An Empirical Study of Developer Discussions in the Gitter Platform

OSAMA EHSAN, Queen's University, Canada
SAFWAT HASSAN, Queen's University, Canada
MARIAM EL MEZOUAR, Royal Military College, Canada
YING ZOU, Queen's University, Canada

Developer chatrooms (e.g., the Gitter platform) are gaining popularity as a communication channel among developers. In developer chatrooms, a developer (*asker*) posts questions and other developers (*respondents*) respond to the posted questions. The interaction between askers and respondents results in a discussion *thread*. Recent studies show that developers use chatrooms to inquire about issues, discuss development ideas, and help each other. However, prior work focuses mainly on analyzing individual messages of a chatroom without analyzing the discussion thread in a chatroom. Developer chatroom discussions are context-sensitive, entangled, and include multiple participants which make it hard to accurately identify threads. Therefore, prior work has limited capability to show the interactions among developers within a chatroom by analyzing only individual messages.

In this paper, we perform an in-depth analysis of the Gitter platform (i.e., developer chatrooms) by analyzing 6,605,248 messages of 709 chatrooms. To analyze the characteristics of the posted questions and the impact on the response behavior (e.g., whether the posted questions get responses), we propose an approach that identifies discussion threads in chatrooms with high precision (i.e., 0.81 F-score). Our results show that inactive members responded more often and unique questions take longer discussion time than simple questions. We also find that clear and concise questions are more likely to be responded than poorly written questions.

We further manually analyze a randomly selected sample of 384 threads to examine how respondents resolve the raised questions. We observe that more than 80% of the studied threads are resolved. Advanced-level/beginner-level questions along with the edited questions are the mostly resolved questions. Our results can help the project maintainers understand the nature of the discussion threads (e.g., the topic trends). Project maintainers can also benefit from our thread identification approach to spot the common repeated threads and use these threads as frequently asked questions (FAQs) to improve the documentation of their projects.

Additional Key Words and Phrases: chat disentanglement, developer threads, thread identification, mixed-effect models, developer chatrooms, Gitter

ACM Reference Format:
Osama Ehsan, Safwat Hassan, Mariam El Mezouar, and Ying Zou. 2020. An Empirical Study of Developer Discussions in the Gitter Platform. 1, 1 (April 2020), 38 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Osama Ehsan, Queen's University, Department of Electrical and Computer Engineering, Kingston, Ontario, Canada, osama.ehsan@queensu.ca; Safwat Hassan, Queen's University, School of computing, Kingston, Ontario, Canada, shassan@cs.queensu.ca; Mariam El Mezouar, Royal Military College, , Kingston, Ontario, Canada, mariam.el-mezouar@rmc-cmr.ca; Ying Zou, Queen's University, Department of Electrical and Computer Engineering, Kingston, Ontario, Canada, ying.zou@queensu.ca.

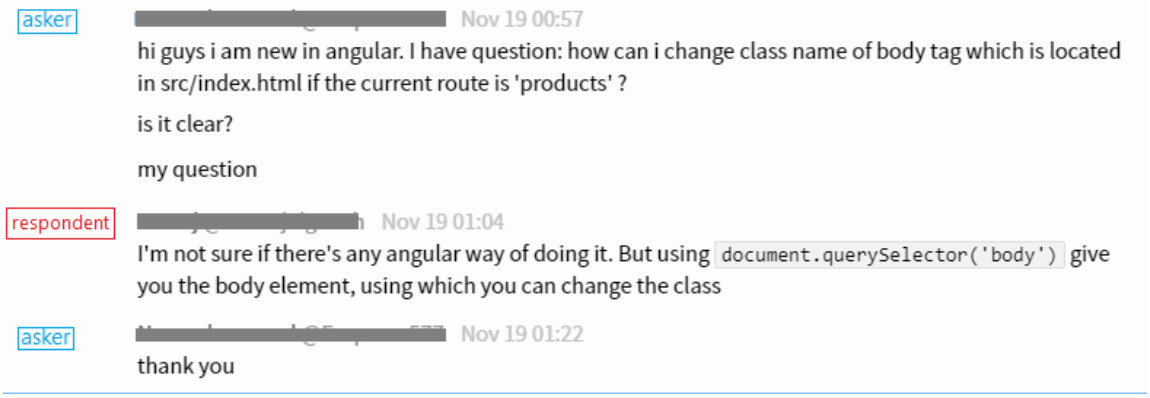

Fig. 1. An example for an annotated screenshot of a discussion thread in the Angular chatroom. The original names of the participating developers are annonymized (i.e., replaced with asker and respondent) to ensure their privacy.

1 INTRODUCTION

Open Source Software (OSS) projects receive contributions from developers from around the world [61]. To facilitate the global collaboration among the widely-distributed developers, different communication platforms (such as emails, Internet Relay Chat (IRC) channels, Q&A forums, and chatrooms) are widely used in the software development process. Recent studies on developer communication practices show that chatrooms are emerging as a popular channel of communication for developers [37]. For instance, developers leverage chatroom platforms (such as Gitter[1] and Slack[2]) to ask for guidance about different development activities. Developers use chatrooms to ask general inquiries, discuss new features in a certain project, or ask for advice to improve the quality of their code [13] [28] [54].

