Practitioners' Expectations on Automated Code Comment Generation

Xing Hu
School of Software Technology,
Zhejiang University
Ningbo, China
xinghu@zju.edu.cn

Xin Xia*
Zhejiang University
Hangzhou, China
xin.xia@acm.org

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Zhiyuan Wan
Zhejiang University
Hangzhou, China
wanzhiyuan@zju.edu.cn

Qiuyuan Chen
Zhejiang University
Hangzhou, China
chenqiuyuan@zju.edu.cn

Thomas Zimmermann
Microsoft Research
Seattle, USA
tzimmer@microsoft.com

ABSTRACT

Good comments are invaluable assets to software projects, as they help developers understand and maintain projects. However, due to some poor commenting practices, comments are often missing or inconsistent with the source code. Software engineering practitioners often spend a significant amount of time and effort reading and understanding programs without or with poor comments. To counter this, researchers have proposed various techniques to automatically generate code comments in recent years, which can not only save developers time writing comments but also help them better understand existing software projects. However, it is unclear whether these techniques can alleviate comment issues and whether practitioners appreciate this line of research. To fill this gap, we performed an empirical study by interviewing and surveying practitioners about their expectations of research in code comment generation. We then compared what practitioners need and the current state-of-the-art research by performing a literature review of papers on code comment generation techniques published in the premier publication venues from 2010 to 2020. From this comparison, we highlighted the directions where researchers need to put effort to develop comment generation techniques that matter to practitioners.

KEYWORDS

Code Comment Generation, Empirical Study, Practitioners' Expectations

ACM Reference Format:
Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. 2022. Practitioners' Expectations on Automated Code Comment Generation. In *The 44th International Conference on Software Engineering, May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510152

*Corresponding Author.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510152

1 INTRODUCTION

Code comments are essential parts of software projects and provide descriptive information about the functionality, design rationale, and usage of a code snippet [4]. They help practitioners use, understand, and maintain software projects [45]. Well-commented source code improves project readability and developer productivity. Despite the intrinsic value of code comments during software development and evolution activities, the creation and maintenance of comments are often neglected. To address these issues, different approaches and tools have been proposed to generate comments from source code automatically [13, 17, 19, 21, 22, 49, 51, 52].

These techniques traditionally rely on manually crafted templates and information retrieval (IR) techniques to generate comments. Template-based approaches mainly rely on elaborate heuristics and templates for different types of programs to generate descriptive comments [32, 42]. However, defining a template requires substantial human effort and extensive domain knowledge. IR-based approaches mainly extract terms from source code and then organize these terms for generating comments [13, 17]. Besides, some studies generate comments by retrieving similar code snippets and using their corresponding comments for comment generation [51, 52]. In recent years, many researchers have taken advantage of deep learning techniques to generate comments by learning from large, publicly available code repositories [19, 21, 22, 49]. These techniques apply neural machine translation models to learn to translate source code to comments [16].

Despite numerous studies on code comment generation, unfortunately, few studies have investigated the expectations of practitioners on research in comment generation. It is unclear whether practitioners appreciate this line of research. Even if they do, it is unclear whether they would adopt code comment generation tools, what factors affect their decisions to adopt, and their minimum thresholds for adoption. The practitioners' perspective is important to help guide software engineering researchers to create solutions that satisfy developers. In addition, some gaps between practitioners' expectations and research have not yet been investigated.

To gain insights into practitioners' expectations on code comment generation, we first conducted semi-structured interviews

with 16 professionals from various companies. Through the interviews, we qualitatively investigated the commenting practices and issues that our interviewees experienced in software development, and their expectations on code comment generation. Then, we validated our findings through a survey answered by 720 professional developers or other IT professionals from 26 countries across six continents. After the survey, we performed a literature review of the state-of-the-art papers. We then compared techniques proposed in the papers against the criteria that practitioners have for adoption.

In particular, we investigated the following four research questions:

RQ1: What is the state of code commenting practices and what are the issues?

This research question studies code commenting practices and issues that practitioners experienced during software development. 82% and 81% of the survey respondents often write comments and are often confused when reading code without comments, respectively. Meanwhile, 69% and 62% respondents considered *lack of comments* and *generic comments* as the main issues, respectively.

RQ2: Are automated code comment generation tools useful for practitioners?

This research question investigates practitioners' willingness to adopt code comment generation techniques. 80% of the survey respondents think code comment generation tools are worthwhile and essential for them. 78% of them agree that these tools can help them understand the source code, especially for existing projects with fewer comments and improve code readability.

RQ3: What are practitioners' expectations on code comment generation tools?

This research question focuses on investigating what to comment and where to comment for different granularity level comments that practitioners expect, and what factors can affect their adoption of a comment generation technique. Most participants (about 85%) expect tools to generate method-level comments. The comments should include information about 1) what the method does (i.e., functionality); 2) how to use the method; and 3) why the method exists (i.e., design rationale). The most important locations to be commented on include complex, tricky, and non self-explanatory methods. The optimal length of a generated comment is 2-3 lines. Before adopting a code comment generation tool, the generated comments should satisfy the amount of additional information (i.e., amount of information beyond what can be easily gleaned from scanning the source code), content adequacy (the amount of content carried over from the input code to the generated comments, ignoring fluency of the text), and conciseness.

RQ4: How close are the current state-of-the-art studies to satisfy practitioner needs and demands before adoption?

This research question investigates the current state-of-the-art research and compares the gap between it and practitioners' expectations. We identified 25 papers that proposed code comment generation techniques and 17 of them generated method-level comments. Most papers generated comments to describe methods' functionality. However, few papers generated comments with "*how to use*" and "*why a method exists*" information. In addition, most papers focus on measuring the overlapped N-grams between generated comments and human-written comments, while it is not preferred by a large majority of our respondents. Also, no papers evaluate *the amount of additional information* in the generated comments, which most practitioners expect.

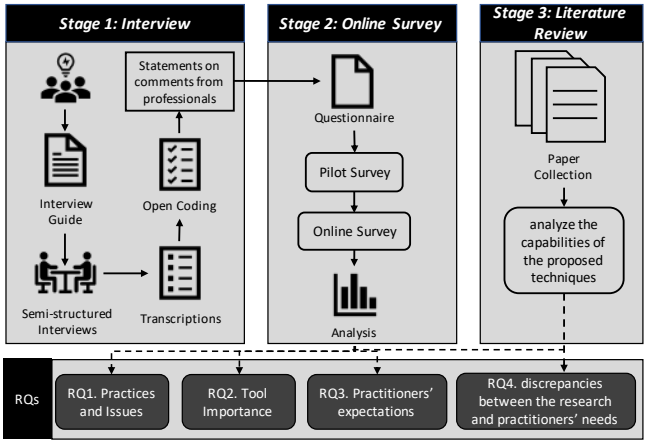


Figure 1: Research Methodology Overview

Our research is meant to help researchers to consider the needs of practitioners to continue the development of better code comment generation techniques that can eventually result in high adoption and satisfaction rate.

This paper makes the following contributions:

- We interviewed 16 professionals and surveyed 720 practitioners from more than 26 countries to shed light on practitioners' expectations, including their views on the importance of comment generation and their thresholds and reasons for adopting or not adopting such techniques.
- We performed a literature review of papers published in the premier publication venues in software engineering and artificial intelligence communities in the last ten years. Then, we compared the current state-of-research with what practitioners want and highlighted what can be done next to meet practitioners' needs and demands.

