL: https://doi.org/10.1007/978-1-84800-044-5, doi:10.1007/978-1-84800-044-5.

[74] Spadini, D., Aniche, M., Storey, M.A., Bruntink, M., Bacchelli, A., 2018. When testing meets code review: Why and how developers review tests, in: Proceedings of the 40th International Conference on Software Engineering, ACM, New York, NY, USA. pp. 677–687. URL: http://doi.acm.org/10.1145/3180155.3180192, doi:10.1145/3180155.3180192.

[75] Spencer, D., Garrett, J., 2009. Card Sorting: Designing Usable Categories. Rosenfeld Media. URL: https://books.google.com/books?id=_h4D9gqi5tsC.

[76] Srinivasa Ragavan, S., Codoban, M., Piorkowski, D., Dig, D., Burnett, M., 2019. Version control systems: An information foraging perspective. IEEE Transactions on Software Engineering , 1–1doi:10.1109/TSE.2019.2931296.

[77] Tang, A., Babar, M.A., Gorton, I., Han, J., 2006. A survey of architecture design rationale. Journal of Systems and Software 79, 1792 – 1804. URL: http://www.sciencedirect.com/science/article/pii/S0164121206001415, doi:https://doi.org/10.1016/j.jss.2006.04.029.

[78] Tang, A., Jin, Y., Han, J., 2007. A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software 80, 918 – 934. URL: http://www.sciencedirect.com/science/article/pii/S0164121206002287, doi:https://doi.org/10.1016/j.jss.2006.08.040.

[79] Tao, Y., Dang, Y., Xie, T., Zhang, D., Kim, S., 2012. How do software engineers understand code changes?: An exploratory study in industry, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA. pp. 51:1–51:11. URL: http://doi.acm.org/10.1145/2393596.2393656, doi:10.1145/2393596.2393656.

[80] Toulmin, S.E., 2003. The Uses of Argument. 2 ed., Cambridge University Press. doi:10.1017/CBO9780511840005.

[81] Tyree, J., Akerman, A., 2005. Architecture decisions: demystifying architecture. IEEE Software 22, 19–27. doi:10.1109/MS.2005.27.

[82] Zhang, S., Ernst, M.D., 2014. Which configuration option should i change?, in: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA. pp. 152–163. URL: http://doi.acm.org/10.1145/2568225.2568251, doi:10.1145/2568225.2568251.

Khadijah Al Safwan is a Ph.D. candidate working with Dr. Francisco Servant at the Department of Computer Science, Virginia Tech. She earned her M.S. in ComputerScience from Virginia Tech in 2018, and her B.S. in Computer Science from the University of Dammam (now Imam Abdulrahman Bin Faisal University) in Dammam, Saudi Arabia in 2015. Her interests include software development in practice, empirical software engineering, and software changes. Find her on the web at http://khsafwan.com/.

Mohammed ElArnaoty is a Ph.D. student working with Dr. Francisco Servant at the Department of Computer Science, Virginia Tech. He earned her M.S. in ComputerScience and his B.S. from Cairo University, Egypt. His interests include empirical software engineering, Applied machine learning in software engineering development in practice, and automated software.

Francisco Servant is a Distinguished Researcher at Universidad Rey Juan Carlos in Madrid, Spain. Before, he was an Assistant Professor in the Department of Computer Science at Virginia Tech. He received a Ph.D. in Software Engineering from the University of California, Irvine, and a B.S. in Computer Science from the University of Granada, Spain. His research focuses on software development productivity and software quality. Find him on the web at https://fservant.com/.