Unlike other chatroom platforms, Gitter is a chatroom platform that is directed towards GitHub projects. Project maintainers use Gitter to create chatrooms for their GitHub projects. Then, developers use chatrooms to discuss issues [18] [53]. A recent report shows that there are more than 300 thousand active developers on Gitter every month [52]. One of the main reasons behind the popularity of Gitter is the integration with the GitHub projects. For example, Gitter enables developers to associate their discussions with bug reports, commits and pull requests of the linked GitHub project [57].

In Gitter, developers (*askers*) raise questions (e.g., how to use a certain feature in a project). Later, other developers (*respondents*) answer the raised questions. The interaction between an asker and respondents in a chatroom result in a discussion *thread*. The interaction in Figure 1 shows an example for an annotated screenshot of a discussion thread in the Angular chatroom[3]. Different from Q&A platforms (e.g., Stack Overflow), which provide support to track the quality of the posts using the accepted answers and the number of votes, chatrooms facilitate an informal discussion between developers. Therefore, we can not directly examine the quality of the responses in the chatrooms.

Prior work proposed approaches to extract useful information from chatroom messages (e.g., identifying the topics of the posted messages) [7] [20]. However, prior work focuses mainly on the analysis of individual messages without giving an in-depth analysis of threads to understand the trends in the chatroom discussions. Prior work also analyzes Q&A platforms (e.g., Stack Overflow) to help developers better write their questions [10] [15] [63] [71], extract the

[1] https://gitter.im/
[2] https://slack.com/
[3] https://gitter.im/angular/angular

discussion topics [3] [4] [31] , recommend answerers for a new question [12] [46] , and identify the related questions to a newly posted question [27] [48]. However, to the best of our knowledge, there is no prior work that analyzes the discussion threads in Gitter to help chatroom developers and project maintainers understand the nature of the discussion threads in chatrooms.

In this paper, we perform an in-depth analysis of the discussion threads in Gitter to gain a better understanding of the communication among developers. We are interested in the analysis of the response time for the raised questions, the total discussion time, and the discussed topics. Moreover, we analyze discussion threads that can help project maintainers prioritize their maintenance activities to provide better support for their projects. For example, the most frequent questions can be spotted, along with their answers to aid the project maintainers to improve the documentation of their projects. In addition, the analysis of the emerging issues in the thread discussions can help project maintainers to resolve these issues proactively.

More specifically, we analyze 6,605,248 messages that are posted by 370,302 developers in 709 chatrooms. The chatroom data is collected from October 10^{th} 2014 until March 30^{th} 2019. First, we propose an approach to automatically identify threads in chatrooms. Then, we perform a qualitative and quantitative analysis of the identified threads to identify the discussion time and discussion topics. Our work analyzes the discussion threads along with the following three research questions (RQs):

RQ1: *What is the accuracy of the proposed approach for identifying a discussion thread?*

In this research question, we evaluate the accuracy of the proposed approach for the identification of the discussion threads by calculating the F-measure of the proposed approach. Our evaluation of automatic labeling for a statistical representative sample of the studied dataset shows F1-Score to be 0.81. Our findings show that threads can be identified from the text messages in chatrooms by measuring three aspects (i.e., the involved users, message content, and back-and-forth communication). Our approach identified 708,294 threads (i.e., 708,294 messages that triggered developer discussions) and 420,857 messages with no response.

RQ2: *What makes a question getting responses in developer chatrooms?*

In this research question, we examine the characteristics of the posted questions which may affect in getting a response. Our findings show that questions posted by inactive developers are more likely to be responded by the members of the chatroom. We also observe that providing details in a message rather than external links are expected to get a response. Our findings show that each chatroom has different behaviors in terms of responding to the queries posted. For example, we identify four patterns of respondents (e.g., developers who respond to simple and long questions).

RQ3: *What features show an association with the resolution outcome of the discussion threads?*

In this research question, we investigate the topics of the threads and the possibility of threads getting resolved (i.e., the posted questions are resolved). Our findings show that askers mostly post *How-to questions*, which includes installation, compilation, implementation inquires, and solving bugs. Around 80% of the studied threads posted in the chatrooms are resolved. In particular, advanced and beginner-level questions have a higher chance of being resolved than other questions. We also find that the topic of the question and whether a question has been edited has a significant effect on resolving threads.

The key contributions of our study are as follows:

- We propose an approach for automatic identification of the discussion threads in developer chatrooms. Our thread identification approach identifies threads with F1 score equals to 0.81.

- We perform a large scale analysis of the discussion threads in chatrooms. In particular, we perform a quantitative analysis of the characteristics of 708,294 developer questions that initiate thread discussions. Our analysis identifies four patterns of respondents (e.g., developers who mainly respond to longer and simple messages). Our results can help project maintainers set up guidelines for their chatrooms to help the askers write their questions in a suitable way for each chatroom.
- We provide a qualitative analysis of the resolution outcomes, resolution types, and the discussed topics in the chatrooms. We provide a taxonomy of the resolution types and the discussed topics in the chatrooms. Our results can be used to observe the trending topics and the ratios of developers getting their problems resolved in the chatrooms.