Paper Structure: Section 2 describes the methodology of our study. Section 3 shows the results of our study. We discuss the implications of our results in Section 4. Section 5 discusses related work. Section 6 draws conclusions and outlines avenues for future work.

2 RESEARCH METHODOLOGY

The overview of the methodology in our study is shown in Figure 1 and consists of three stages. ***Stage 1***: Interviews with professionals on their practices on commenting, issues they face related to code comments, and their expectations on code comment generation techniques. ***Stage 2***: An online survey for confirming and extending the conclusions about code comments based on the interview. ***Stage 3***: Perform a literature review to analyze whether and to what extent current state-of-the-art research has satisfied practitioners' needs and demands. The interviews and survey were approved by the relevant institutional review board (IRB).

2.1 Stage 1: Interview

The interview aims to understand commenting practices and issues that professionals experience during software development and practitioners' expectations on code comment generation tools. This section presents the interview process.

2.1.1 Protocol. The first author conducted a series of face-to-face semi-structured, in-depth interviews based on an interview guide to enable a detailed exploration of the participants' views and experiences. We developed the interview guide through a brainstorming process. We invited 16 software practitioners to participate in the interviews from 10 IT companies worldwide. Each interview took 30-40 minutes. In the remainder of the paper, we denoted these 16 interviewees as I1 to I16.

Each interview had three parts. In the first part, we asked some demographic questions about the interviewee's background (e.g., job role, length of work experience, and team size). In the second part, we asked open-ended questions about what they consider to be good/bad code comments. This part aimed to allow the interviewees to speak freely about their opinions and experience without the interviewer biasing their responses. In the third part, we asked the interviewees to discuss the commenting practices and issues that they faced related to code comments. We also asked about the importance of automated comment generation tools and their expectations on these tools.

At the end of each interview, we thanked interviewees and briefly informed them of our next plan.

2.1.2 Interviewees. We invited professionals from our networks in the software industry who were working full time in different roles (e.g., developers and architects) to participate in the interviews. We sent 20 formal invitations to invite potential interviewees, and 16 interviewees agreed to participate in the interviews from ten IT companies worldwide. These 16 interviewees had an average of 4.2 years of professional experience in software development (min: 1, max: 11, median: 4.2, sd: 2.5).

2.1.3 Data Analysis. The first author analyzed the interviews by transcribing them and then performed open coding to generate codes of the interview contents using NVivo qualitative analysis software [1]. Then, the second author verified the initial codes created by the first author and provided suggestions for improvement. After incorporating these suggestions, two authors separately analyzed the codes and sorted the generated cards into potential statements. The overall Cohen's Kappa value between the two authors was 0.78, which indicated substantial agreement between the them. The two authors discussed their disagreements to reach a common decision. To reduce bias from the two authors sorting the cards to form initial statements, they both reviewed and agreed on the final set of statements. Eventually, based on the results of the interviews, we derived 6 commenting practices, 6 commenting issues, 5 conclusions for tool importance, 12/17/14 conclusions for expectations on class/method/statement comment generation, and 11 factors that affect the adoption of code comment generation tools.

2.2 Stage 2: Online Survey

To confirm the statements made by the interviewees (i.e., Stage 1), we conducted an anonymous online survey with more participants. The survey aimed to validate and quantify the observations from our interviews.

2.2.1 Design. The survey included different types of questions, e.g., multiple-choice questions, short answer questions, and rating questions (in 5-point Likert scale: Strongly Disagree to Strongly Agree). We included the category "I don't understand" to filter respondents who do not understand our brief descriptions.

The survey consists of six sections:

- **Demographics:** The survey first asked for demographic information about the participants, including country/area of residence, primary job role, experience in years, and team size.
- **Commenting Practices:** This section investigated practitioners' commenting practices during software development, specifically, their practices on writing and reading comments, the commenting distributions in different projects, as well as the commenting review practices in practitioners' teams.
- **Commenting Issues:** This section focused on commenting issues that practitioners faced during software development, including outdated comments, too long comments, and redundant comments in projects.
- **Tool Importance:** This section provided respondents with a brief description of code comment generation tools and asked them how they perceive the importance of such line of tools with the following statements: (i) *Essential*: I will use this tool every day to help software development or code comprehension; (ii) *Worthwhile*: I will use this tool to help software development or code comprehension; (iii) *Unimportant*: I will not use this tool; (iv) *Unwise*: This tool will harm my or my team's productivity. Then, we asked practitioners about the importance aspects (e.g., improving development efficiency and code readability).
- **Practitioners' Expectations:** This section investigated practitioners' expectations on these tools, including preferred granularity levels (i.e., generating class-level, method-level, and statement-level comments). Then, we asked what information should be included in generated comments, the locations to be commented and preferred lengths for different level comments.
- **Tool Adoption:** This section asked respondents factors that affect their likelihood to adopt a code comment generation technique. Specifically, we asked the *minimum Turing Test rate* (the percentage of generated comments that are indistinguishable from human-written comments by a human evaluator), *maximum revised rate* (the percentage of the content in a generated comment that is needed to be revised before adoption), and *minimum efficiency* (the time of a tool to give a recommendation).

At the end of the survey, we allowed respondents to provide free-text comments, suggestions, and opinions about code commenting and our survey. A respondent may or may not provide any final comments.

We piloted the preliminary survey with a small set of practitioners who were different from our interviewees and survey takers. We obtained feedback on (1) whether the length of the survey was appropriate, and (2) the clarity and understandability of the terms. We made minor modifications to the preliminary survey based on the received feedback and produced a final version. Note that the collected responses from the pilot survey were excluded from the presented results in this paper.

To support respondents from China, we translated our survey to Chinese before distributing it to them. We chose to make our survey available both in English on Google Forms, and in Chinese on a popular survey website in China [3]. We chose to make our

Table 1: Participants roles & programming experience

Role	Population	<1 y	1-3 y	3-5 y	5-10 y	>10 y
Development	534	32	110	162	173	57
Testing	59	7	13	18	16	5
Algorithm/ML Model Design	55	9	16	15	10	5
Project manager	14	0	0	1	3	10
Architect	23	0	0	3	9	11
Others	35	4	11	10	5	5
	720	52	150	209	216	93

survey available in Chinese and English as the earlier is the most spoken language and the latter is an international lingua franca.

2.2.2 Participant Recruitment. We followed two steps to invite participants:

- We contacted professionals within our social network from IT companies and asked their help to disseminate our survey. Specifically, we sent invitations to our contacts in Tencent, Microsoft, Alibaba, Google, Huawei, and other companies, encouraging them to disseminate our survey to some of their colleagues. By following this strategy, we received 598 responses.
- We mined GitHub repositories to extract their contributors' public email addresses. Specifically, we sought repositories with the top popular open source projects (based on their number of stars). We sent emails to 2000 potential developers with a link to our survey. We aimed to recruit open-source practitioners who have software development experience in addition to professionals working in the industry. Out of these emails, we received eight automatic replies notifying us of the absence of the receiver; two receivers replied that they would not answer any survey. Finally, we received 137 responses.

In total, we received 735 survey responses. We discarded two incomplete surveys and 13 responses with less than two minutes of survey completion time. The data reported herein were from the remaining 720 valid responses. The 720 respondents resided in 26 countries across six continents. The top two countries where the respondents came from were China and United States. An overview of the surveyed participants and their experience was depicted in Table 1. Most participants were engaged in software development and had 3-5 years of professional experience.

2.2.3 Data Analysis. We analyzed the survey results based on the question types. For multiple-choice questions, we reported the percentage of each option is selected. In terms of open-ended questions, we analyzed the survey results qualitatively by inspecting responses. To understand trends in the Likert-scale questions, we created bar charts (many of which are shown in the remainder of this paper). We dropped "I don't know" ratings that form a small minority (about 1%) of all ratings.

Replication Package. The interview guide and questionnaire used to run our study are available in our replication package [2].

2.3 Stage 3: Literature Review

Research papers about code comment generation techniques are usually published in software engineering and artificial intelligence fields. Therefore, we went through full research papers published in ICSE, ESEC/FSE, ASE, ICPC, SANER, MSR, ICSME, TSE, TOSEM, EMSE, ACL, IJCAI, ICLR, NIPS, and AAAI from 2010 to 2020. We selected papers from the above conferences and journals as they are premier publication venues in software engineering and artificial intelligence research communities, and state-of-the-art findings are published in these conferences and journals.

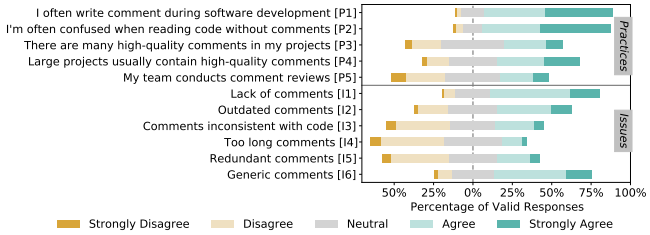


Figure 2: Comment practices and comment issues

Table 2: The agreement rate of statement [P1] and [P2] in terms of practitioners' experiences.

Statements \ Experiences	<1 y	1-3 y	3-5 y	5-10 y	>10 y
[P1][1]	0.82	0.80	0.83	0.82	0.80
[P2][2]	0.9	0.88	0.83	0.77	0.71

[1] [P1] I often write comments during software development
[2] [P2] I'm often confused when reading code without comments

We read the titles and abstracts of all papers and judged whether each of the papers proposes a new code comment generation technique that can help practitioners generate comments during software development. We included papers on IR-based code comment generation (e.g., [17]), template-based code comment generation (e.g., [42]), and deep-learning-based code comment generation (e.g., [22]). We excluded papers on other types of software documentation generation (e.g., commit message generation [23][28]), and empirical study on comment generation (e.g., [4]).

For each code comment generation paper, two authors read its content and analyzed the capabilities of the proposed technique in terms of the following factors: granularity level, what-to-comment, where-to-comment, and evaluation criteria, respectively. For example, Wei et al. [50] declared that they took the first sentence or line in JavaDoc as the output of their proposed approach, thus we classified its length as one line. If a paper did not declare the capabilities explicitly, the two authors checked the contents and discussed its capabilities. For example, Moreno et al. [32] proposed the *Factory stereotype* to generate comments for factory class; thus, we inferred that it satisfied the statement [C9], i.e., commenting at *Classes with design patterns*. Two authors discussed the differences in the capability analysis and confirmed the final result through further paper reading. Among the selected venues, we found no comment generation paper in MSR and ICSME. We will discuss the literature review results in Section 3.4.

3 RESULTS

We explain the results of research questions that investigate comment generation techniques from the perspective of practitioners.

3.1 RQ1: Commenting Practices and Issues

In RQ1, we explored commenting practices, including practitioners' practices on writing/reading comments during development, quantity and quality of comments in their projects, and the commenting review practices in their team. Besides, we also reported the main commenting issues that participants frequently face. Figure 2 illustrates respondents' rating of some statements related to commenting practices and issues.

3.1.1 Commenting Practices. **For developers**, we find that more than 82% participants often write code comments during software

development. Interestingly, almost the same proportion of participants (81%) are often confused when reading code without comments. Although most participants often write comments, the quantity of comments is still not enough for developers to read source code. Participants' opinions on writing and reading code comments are contradictory. As one participant in our survey stated: "*Everyone wants others to write as many comments as possible, but they don't want to write comments*". Table 2 illustrates the agreement rate (percentage of Agree or Strongly Agree) of statements [P1] and [P2] in terms of practitioners' experiences. We can find that there are no clear differences in writing comments for practitioners with different years of experience. However, junior practitioners are statistically significantly more confused when reading code without comments (with Mann-Whitney Wilcoxon Test p-value<0.001).

Considering software projects, only 37% participants indicate that there are many high-quality code comments in their projects. On one hand, the quantity of code comments was limited; on the other hand, lots of code comments had various issues making it difficult for participants to infer useful information from comments. In addition, 52% of participants agreed that larger projects usually contained much higher quality comments. These projects usually needed team cooperation and high-quality comments could help to improve collaborative development. In terms of projects, high-quality code comments were essential as participants stated in our survey: (1) "*Comments in projects are very useful to understand the code logic, especially indispensable for facilitating project handover.*" (2) "*Comments can facilitate project maintenance and fault location*". **Few teams** (30%) conducted comment reviews during the code review, even though comment quality was important for developers and software projects. Without comment review, issues in code comments cannot be captured in time. In addition, one participant pointed out the importance of comments on code review, as "*During the development process, it is important but not urgent, but it is necessary for code review.*"

For commenting practices, participants mainly have two attitudes:

- 👍 Comments are **necessary**. Most developers thought that comments were essential and should be provided along with the source code. A respondent shared her/his experience on working with a legacy system to support the effectiveness of the code comments. *"The system has a lot of comments in each program in which hundreds of lines of comments accompany several code lines. This system was introduced to Hong Kong in the 1980s and then moved to Guangzhou and Beijing. Thanks to the abundant (even redundant) comments, the legacy system is still maintainable being running for almost 40 years, and it is still the indispensable core system of a company."* The respondent expressed that their experience made them realized the benefit of writing proper code comments in the long term and made them optimistic about the future of the comment generation tools.
- 👎 Comments are **unnecessary**. A fraction of developers thought code comments were unnecessary and writing self-explanatory code was much more important. As a participant stated: "*Source code is the best comments. It is more important to write code that is easier to understand. Maintaining the source and comments at the same time is time-consuming and reduces efficiency.*"

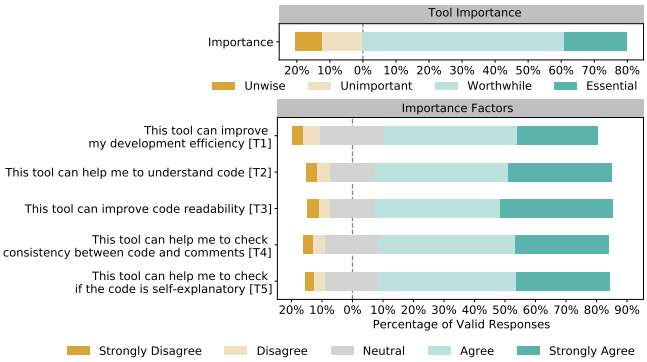


Figure 3: Tool importance and the importance factors

> **Finding 1.** Junior practitioners find it harder to read source code without comments. The quantity and quality of comments in software projects are limited and few teams conduct comment reviews.