Paper Organization: The rest of this paper is organized as follows. Section 2 gives background information about developer chatrooms and describes the features of Gitter. Section 3 illustrates our approach for collecting and processing Gitter data. Section 4 represents the results of our research questions. Section 5 discusses the implication of our study. Section 6 summarizes the prior studies and related work in this domain. Section 7 explains the limitations and threats to the validity of our study. Finally, Section 8 summarizes our work.

2 BACKGROUND

In this section, we describe how developers use Gitter to communicate with each other. As described in Section 1, Gitter enables project maintainers to create a chatroom and link it to their GitHub project. Consequently, developers can visualize the ongoing activities of the associated repository, such as tracking the status of the reported issues and following the comments on pull requests. Figure 2 shows the three main elements of a Gitter chatroom, i.e., (1) developers, (2) the posted messages, and (3) the project activities.

(1) Developers: Developers use Gitter by raising a new question (*asker*) or responding to the raised question (*respondents*). For example, as shown in Figure 2, a developer asks a question, and another developer responds to the raised question. Developers access Gitter chatrooms with their GitHub credentials. Hence, the developer's GitHub profiles are visible to other developers through Gitter. We leverage the provided GitHub profile data in our qualitative analysis in Section 4 (RQ2).

(2) The posted messages: Developers (either askers or respondents) post their messages, which contain the plain text of the questions or responses. In addition, the posted messages can contain the following information:

- *User mentions:* In Gitter, messages are presented in sequential order (without being structured in a clear thread format). Hence, developers tag (using `@username`) other developers to respond to an earlier message. In addition, developers may tag specific developers to ask them to guide the raised question.
- *Issue mentions:* When developers try to refer to any issue from the repository, they use the notation `#issue-number`. This notation is referred to as issue mentions. The feature is used by developers to refer to some existing issues and discuss these issues explicitly. For example, a developer asked the following question in the Angular chatroom. *"I opened an issue regarding the angular router in case anyone is interested or might be familiar with a fix. angular/angular/33000*".

Finally, the conversation between an asker and the respondents results in a discussion thread. A single thread contains at least two messages from two different developers. Figure 2 shows an example of a discussion thread with an asker raising an issue about setting the default value of the radio buttons using the Angular platform. The respondent answers the issue. As a result, the asker expresses the appreciation of the answer.

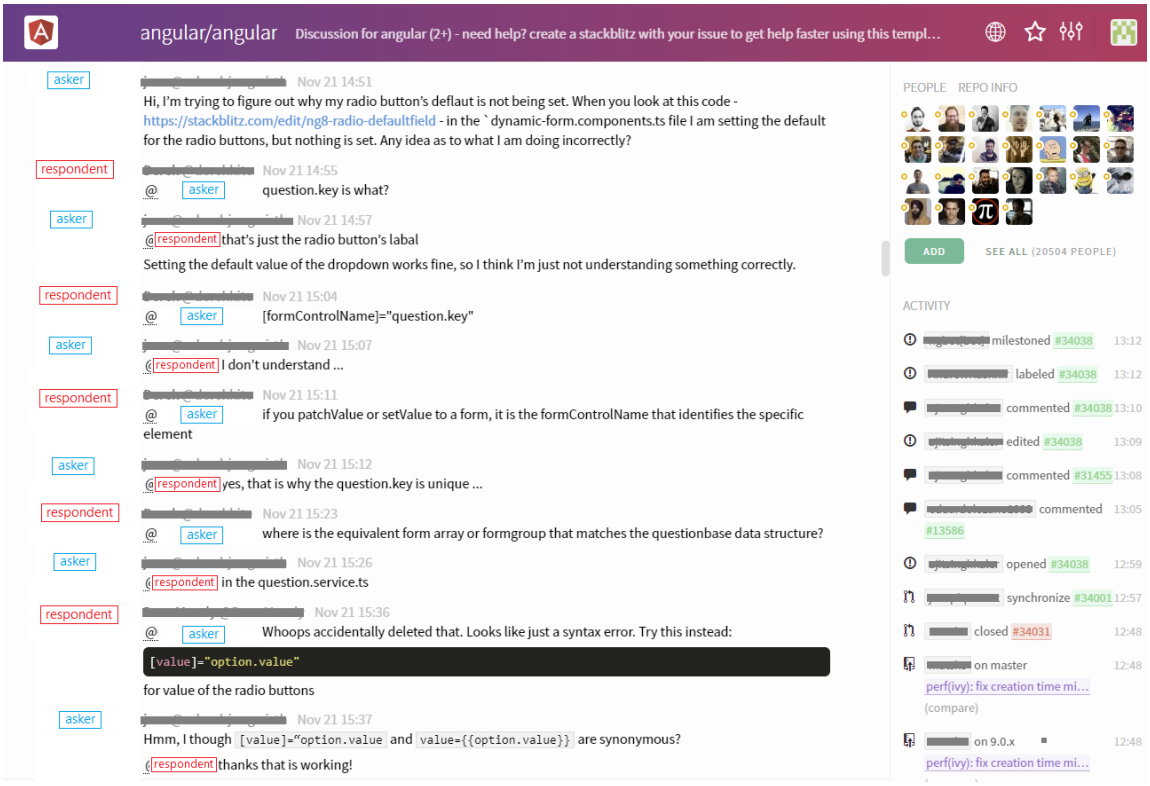

Fig. 2. An annotated screenshot of an example discussion thread in the Angular chatroom. The original names of the participating developers are annonymized (i.e., replaced with asker and respondent) to ensure their privacy.

(3) GitHub Project activities: The project activity panel shows the status of the created issue reports in the linked GitHub project. Project activities help developers to directly see the created issue reports in GitHub, comments on the existing issues, any updates to the status of the issue, and code commits.

3 EXPERIMENT SETUP

In this section, we describe the approach used to collect the chat messages from the Gitter chatrooms, and the steps conducted to process the collected messages. Figure 3 shows an overview of our approach. The details of our approach are described in the following sub-sections.

3.1 Collecting chat data

The chat messages are collected from the selected chatrooms in Gitter in the following steps:

(a) Selecting public chatrooms: In Gitter, every chatroom belongs to a specific chatroom category. Gitter contains 24 chatroom categories, such as Ruby, Android, Frontend, DevOps, and Data science[4]. For example, the chatrooms Angular and VueJS belong to the chatroom category JavaScript. In our study, we select all the publicly available 856 chatrooms that belong to 24 categories.

[4]https://www.gitter.im/home/explore/

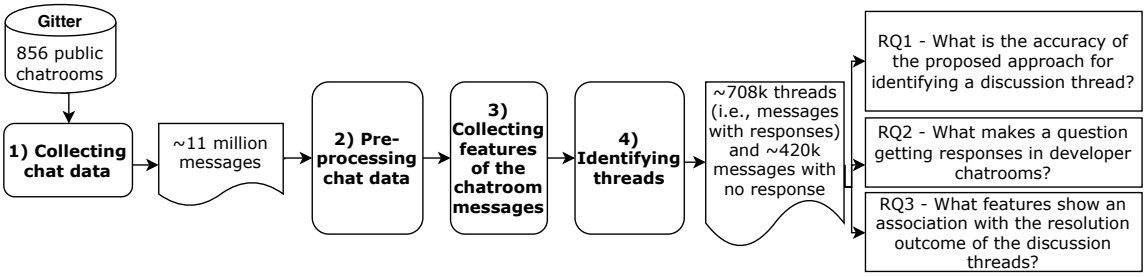

Fig. 3. An overview of our data collection and data processing steps.

(b) Crawling the chat history of the selected chatrooms: Gitter provides an API [5] that enables the members of a given chatroom to download the chatroom history (i.e., all the posted messages in a chatroom). Table 1 shows the list of attributes available for each message in the chat history. We downloaded the messages of the 856 publicly available chatrooms resulting in a total of 11,049,802 messages that are posted by 370,886 developers. We collected Gitter data in March 2019. Hence, the history of the collected chat messages spans over a duration starting from the creation date of each chatroom (i.e., the oldest creation date is October 10th, 2014) and to ending date on March 30th, 2019.

(c) Filtering out the inactive chatrooms: In Gitter, developers create chatrooms; however, not all the created chatrooms are actively used. To understand the discussion threads that occur among developers of a given chatroom, we select active chatrooms (i.e., with more than ten messages). The chatrooms with less than ten messages rarely have any discussion threads. After discarding the chatrooms with less than ten messages, we end up with 709 chatrooms with a total of 11,049,686 messages posted by around 370,302 developers.

3.2 Pre-processing chat data

In this study, our objective is to analyze discussion threads by first identifying the messages that belong to the same thread. In developer chatrooms, messages that belong to the same thread may contain similar words (e.g., discussing the same technical issues). However, Gitter messages contain informal text which may contain typos, slangs that makes it difficult to measure the similarity among the messages that belong to the same thread. To facilitate the identification of the discussion threads, we follow the standard text pre-processing steps [44] as follows.

Removing the stop words. Chat messages between developers may include a significant amount of non-informative keywords (e.g., *a, an, the, of, for*). We remove the stop words from the messages using the NLTK stop words corpus [38] [39]. To enrich the stop words corpus, we follow the approach by Noei et al. [44] by identifying the most frequent words and adding them to the corpus of existing stop words. The new words added to the corpus of stop words are the following words: *could, can, can't, yeah, quo, quot, might, and, hey, ill, you, someone, would*, and *also*.

Lemmatizing the text messages. Chat messages usually contain different representations of the same word (i.e., a word can appear under different forms in the chat messages). Lemmatization is a process used to convert the different forms of a given word to the normalized form. For example, *implemented*, *implementing*, *implements* are lemmatized to the root word *implement*. Hence, we perform the lemmatization to convert different forms of words to the normalized representation [49].

[5] https://developer.gitter.im/

Table 1. The attributes collected for each chat message and their descriptions.

Message attribute	Description
id	ID of the posted message
text	The posted message content
message time	Date and time of the posted messages
message edit time	Date and time at which message is edited
FromDeveloper	GitHub developer ID who posted the message
mentions	List of the mentioned developers
issues	List of referenced GitHub issues
URLs	List of URLs included in the posted message
HTML	The posted message in an HTML layout/structure

Spell checking. Chat messages may contain typos, which can negatively impact the quality of the text analysis [45]. Hence, we use the *sym spell checker* [14] to correct the spelling mistakes in chat messages.