3.1.2 Commenting Issues. Figure 2 showed respondents' ratings of comment-related issues they faced during software development. 69% and 62% respondents considered *lack of comments* and *generic comments* as the main issues, respectively. Without code comments, practitioners tend to read the source code and resorted to external sources of information to understand the source code [5]. One participant stressed the issue by using her/his personal experience: "*Without code comments, the source code is not only unreadable by others, but also may not be understood by myself after a period of time*". *Generic comment* was another issue during software development. Participants cannot get useful information from the code comments without more details. *Outdated comments* were perceived as a frequent issue by almost half (i.e., 47%) of the surveyed practitioners. This issue often occurs during software development. Developers may forget or ignore the updates of comments when changing source code [29]. *Inconsistency* between code and comment were also perceived as a frequent issue by practitioners (31%). 27% of the participants thought that *redundant comment* was a frequent issue. As one participant stated: "*The more noise, the harder it is to notice valuable information in comments. ... Redundant comments are far less intuitive than looking at the code...*" In terms of the length of code comments, only 16% of participants regarded too long comments as an issue. This issue may increase the time to understand the source code, as one participant stressed: "*Too long comments may affect the efficiency, and I prefer precise and concise comments.*"

> **Finding 2.** Lack of comments and generic comments that do not provide much information are the most frequently encountered issues.

3.2 RQ2: Tool Importance

Figure 3 illustrated the percentages of ratings of various categories (i.e., Essential, Worthwhile, Unimportant, Unwise) and importance factors. We could notice that most respondents (i.e., 80%) gave "*Essential*" and "*Worthwhile*" ratings. Around 18.5% of respondents rated comment generation tool as an "*Essential*" tool and would use it every day during software development.

We further investigated the factors that affected the importance of the code comment generation tools. All factors received similar agreement from participants. As shown in Figure 3, improving

code readability (78% *Agree* or *Strongly Agree*) was the most important factor among all importance factors. Comment generation tools could help to improve development efficiency, understand the source code, and improve code readability. As participants stated: (1) "*Automated comment generation tool can improve efficiency and productivity*"; (2) "*If the tool can automatically extract the meaning of the program, it will greatly improve the efficiency of code comprehension and to a certain extent can assist in determining whether the program is correct;*" (3) "*... It improves code readability and at the same time improve developers' coding ability, which can help developers write more logically clear code.*"

In addition, a good comment generation tool could also help to check the consistency between code and comments (supported by 75% of respondents) and check whether a code snippet is self-explanatory (supported by 75% of respondents). Inconsistency between the source code and its comment was a critical issue. A good comment generation tool indicated that it could understand the meaning of the source code. Participants could check the consistency by comparing the generated comments and existing comments. On the other hand, a well-generated comment also indicated that the given source code could be well understood by the machine, thus it was very self-explanatory.

We also analyzed the reasons why participants thought the tool was unwise or unimportant (140 participants). The reasons given by them could be grouped into the following categories:

- Useless. Some participants (40) thought this tool was not meaningful. As one participant stated: "*Comments of this sort, which just restate *what* the code does, in 'natural language', are not useful. They are redundant with the code itself, which a programmer should be able to understand at this level on its own, and once created, they risk becoming out of date and inconsistent with the actual code. Comments are useful when they include the human insight that is *not* embodied in the code itself.*"
- Not trustworthy. Some participants (32) doubt the accuracy of the tool. As one participant stated: "*The generated code comments are probably incorrect and developers who do not write comments will not revise the generated comments. It will further affect the readability of the code*"

> **Finding 3.** 80% of the survey respondents think code comment generation tools have the potential to be useful for them. While this finding does not show much these tools would actually help, it shows that most developers do not, out of hand, dismiss the idea of code generation as unneeded.

Although most participants thought such a comment generation tool was useful, few of them had used such a tool. According to the 16 interviewees, only 3 of them have used such tools to generate comments or documentation for source code. These tools help them to generate comment templates and they fill the comment content. Other interviewees have never heard of such comment generation tools or techniques: (1) *"...I used a tool named Doxygen to generate documentation according to the comments and I never heard of a tool that can generate comments. -I7"* (2) *"I just used the tool built-in the IntelliJ to manage my comments. When I typed /**, it can generate a template and I write the comment in the template. -I11"*

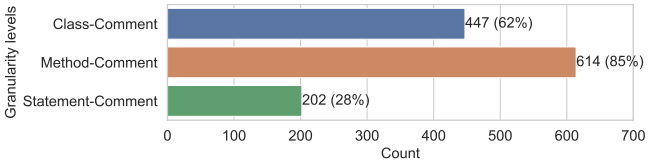


Figure 4: The number of respondents specifying various preferred granularity levels

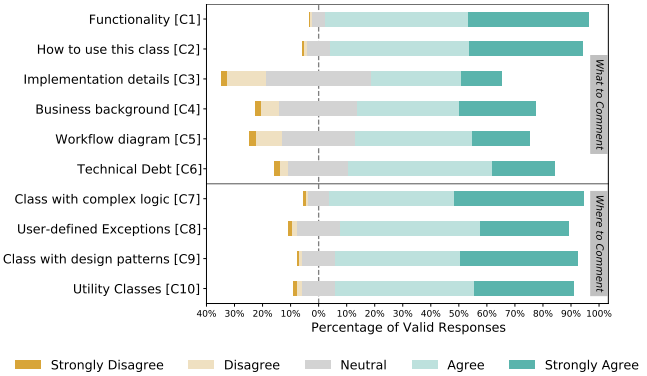


Figure 5: Expectations on class-level comments

3.3 RQ3: Practitioners' Expectations

Different comment generation techniques generate comments for different granularity levels, e.g., class, method, and statements. Figure 4 illustrated participants' preferred granularity levels on generated comments. Among all participants, 62%, 85%, and 28% participants preferred generating class-level, method-level, and statement-level comments, respectively. Note that the percentages did not add up to 100% since a respondent could indicate more than one preferred granularity level. Among the three granularity levels, method-level comments were the most needed comments to be generated and a small number of participants need statement-level comments. In the following part of this section, we will report practitioners' expectations on these three levels of granularity of comments from different aspects, e.g., "*what to comment*", "*where to comment*", and the *preferred length*. "*What to comment*" corresponds to the information that generated comments should include before participants adopt this tool. "*Where to comment*" corresponds to the location that participants expect the tool to generate comments. *Preferred length* aims to investigate practitioners preferences on the lengths of generated comments.

> **Finding 4.** Method-level comments is the most needed type of comments. A small part of participants (28%) expect tools to generate statement-level comments.

3.3.1 RQ 3.1: Expectations on class-level comments. Figure 5 illustrates participants' expectations on class level comments, including "*What to comment*" and "*Where to comment*".