To further improve our data set of chat messages, we combine the consecutive messages from the same developer. Developers usually split their messages into multiple consecutive messages. Figure 2 shows an example of consecutive messages from an asker. For example, the asker posted the message *"hmm, I thought ..."* and then the message *"thanks that is working"*. We combine the consecutive messages that are immediately posted after each other in a chatroom from the same developer into a single message. The merging of the consecutive messages results in 6,605,248 messages from 370,302 unique developers.

3.3 Collecting features

In this section, we describe the feature collection related to every message, which is later used in Section 4. The features are divided into two categories: *A) message features* that describe the posted messages; and *B) user features* that characterize the authors of the messages. Table 2 lists the collected features, along with their types (i.e., categorical or numerical), and the rationale for selecting each feature.

A) Message features: For every chat message initiating a thread, we collect multiple features associated with the chat message. We describe below the list of message features.

- *Lexicons:* The lexicon feature represents the total number of words in a message.
- *Code snippets:* We identify the presence of code snippets in a message. The presence of code snippets in a chat message is indicated using the tag `<code>` in the HTML attribute of the collected chat data.
- *URLs:* We identify the presence of links (i.e., whether an asker refers to an external online resource) in a message. The presence of links in a chat message is indicated using the `URL` attribute of the collected chat messages.
- *Weekday:* We collect the day when the message is posted during the weekdays. The weekday (i.e., Monday, Tuesday, Wednesday, Thursday, and Friday) feature in a chat message is extracted using the `time` attribute of the collected chat messages.
- *Daytime:* We collect the time when the message is posted during the daytime (i.e., from 09:00 a.m. to 05:00 p.m.). The daytime feature of a chat message is extracted using the `time` attribute of the collected chat messages.
- *Edited:* We identify whether the posted message is edited (e.g., an asker may add further details to the posted question). The edited feature of a chat message is identified using the `edited` attribute of the collected chat messages.

Table 2. The collected features for each message in Thread$_{starters}$ along with their type and rationale.

Feature name	Type	Rationale
Message features		
lexicons	numerical	Length of the messages can impact the response behavior. For example, developers may tend to respond to long and descriptive messages [30] [50].
code snippets	categorical	Presence of code snippets corresponds to the code details included in message which can help respondents in understanding the question [11] [63].
URLs	categorical	Presence of URLs in the message can affect response behaviors. For example, developers may tend not to open untrusted external links [11] [50] [63] [71].
weekdays	categorical	Developers could be willing to help in weekdays more than on weekends [9] [11] [63].
daytime	categorical	Developers could be active and responding more during the daytime [9] [11] [63].
edited	categorical	The message posted and edited afterwards can have an impact on getting a response as asker might have included more details for respondents to clarify the posted question [11].
issue mentions	categorical	As chatrooms are associated with corresponding GitHub projects, asker may mention the issues related to the raised question. Hence, developers who are familiar with the mentioned issue may respond to the raised question.
readability (CLI)	numerical	The readability of the posted messages may impact the response behavior (e.g., whether the message being responded) [32] [50] [51] [72].
message singularity	numerical	Respondents may tend to answer unique questions rather than a repeated and similar question.
Developer features		
commits/issues/pull request/ reviews	numerical	GitHub related activities (e.g., the number of code commits) may impact the behavior of the respondents. For example, active developers may tend to solve raised questions.
active GitHub contributor	categorical	Activity in the GitHub represents that a developer is contributing to the open-source community. An active GitHub contributor may be willing to contribute and help other developers solve their raised questions in the linked chatroom.
active chatroom participant	categorical	Activity in the chatrooms represents that a developer has more knowledge about the specific project. Such a developer is expected to have more knowledge in terms of communicating within a chatroom.

- *Issue mentions:* We calculate the number of issue reports that are mentioned in a message. The issue mentions feature of a chat message is identified using the `issues` attribute of the collected chat messages.
- *Readability (CLI):* Coleman Lieu Index (CLI) [41] is used to identify the readability of the text (e.g., the readability of bug reports [32] and pull requests [72]). We use CLI to identify the readability of the posted messages. CLI indicates the level of difficulty of a text in terms of readability score from 1 (easy) to 12 (hard). Equation 1 shows how CLI is calculated.

$$CLI = 0.0588 * L - 0.296 * S - 15.8 \quad (1)$$

where L is the average number of characters per 100 words, and S is the average number of sentences per 100 words.

- *Message singularity:* Message singularity shows the uniqueness of a chat message with respect to the other messages (i.e., the ongoing discussions) in the same chatroom. The basic goal behind the feature "message singularity" is to identify whether the message contains a frequently discussed topic, such as discussing a commonly used API in the related GitHub project. A similar metric is used by Ponzanelli *et al.* [51] in identifying low-quality questions. We compute the singularity of a chat message with the messages that are posted within the last 30 days in the same chatroom. In particular, we used the Term Frequency-Inverse Document Frequency (TF-IDF) [55] to measure the singularity of the message. TF-IDF is a measure to calculate the relevance of a word for the document. In our context, we calculate the relevance of the chat message with respect to the chat messages of the last 30 days from the same chatroom.

B) User features: We calculate a set of features that are related to the author of the chat message (i.e., the asker). The user features are collected from: *a)* GitHub and *b)* the chatroom where the query was posted. For the GitHub related features, we crawl the yearly summaries of the developer activities (e.g., the number of submitted pull requests). For each asker, we collect the yearly summaries of the asker activities for three consecutive years (2016, 2017, and 2018), as 90% of the studied chat messages are posted in this period. In particular, we collect the following user-related features.

- *Commits:* A commit is an individual change to a file or set of files. We collect the number of commits for each asker.
- *Issues:* Issue reports contain the identified bugs, the suggested improvements, and the raised questions that are related to code repositories. We collect the number of reported/resolved issues for each asker.
- *Pull requests:* Pull requests are proposed code changes to a code repository. Pull requests are submitted by a developer and accepted or rejected by the repository maintainers. We extract the number of pull requests by each asker.
- *Reviews:* Code reviews are defined as submitting a review of a pull request. We compute the number of code reviews by each asker.
- *Active chatroom participant:* An active chatroom participant in a chatroom is defined as the number of messages posted by a developer in the chatroom. As the developer can be active at some time and inactive at other times, we label the developers as active or inactive based on their activity in the last 30 days. We used a simple formula for the calculation of the participation status of each developer, as presented in Equation 2. For each message that is posted by an asker, we calculated the total number of messages that are posted by every developer in the same chatroom (in the last 30 days). Then, we identified the 4^{th} quantile (threshold) of the distribution. If the number of messages posted by an asker (U) in the last 30 days false in the 4^{th} quantile, then the developer is labeled as active otherwise, we label the developer as inactive.

$$ActiveChatroomParticipant = \begin{cases} Active & \text{if } U > threshold, \\ Inactive & \text{otherwise.} \end{cases} \tag{2}$$

where

$threshold = 4^{th}$ quantile of messages by unique developers (last 30 days)

U = The total number of messages by single developer (last 30 days)

Table 3. Examples of two manually identified threads (T1 and T2) from Sample$_{initial}$ extracted from the Ruby chatroom.

Message ID	Developer	Message	Thread number
M1	D1	*"hi"*	T1
M2	D2	*"hello"*	T1
M3	D1	*"is rails hard to learn?"*	T1
M4	D2	*"Are you familiar with any other MVC framework?"*	T1
M5	D1	*"me? not"*	T1
M6	D2	*"There may be an initial steep learning curve to get the hang of the MVC framework, but overall Rails is quite easy to learn and there are tons of good tutorials out there to help with this"*	T1
M7	D1	*"Thanks. what about ruby? or something like that I heard before ruby and rails not same thing?"*	T1
M8	D3	*"So, my app is getting a "http error 500" server error. When I run rake on my rails production server I get a "migrations are pending" error, but my "rake db:migrate" runs fine. Anyone run into this or know how to fix this problem?"*	T2
M9	D2	*"Ruby is the language, Rails is an MVC framework built on Ruby @D1"*	T1
M10	D4	*"It's helpful to know ruby though if you're using rails. In my opinion. @D1"*	T1
M11	D1	*"@D2 Thanks you alot"*	T1
M12	D5	*"@D3 try RAILS-ENV=production rake db:migrate"*	T2

- *Active GitHub contributor:* This feature marks whether an asker is an active contributor in the linked GitHub project. First, we extract the number of contributions (e.g., GitHub commit messages or code reviews) by each developer. Then, we define the active GitHub contributor feature as true if there is at least one GitHub activity (e.g., one pull request) in the current year. Not all developers frequently participate in GitHub (the median of GitHub activities of a developer is one activity per year). Hence, an asker/respondent with at least a single contribution shows that the developer was active on GitHub at some point during the year.

3.4 Identifying threads

To identify threads in a chatroom, we apply the following two-step approach.

Step 1: Identifying the common features in the messages of a thread. To understand the nature of the messages in a single thread, we randomly select 1,000 consecutive messages (Sample$_{initial}$) from a random chatroom (Ruby[6]). The first and the third author of this paper systematically analyzes Sample$_{initial}$ by manually labeling the messages that belong to the same thread, as shown in Table 3. For each manually labeled thread, we identify one or more features that are shared by the same-thread messages. For example, in Table 3, Message *M9* can be identified as part of Thread *T1* with the help of the user mention feature. Based on our manual analysis, we observe three categories of features that can be used to identify the messages belonging to the same thread as follows.