What to Comment. We noticed that the top-3 most information needed to be commented was: *functionality*, *how to use a class*, and *technical debt*. 94% of them expected the *functionality* was included in the generated comments. Functionality description was important for developers, as one participant stated: "*The comment is the functionality description of the source code. It can help developers to understand what the code does...*". The second top information was *how to use a class* which usually described the expected set-up of using the class. The *Technical Debt* such as TODO comments

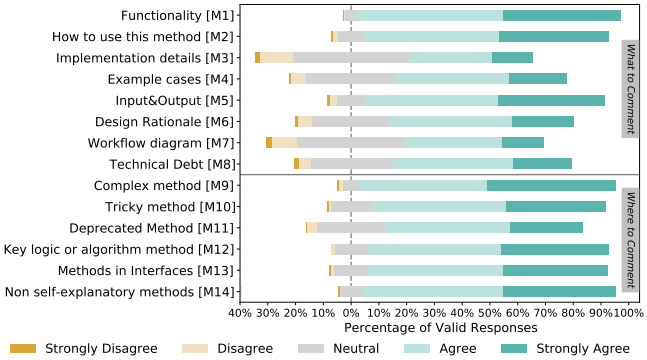


Figure 6: Expectations on method-level comments

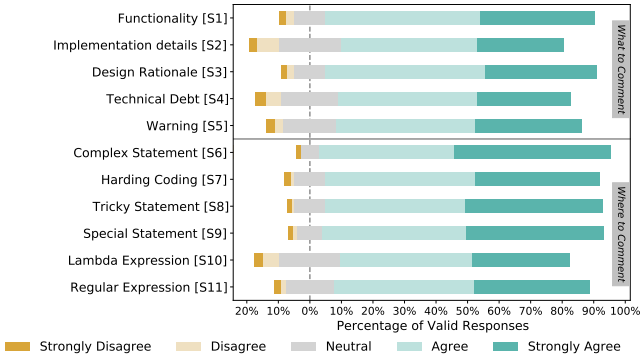


Figure 7: Expectations on statement-level comments

could be used to track problems developers saw and ideas in the class. To comment on a class, the *workflow* description was also important for developers and 62% participants expected tools to generate workflow diagrams for them. A clear workflow diagram could help developers organize and understand the business logic just like one respondent stressed: *"For a business project, it is more important to describe the workflow diagram and the version updates."* The *implementation details* received the least agreement among all statements. As one participant stated: "*... If the tool is just translating the code implementation logic, there is no different from reading the source code...*".

Where to Comment. Commenting at suitable locations was important for code readability and code comprehension. Commenting for *classes with complex logic* received the most agreements (more than 91% respondents). According to interviewees, complex classes were challenging to understand as they usually (1) have long code length; (2) have many loops and conditional statements (e.g., if/switch statements); (3) have many API invocations. It is time-consuming for developers to read these classes and comments are essential for understanding. The other three types of classes received similar agreements, i.e., 86%, 85%, and 82% respondents agree that comments are needed for *classes with design patterns*, *utility classes*, and *user-defined exceptions*, respectively. *Classes with design patterns* usually had special solutions and algorithms. *Utility classes* provided many methods for multiple other classes (shared code) and could be reused many times in a codebase. They should be well-commented for ease of code reuse. *User-defined exceptions* should be commented the trigger conditions in exceptions.

> **Finding 5.** For class-level comments, functionality and how to use a class are the most important information that participants expect automated comment generation tool to generate. Classes with complex logics and design patterns should be well-commented.

3.3.2 RQ 3.2: Expectations on method-level comments. Figure 6 illustrated participants' expectations on method-level comments.

What to Comment. Similar to expectations on class-level comments, *functionality* and *how to use a method* were the top-2 preferred information that should be included in method-level comments. Participants also provided the least support to comments documenting *Implementation details*. Compared to class-level comments, fewer participants agreed to generate comments that included a *workflow diagram* and highlight *technical debt*. 86% participants agreed to generate comments with *Input and Output* information. Parameters and return types were two important parts of methods. Generating the *Input and Output* information was helpful for developers to call methods correctly. 66% participants expected a code comment generation tool to generate comments with the *design rationale* that included authors' intent and why a method exists. As one participant stated: "*Comments have to be insightful and not just describing what the code is doing. Comments are meant to provide background to what the code is doing and why.*" 62% of all respondents supported comments that included *example cases*. *Example cases* provided concrete details on how a method should be used.

Where to Comment. Similar to class-level comments, almost all respondents (92%) supported commenting *complex methods*. The next important methods requiring comments were those that were *non self-explanatory* (supported by 91% respondents). The next important types of methods requiring comments were *tricky method*, *key logic or algorithm method*, and *interfaces*. There was no clear winner among these three types of methods. *Tricky methods* (e.g., check numbers are equal or not using bitwise XOR operators instead of comparison operators) usually had special algorithms and others might be confused by authors' intent behind the source code. *Key logic or algorithm methods* were mainly for processing business logic and good comments were helpful. Commenting *methods in interfaces* was essential as they were often invoked by others. Commenting deprecated methods was also considered helpful (as supported by 71% of the respondents).

> **Finding 6.** For method-level comments, information about functionality, how to use, input and output, and design rationale are considered important. A high proportion of respondents expect automated comment generation tool to generate such pieces of information. In other words, comments should explain code from "*what*", "*how*", and "*why*" aspects. All the different kinds of methods except the deprecated methods received a similar level of support from respondents.

3.3.3 RQ 3.3: Expectations on statement-level comments. Figure 7 illustrated practitioners' expectations on statement-level comments.

What to Comment. Similar to class-level comments and method-level comments, the most information should be included was *functionality* and *design rationale*. The implementation details were also the least important among all statements. *Technique debt* and *warning* (e.g., "Don't input int-type values") received similar agreements.

Where to Comment. 93% of participants agreed to add comments on *complex code statements*. 89% participants expected this tool generate comments for *special statements*. These statements usually processed special business logic, such as, a statement only accepted

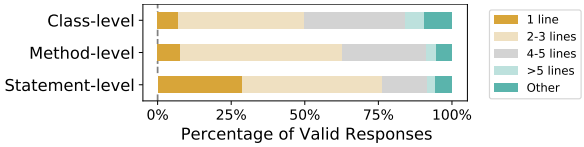
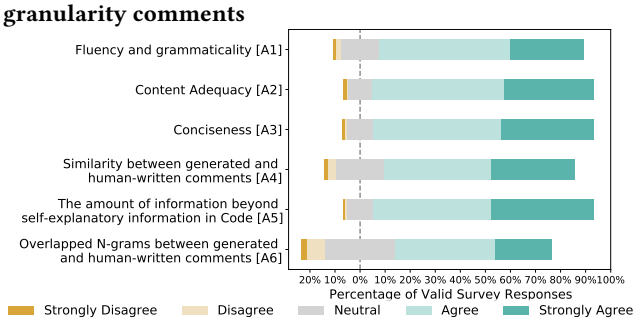
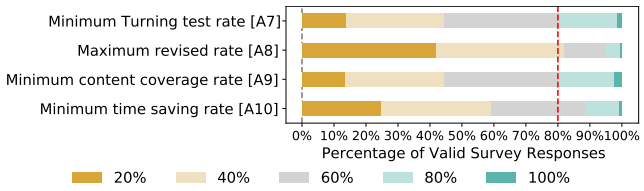


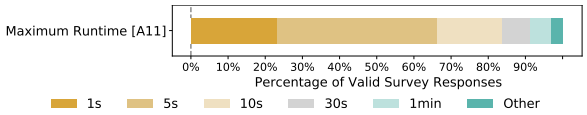
Figure 8: Participants' expectations on lengths for different granularity comments



(a) Evaluation criterion



(b) Effectiveness



(c) Minimum efficiency

Figure 9: Factors that affect practitioners' likelihood to adopt a code comment generation technique.

strings with a special format. *Hard coding* and *tricky statement* received 88% and 89% agreements, respectively. 81% and 73% participants rated the *regular expressions* and *lambda expressions* that should be commented.

> **Finding 7.** For statement-level comments, most respondents require such comments to include *functionality* and *design rationale* information. All the types of statements, except the *Lambda expressions* statement, received similar level of support from respondents.