1- The behaviors of the involved users: As described in Section 2, Gitter enables developers to directly ping a specific developer using the *mention* feature to answer a specific message. The *mention* feature is frequently used and can be leveraged to link the same-thread messages. Table 3 shows a sample of messages exchanged among more than two developers from our dataset Sample$_{initial}$. *M12* shows an example of the use of a mention, where developer *D5* explicitly mentions developer *D3* (the author of message *M8*). Consequently, messages M8 and M12 labeled as part of

[6]https://gitter.im/ruby/ruby

Table 4. The metrics used for the thread identification in a Gitter developer chatroom

Metric name	Description	Rationale
	Developer metrics	
1) Mentions	Given a group of involved developers in a thread T_i, we identify that a message M_i belongs to thread T_i, if the message **mentions** any of the involved developers.	Developers usually involve other developers in a discussion by mentioning them. Hence, messages containing mentions to any of the involved developers are linked to the same thread.
2) Involved developers	Given a group of involved developers in a thread T_i, we identify that a message M_i belongs to the thread T_i, if the message is posted by one of the involved developers.	Once the discussion starts, developers stop mentioning each other, while continuing the discussion.
	Content Metrics	
3) Bi-grams	Bi-grams are sequences of two relevant words that do not appear consecutively by accident. Given a new message M_i and the identified thread T_i, we identify the message M_i as part of the thread T_i, if at least one bi-gram of the message M_i and the existing messages in the thread T_i matched.	A message from the same thread contains similar word pairs (i.e., bi-grams) to thread [19]. Hence, it is expected that the messages of the same thread share the same pair of words.
	Discussion Metrics	
4) Back-and-forth communication	Given a new message M_j and identified thread T_i, the messages M_i and M_j (from developers D1 and D2 respectively) belong to the thread T_i, if the two developers D1 and D2 repeatedly communicate in a back-and-forth manner with each other for n times, where n can be an arbitrary number, such as 2 or 3.	Developers can start a discussion without mentioning each other. Instead, they engage in a back-and-forth exchange of the messages. Hence, the pattern of exchange can be identified by tracking the occurrences of the consecutive back-and-forth messages between two involved developers.

the same thread (*T2*). As a given thread progresses, the involved developer is likely to no longer mention each other. If a developer D_i is already part of the thread, subsequent messages from the same developer D_i (within a reasonable time window) are likely to belong to the same thread. We refer to the feature that captures the involved developers in a thread as ***user involvement***. For example, as shown in Table 3, the thread *T1* shows how the threads keep on progressing with developers responding without mentioning each other.

2- The similarity of the discussed content: The same-thread messages predictably share related content. For example, if a question is about a button placement in the UI, the answer dominantly contains text that is related to handling the UI. Additionally, as a thread progresses, the thread content (i.e., the topic discussed in the thread) expands, as more follow-up messages are posted. Therefore, the same-threads messages can be identified by assessing the ***textual similarity*** between the consecutive messages in a chatroom. Table 3 shows thread *T1*, where the content of the messages represents a discussion about Ruby and Model-View-Controller (MVC) framework.

3- Back-and-forth communication among users: Another recurring indicator of same-thread messages is the pattern of the discussion. In some cases, an asker never mentions a specific developer, and the exchanged messages do not share similar textual content. Instead, we observe repeated back-and-forth communication between two developers. As shown in Table 3, we can observe this pattern by looking at the message sequence from messages M_1 to M_6. Respondent

Table 5. An example of using our approach to identify threads in a chatroom. Our approach starts with the first message M_1 and assigns the following messages (M_3, M_4, M_5, and M_6) to the same thread T_1 based on metric values of these messages.

Developer	Message	Message bi-grams	Mentions	Involvement	Bi-grams	ThreadID	IG	BG
D_1	M_1	{BG_1}	-	-	-	T_1	{**D_1**}	{**BG_1**}
D_2	M_2	{BG_2}	No	No	0	-	-	-
D_3	M_3	{BG_3}	Yes	No	0	T_1	{D_1, **D_3**}	{BG_1, **BG_3**}
D_1	M_4	{BG_4}	No	Yes	0	T_1	{D_1, D_3}	{BG_1, BG_3, **BG_4**}
D_3	M_5	{BG_5}	No	Yes	0	T_1	{D_1, D_3}	{BG_1, BG_3, BG_4, **BG_5**}
D_4	M_6	{BG_5}	No	No	1	T_1	{D_1, D_3, **D_4**}	{BG_1, BG_3, BG_4 ,BG_5}
D_5	M_7	{BG_6}	No	No	0	-	-	-
D_6	M_8	{BG_7}	No	No	0	-	-	-
D_7	M_9	{BG_8}	No	No	0	-	-	-

D_2 do not mention asker D_1. However, both developers continue discussing the question posted by the asker. The repeated back-and-forth communication between two developers can be used to identify thread T_1.

Step 2: Automatic labeling of the discussion threads in chatrooms. Using the observations derived from the manual analysis step, we propose a set of metrics to identify messages relevant to the same thread. Table 4 shows the collected metrics, along with the rationale for the collection of these metrics. Using the metrics shown in Table 4, we design an approach that parses the messages posted in a chatroom, and automatically groups the messages that belong to the same thread. Our approach (presented in Algorithm 1) starts with a single message (M_1) and inspect the following messages as follows.

1. Tracking the involved developers: Our approach starts by checking if any of the following messages mention the developer D_1 of the initial message M_1. Table 5 shows an illustrative example of our approach. In the message M_3, the developer D_1 is mentioned. So, we added the developer D_3 to the *Involvement Group (IG)*, which contains all the developers participating in the current thread. Next, our approach identifies any message that is posted by the developers of Involvement Group (IG) or if other developer mentions any developer from the IG (e.g., adding the message M_4 to the thread T_1).