3.3.4 RQ 3.4: Preferred Length. Figure 8 demonstrated practitioners' expectations on comment length of comments for code units of different levels of granularity (classes, methods, and statements). We could observe that most respondents support 2-3 lines to be the most suitable length. For class-level comments, many participants also agreed to generate comments with 4-5 lines.

> **Finding 8.** Expect tools to generate comments with 2-3 lines for the three levels of granularity (class, method, statement).

3.3.5 RQ 3.5: Adoption Factors. Figure 9 illustrated factors that affect practitioners' likelihood to adopt a code comment generation tool, including *evaluation criterion*, *effectiveness*, and *efficiency*.

Evaluation Criteria. We asked participants' opinions on evaluating code comment generation tools. We observed that the top-3 preferred evaluation criteria were: *amount of additional information* (i.e., amount of information beyond what can be easily gleaned from scanning the source code), *content adequacy* (i.e., the amount of content carried over from the input code to the generated comments), and *conciseness*. More than 88% participants rated agree or strongly agree for these three criteria. 82% participants cared about the fluency and grammaticality of the generated comments. 76% participants thought the *similarity* between machine-generated comments and human-written comments was important, while the *overlapped N-grams* between them received the least support (63%).

Effectiveness. Figure 9(b) showed the percentages of respondents who were satisfied with different rates. The satisfaction rate would be 80% if at least 60% generated comments could pass the Turing Test. If the generated comments contained 60% content of the source code, the satisfaction rate would be 80%. If the revised content in the generated comments was no more than 40%, the satisfaction rate would be 82%. Compared to writing comments, if a participant could save 60% of time using the tools, the satisfaction rate would be 89%.

Efficiency. Figure 9(c) showed the maximum amount of time practitioners were willing to wait for a comment generation technique to provide a recommendation. Few respondents were willing to wait more than one minute for a comment generation technique to do its job (less than 6%). Most participants expected this tool can finish its computation in less than 5 seconds.

3.4 RQ4: Current state-of-the-art research

At the end of our literature review process, we totally identified 25 papers from the premier publication venues in software engineering and artificial intelligence communities. Table 3 showed the capabilities of the state-of-the-art code comment techniques. The Likert score was the average score of different agreements: Strongly Disagree (1 score), Disagree (2 scores), Neutral (3 scores), Agree (4 scores), Strongly Agree (5 scores).

Granularity Level: From Table 3 we could observe that only one paper (i.e., [32]) worked at class-level, which was the second most preferred option. Most papers worked at the method-level granularity that was the most preferred option. Several papers worked at the statement-level. We provided more detailed information below:

- **Class-level comments**: Only Moreno et al. [32] proposed a class-level comment generation technique. This work generated functionality descriptions for complex classes (e.g., class that consists of accessors and mutators) and classes with design patterns.
- **Method-level comments**: 17 papers proposed approaches to generate comments for methods. Most of the proposed tools mainly generated functionality comments that described what a method does. Two papers generated comments to describe how to use a method. In addition, three papers could generate comments to explain the design rationale and answer why a method exists. Although implementation details were not preferred by a large majority of our respondents, four papers proposed techniques to generate comments with implementation details. Only one paper aimed to generate comments for complex methods, i.e., methods with many API invocations.
- **Statement-level comments**: There were four papers generating comments for statements. These tools generated comments that described the functionality of statements. Among them, two

Table 3: Capabilities of Current State-of-Research.

	Statement		Likert score	Papers
Class-level comments	*What to Comment*			
	Functionality	[C1]	4.35	[32]
	How to use this class	[C2]	4.28	-
	Implementation details	[C3]	3.43	-
	Business background	[C4]	3.79	-
	Workflow diagram	[C5]	3.69	-
	Technical Debt	[C6]	3.88	-
	Where to Comment			
	Complex Class	[C7]	4.34	[32]
	User-defined Exception	[C8]	4.09	-
	Class with design patterns	[C9]	4.25	[32]
	Utility classes	[C10]	4.17	-
Method-level comments	*What to Comment*			
	Functionality	[M1]	4.36	[6, 7, 14, 19, 20, 22, 25, 26, 29, 37, 38, 42, 49, 50]
	How to use this method	[M2]	4.24	[30][31]
	Implementation Details	[M3]	3.44	[38][29] [30][31]
	Example cases	[M4]	3.76	[31]
	Input&Output	[M5]	4.20	[44][34]
	Design Rationale	[M6]	3.81	[42][30][31]
	Workflow diagram	[M7]	3.52	-
	Technical debt	[M8]	3.79	-
	Where to Comment			
	Complex Method	[M9]	4.36	[21]
	Tricky Method	[M10]	4.18	-
	Deprecated Method	[M11]	3.93	-
	Key logic and algorithm method	[M12]	4.24	-
	Methods in interfaces	[M13]	4.22	-
	Non Self-explanatory methods	[M14]	4.30	-
	Other	-	-	[6, 7, 14, 19, 20, 22, 25, 26, 34, 37, 38, 42, 44, 48–50, 53]
Statement-level comments	*What to Comment*			
	Functionality	[S1]	4.17	[52][43][11][22]
	Implementation details	[S2]	3.86	-
	Design Rationale	[S3]	4.17	-
	Technical debt	[S4]	3.91	-
	Warning	[S5]	4.04	-
	Where to Comment			
	Complex Statement	[S6]	4.39	[52][43]
	Harding Coding	[S7]	4.21	-
	Tricky Statement	[S8]	4.28	-
	Special Statement	[S9]	4.28	-
	Lambda Expression	[S10]	3.93	-
	Regular Expression	[S11]	4.12	-
	Other	-		[11][22]
Length	1 line	-	-	[7, 11, 14, 19–22, 25, 26, 29, 42–44, 48–50, 52, 53]
	2-3 lines	-	-	[32]
	4-5 lines	-	-	[30][31]
	>5 lines	-	-	-
	Other	-	-	[37, 38]
Evaluation	Fluency and Grammaticality	[A1]	4.06	[32][50][22]
	Content Adequacy	[A2]	4.21	[32][50][42][52][30][31][22]
	Conciseness	[A3]	4.22	[32][42][52][30][31]
	Similarity	[A4]	4.03	[50][20][53]
	Amount of additional information	[A5]	4.27	-
	Overlapped N-grams	[A6]	3.74	[6, 7, 11, 14, 19–22, 25, 26, 29, 34, 48–50, 53]

papers aimed to generate comments for complex statements, e.g., API invocation statements.

> **Finding 9.** Most papers generate comments to describe what a code snippet does (e.g., functionality and implementation details), while a few papers describe how to use and why it exists. Considering the types of code units that need to be commented on, most studies generate comments for all types of code units. However, commenting at the right place is far better than commenting anywhere.

Preferred Lengths: As Table 3 showed, most proposed tools aimed to generate one line code comments. Only 5 papers proposed tools to generated comments with more than one line. However, 2-3 lines comments were supported by most participants.

> **Finding 10.** There is a great discrepancy between the current tools (1 line) and most practitioners expect (2-3 lines) on the length of comments.

Evaluation: We could find that most papers evaluate the quality of generated comments by computing the overlapped N-grams between generated comments and human-written comments, such as BLEU [35], METEOR [10], and ROUGE [27]. These criteria usually involved automated evaluation of generated comments. Unfortunately, evaluating overlapped N-grams was not preferred by a large majority of our respondents. No paper evaluated the amount of additional information beyond what can be easily gleaned from scanning the source code, which most respondents expected to be used to evaluate the generated comments. There are 6, 5, 3, and 3 papers that evaluated the effectiveness of the proposed tools in terms of content adequacy, conciseness, similarity, and fluency and grammaticality, respectively.

> **Finding 11.** Most papers focus on measuring the overlapped N-grams between generated comments and human-written comments that is not preferred by a large majority of our respondents. The criterion *amount of additional information* (i.e., amount of information beyond what can be easily gleaned from scanning the source code) that practitioners valued most is ignored by all studies.

4 DISCUSSION

4.1 Implications

Our results highlight a number of points to be further discussed and several implications for the research community:

4.1.1 Comment completion tools. In addition to generating comments from source code, many developers also expect a tool that can complete comments while they are writing comments. One concern of practitioners about comment generation tools is that they have to spend additional effort to check if the generated comments can express the source code. In fact, our participants mentioned this concern, e.g., *"I don't believe this tool can generate correct comments, thus I have to double-check the generated comments. Compared with writing comments myself, the checking process is more time-consuming. - I2"* A comment completion tool can alleviate this issue, and developers can choose comment recommendations (e.g., the next token, phrase and even sentence) while writing comments. It can not only speed up the commenting process, but also allow the developer to choose the content of the comments. As one practitioner stated: *"Instead of a comment generation tool, I expect a tool that can complete comments during the development, just like the code completion tools in IDE. In this way, I can write comments more efficiently. - I2"*

4.1.2 Identifying where to write comments. According to the reply of our interviewees and respondents of our survey, too many comments are also harmful to code readability and understanding. From the literature review, we can observe that most papers generate comments for any code snippets except constructors or test cases [19, 26]. However, respondents expect tools to generate comments for complex and non self-explanatory code instead of any pieces of code. They point out that it is unnecessary to generate comments for source code that is easy to understand. It is challenging for existing techniques to generate accurate comments for a complex piece of code with long lengths, many API invocations, and many conditional statements. Thus, comment generation techniques should by improved to generate accurate comments for particular locations that practitioners expect.

4.1.3 Describing why a code snippet exists. In addition to describing what a code snippet does and how to use it, code comments should describe why a code snippet exists, i.e., the design rationale and the intent of a developer. However, few studies can generate comments with this information. This is also an important factor as one participant stated: "*A good comment explains "why" not "how." A computer*

is not able to explain "why." Only a human can do that. To generate a comment automatically means that a program must understand the author's intent. This would require artificial intelligence." Thus, to improve the trustworthiness of comment generation techniques, these techniques should have to mine the intent behind the source code.

4.1.4 Evaluation Criterion. Evaluation criterion is another important factor that should satisfy practitioners' expectations. Existing studies usually evaluate the generated comments by comparing the generated comments with human-written comments in terms of overlapped N-grams (such as BLEU scores and ROUGE). However, the overlapped N-grams is the least important among all evaluation criteria. Practitioners expect to evaluate the amount of additional information, whereas none of the collected papers has mentioned this metric. In addition, other metrics, such as Turing Test passing rate, revised rate, and time-saving rate, are also missing in publications.

4.1.5 Detecting inconsistencies between comments and source code. Among all commenting issues we highlighted in our survey, inconsistency between comments and source code is not the most frequently encountered issues, but is the most serious issue. According to Tan et al. [46], many software bugs are caused by a mismatch between programmers' intention and code's implementation. From our survey, 74% respondents agree that a good comment generation tool can help to check the consistency between comments and code. A tool that can detect inconsistency and recommend a good comment simultaneously is needed by practitioners.

4.1.6 Checking if the source code is self-explanatory. Writing self-explanatory code is a common practice for developers. During the development process, developers usually check if the code is self-explanatory. I3 stressed that a good comment generation tool can also help her/him to check the code automatically: *"For me, a comment generation tool could not only help me to write comments and understand programs, but also can check if my code is self-explanatory. If the generated comments can express my code, it means that my code can be comprehended by machines, thus indicates that the code is self-explanatory.-I3 "*

4.2 Threats to Validity

It is possible that some of our survey respondents do not understand code comment generation techniques or our questions well, and thus their responses may introduce noise to the data that we collect. To reduce this threat, we drop responses submitted by people who are neither professional software engineers nor participants of open source projects. We also drop responses by respondents who complete the survey in less than two minutes. Still, we cannot fully ascertain whether participant responses are accurate reflections of their beliefs. This is a common and tolerable threat to validity in many past studies about practitioners' perceptions and expectations, e.g., [24], which assume that the majority of responses truly reflect what respondents truly believe. Another threat is that our participants may not be representations of typical software engineers and that as result our findings may not apply to others. Since we surveyed employees of many software companies as well as open source, we believe this is a minor threat for our study.

5 RELATED WORK

The two major lines of research related to our work are (i) developing tools and approaches to automatically generate code comments and (ii) empirically investigating software documentation practices.

5.1 Automated Code Comment Generation

There has been much work proposing techniques to support the automated generation of code comments. These techniques vary from manually-crafted templates [31, 32, 42], IR techniques [13, 51] to deep-learning-based models [19, 22]. Sridhara et al. [42] and Moreno et al. [32] define heuristics and stereotypes to select the information and create summaries through manually-crafted templates. IR-based approaches [17] usually leverage IR techniques, such as LSI and VSM, to choose top terms from given code snippets. Some researchers [51, 52] retrieve a similar code snippet from a codebase and use its comment to generate comments. Many neural networks have been proposed to generate comments by training on large-scale code corpora in recent years. Iyer et al. [22] propose an encoder-decoder framework to generate comments for C# and SQL statements. Inspired by the neural machine translation, Hu et al. [19] propose the DeepCom to generate comments for Java methods by the seq2seq model. To integrate the structure-information of the source code, Hu et al. [19, 20] and Leclair et al. [26] propose combining the sequential AST information and semantic information together to generate comments. Chen et al. [12] exploited comment categories to boost code summarization. In addition, some studies [53] [50] combine these three techniques, includes, templates, IR, and neural networks.

5.2 Studies on documentation practices

Studies on documentation practices are highly related to this work [4, 5, 8, 15, 18, 40, 41, 45, 47]. Some studies focused on empirical research of general software documentation aspects, including tutorials, logs, and code comments [4, 5]. Table 4 shows an overview of empirical studies on software documentation practices. For example, Aghajani et al. [5] analyzed the issues in different types of software documentation by mining open source software repositories and artifacts related to software documentation. They [4] also presented practitioners' perspectives on software documentation by surveying software practitioners. Some studies investigated a specific type of documentation. For example, Head et al. [18] investigated the information that may be missing from API documentation and provided an understanding of trade-offs of improving missing documentation in header files. Sohan et al. [40] and Uddin et al [47] also investigate the API documentation. Alsuhaibani et al. [8] conduct a questionnaire with software developers to analyze the method names. Safwan and Servant [39] investigated how developers decompose the rationale of code commits. Also related are studies that investigated comment types [36]. Pascarella et al. [36] focus on analyzing the comment types that developers write in source code files and automating classifying code comments. These studies mainly investigated the documentation practices and issues. In addition, they are limited in analyzing "*what*" and "*where*" to comment that developers expect for different types of comments. Different from the aforementioned prior works, we not only reveal the issues with code comments, but also provide detailed expectations that developers have to improve the comment generation

Table 4: Summary of previous works on software documentation practices

Study	Artifacts	Methodology	Summary of findings
Fluri et al. [15]	Code Comments	Investigation with three open source systems	When code and comments coevolve, both are changed in the same revision: 97% of comment changes are done in the same revision as the associated source code change. But code and comments rarely co-evolve
Uddin et al. [47]	API documentation	Questionnaire with 230 software professionals	Respondents prioritized addressing five content-related problems, including incompleteness, ambiguity, unexplained examples, obsoleteness, and inconsistency
Pascarella et al. [36]	Code comments	Exploratory investigation on six major Java OSS systems	Classify comments into 16 inner categories and 6 top categories and the most prominent category of comments summarizes the purpose of the code
Sohan et al. [40]	Usage Examples in API documentation	Study with 26 developers	REST API client developers face productivity problems with using correct data types, data formats, required HTTP headers and request body when documentation lacks usage examples.
Head et al. [18]	API Documentation for C++	Interviews with 18 developers and 8 API maintainers	Updating documentation may provide only limited value for developers, while requiring effort maintainers don't want to invest.
Safwan and Servant [39]	Code Commits	Interviews with 20 software developers and questionnaire with 24 developers	Software developers decompose the rationale of code commits into 15 separate components and the most frequent components are committer, modifications, and location.
Aghajani et al. [5]	Software Documentation	Qualitatively analyze 878 artifacts from open source software repositories	Built 163 types of documentation issues and frequent issues related to the correctness, up-to-dateness and completeness of the information reported in the documentation.
Stapleton et al. [45]	Code Comments	Human study involving 45 both university students and professional developers	participants performed significantly better using human-written summaries versus machine-generated summaries.
Aghajani et al. [4]	Software Documentation	Questionnaire with 146 professional software practitioners	Code Comment and Contribution Guideline were the two documentation types considered as more useful for the different tasks.
Alsuhaibani et al. [8]	Method Names	Questionnaire with 1,604 software developers	Developers are supportive of clearly articulating method naming standards and feel it has a positive impact to code comprehension.
Sondhi et al. [41]	Javadoc comments	Study method documentation and commits logs of 11 open-source projects	62% of the studied Javadoc comments being dependent on other entities
Arafat and Riehle [9]	Code Comments	Investigation on density of comments in open source software code	Successful open source projects follow a consistent practice of documenting their source code and the comment density is independent of team and project size.
Nielebock et al. [33]	Code Comments	Questionnaire with 277 developers	Comments seem to be considered more important in previous studies and by their participants than they are for small programming tasks.

tools. Moreover, we identify and present the gap between practitioners' expectations and capabilities of existing tools proposed by various studies. Our findings show the code comment generation tools have the potential to be useful for developers.

6 CONCLUSION AND FUTURE WORK

Code comment generation is a popular area of research in recent years. In this work, we interviewed 16 professionals and surveyed 720 practitioners on commenting practices and issues they face and their expectations on code comment generation tools. Practitioners are enthusiastic about research in comment generation techniques and expect tools to generate comments for different granularity levels (especially class and method levels). Practitioners expect a comment generation to satisfy factors in terms of comment content, comment locations, evaluation criteria, effectiveness, and efficiency. We also compare capabilities of current state-of-research in comment generation with practitioners' expectation for adoption to identify discrepancies. We point out the limitations of the current state-of-research and avenues for future work to make code comment generation techniques well-adopted by practitioners. Future studies could put more effort into generating comments at the right locations instead of generating comments for all types of code units. Besides, studies could put more effort into investigating the evaluation criteria that practitioners valued most.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation of China (No. 62141222 and No. U20A20173) and the National Research Foundation, Singapore under its Industry Alignment Fund – Prepositioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] 2021. Nvivo qualitative data analysis software.

[2] https://github.com/xing-hu/Practitioners-Expectations-on-Automated-Code-Comment-Generation.

[3] https://www.wjx.cn.

[4] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. 2020. Software documentation: the practitioners' perspective. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 590–601.

[5] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1199–1210.

[6] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007.

[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[8] Reem S Alsuhaibani, Christian D Newman, Michael J Decker, Michael L Collard, and Jonathan I Maletic. 2021. On the Naming of Methods: A Survey of Professional Developers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 587–599.

[9] Oliver Arafat and Dirk Riehle. 2009. The Commenting Practice of Open Source. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 857–864. https://doi.org/10.1145/1639950.1640047

[10] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 65–72.

[11] Ruichu Cai, Zhihao Liang, Boyan Xu, Yuexing Hao, Yao Chen, et al. 2020. TAG: Type Auxiliary Guiding for Code Comment Generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 291–301.

[12] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.

[13] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 13–22.

[14] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured neural summarization. *arXiv preprint arXiv:1811.01824* (2018).

[15] Beat Fluri, Michael Wursch, and Harald C Gall. 2007. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 70–79.

[16] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to Comment "Translation": Data, Metrics, Baselining & Evaluation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 746–757.

[17] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.

[18] Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. 2018. When not to comment: questions and tradeoffs with api documentation for c++ projects. In *Proceedings of the 40th International Conference on Software Engineering*. 643–653.

[19] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[20] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

[21] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred api knowledge.(2018). In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelli-gence (IJCAI 2018), Stockholm, Sweden, 2018 July 13*, Vol. 19. 2269–2275.

[22] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[23] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 135–146.

[24] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649.

[25] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*. 184–195.

[26] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 795–806.

[27] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*. 74–81.

[28] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 373–384.

[29] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating Just-In-Time Comment Updating. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–597.

[30] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. 279–290.

[31] Paul W McBurney and Collin McMillan. 2015. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2015), 103–119.

[32] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[33] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2018. Commenting Source Code: Is It Worth It For Small Programming Tasks? *Springer Empirical Software Engineering (EMSE)* 24, 3 (2018), 1418–1457. https://doi.org/10.1007/s10664-018-9664-z

[34] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020. Learning to Update Natural Language Comments Based on Code Changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 1853–1868.

[35] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[36] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 227–237.

[37] P. Rodeghero, C. Liu, P. W. McBurney, and C. McMillan. 2015. An Eye-Tracking Study of Java Programmers and Application to Source Code Summarization. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1038–1054. https://doi.org/10.1109/TSE.2015.2442238

[38] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. 2014. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*. 390–401.

[39] Khadijah Al Safwan and Francisco Servant. 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*. 397–408.

[40] SM Sohan, Frank Maurer, Craig Anslow, and Martin P Robillard. 2017. A study of the effectiveness of usage examples in REST API documentation. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 53–61.

[41] Devika Sondhi, Avyakt Gupta, Salil Purandare, Ankit Rana, Deepanshu Kaushal, and Rahul Purandare. 2021. On Indirectly Dependent Documentation in the Context of Code Evolution: A Study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1498–1509.

[42] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 43–52.

[43] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 101–110.

[44] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Generating parameter comments and integrating with method summaries. In *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 71–80.

[45] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A Human Study of Comprehension and Code Summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. 2–13.

[46] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 145–158.

[47] Gias Uddin and Martin P Robillard. 2015. How API documentation fails. *Ieee software* 32, 4 (2015), 68–75.

[48] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[49] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems 2019*. Neural Information Processing Systems (NIPS).

[50] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 349–360.

[51] E. Wong, Taiyue Liu, and L. Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 380–389. https://doi.org/10.1109/SANER.2015.7081848

[52] Edmund Wong, Jinqiu Yang, and Lin Tan. 2013. Autocomment: Mining question and answer sites for automatic comment generation. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 562–567.

[53] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1385–1397.