2. The similarity of the discussed content: In the next step, our approach checks the content similarity by identifying the matching bi-grams. If there is a matching bi-gram, then we update the Involvement Group (IG) and the Bi-Gram Group (BG), and label the thread with the same ID. The scenario can be seen in Table 5 in the message M_6, where there is a matching bi-gram. It means that the bi-gram BG_5 in the message M_6 is similar to the bi-grams (i.e., BG_1, BG_3, BG_4, and BG_5) in the existing messages of the thread T_1. The message M_5 *"did you read ellismg's reply about **bcl rewriter** ? because that is failing in your build gist ..."* and the message M_6 *"apparently I didnt. Yeah, that's actually what I'm doing (skipping **Bcl Rewriter**) ..."* contain the same bi-gram *"bcl rewriter"* due to which they have the same thread ID.

3. Back-and-forth communication among developers: Our approach identifies that two messages M_i and M_j belong to the same thread, if there is a back-and-forth communication between the developers of the messages M_i and M_j.

Sliding widow for message lookup: Due to the dynamic nature of the chatroom discussions, it is essential to decide a proper window size for the identification of the relevant messages. However, not all threads have the same length (i.e., the same number of messages in a thread). Our initial qualitative study on the Ruby chatroom shows that threads can be of different lengths ranging from two messages to more than 15 messages. If the window is set too wide, the algorithm examines many messages which are not part of the relevant thread. Similarly, if we set the window too narrow, the

Algorithm 1: An algorithm for threads identification

```
Data: List of Messages from chatrooms (M)
Result: List of identified threads with unique IDs
sw = 4                                                  ▷ sw: sliding window size
M.Bigrams = bigrams()                                   ▷ calculate bi-grams for all messages
while Messages do
    curerntMessage                                      ▷ iterate each message one by one
    threadId                                            ▷ unique identifier for threads
    invovlmentGroup = curerntMessage.user               ▷ initialize with the asker
    bigrams = curerntMessage.bigrams                    ▷ bi-grams in the currentMessage
    while sw do
        sMessage                                        ▷ check each message in the sliding window
        mentions = IdentifyMentions(sMessage, invovlmentGroup)
        involved = IdentifyInvolvement(sMessage, invovlmentGroup)
        bigrams = IdentifyBigrams(sMessage, bigrams)
        pattern = IdentifyBackAndForthPairs(sMessage)
        if pattern == 3 then
            labelMessages(sMessage)
            /*Go back and label all the pair of back-and-forth communication*/
        end
        if mentions OR involved OR bigrams then
            LabelThreads(sMessage, curerntMessage, threadId)
            invovlmentGroup.append(sMessage.user)       ▷ add sMessage.user to IG
            bigrams.append(sMessage.bigrams)            ▷ add sMessage.bigrams to BG
            if message labeled in last half of sw then
                sw += 4
            end
        end
    end
end
```

algorithm might miss the rest of a thread if a thread discussion is continued after the sliding window. To address this problem, we empirically set up the window size by calculating the mean size (in terms of the number of messages) of the threads in our initial qualitative study ($Sample_{initial}$). We found that the mean of the number of messages in a thread is four messages. Hence, we empirically setup the lookup window size to four messages. To accommodate the dynamic behavior of threads, we used a *sliding window* concept to capture the long threads as well. If our approach can find any message belonging to a thread in the last half of the current window (i.e., in the last two messages). Our approach slides the window to the next four messages and keeps looking for new messages. As soon as there are no messages relevant to the current thread in the last half of the window, we stop looking for the new messages.

As shown in Table 5, our approach started from M_1. According to our current window, we have to look for M_2, M_3, M_4, M_5. Our approach can find messages in the last half (i.e., M_4 and M_5). So, our approach slides the window to the next four messages, which are M_6, M_7, M_8, and M_9. However, our approach is not able to find any messages (in the last half of the window in this pass), which are related to a thread that started from M_1. Therefore, our approach stops looking for any more messages for the current thread.

4 RESULTS

In this section, we present the results of all of the research questions. We discuss motivation, approach and findings for each of the research question.

4.1 RQ1: What is the accuracy of the proposed approach for identifying a discussion thread?

Motivation. A developer chatroom is a medium of communication characterized by entangled, and possibly simultaneous, discussions (i.e., threads) exchanged among the developers. To benefit from existing discussions, developers can examine the previous discussions in the chatrooms to see if a similar query is already asked or to monitor the ongoing discussions. However, manually tracking discussion threads is a tedious and time-consuming process due to the synchronous nature of chatrooms [64]. Hence, prior studies [20][65][66] propose approaches to extract useful information (e.g., users who are mostly interact together) from chat data. The proposed approaches include correlation clustering, models for automatic summarization, and pattern matching. However, the proposed approaches in prior work are tailored for thread identification of a specific dataset (e.g., teenager chatrooms and student-teacher chatrooms) which makes them not feasible to apply for thread identification in developer chatrooms. The outcome of our approach can help the project maintainers identify the recurring issues, and possibly highlight the areas of improvements in the associated projects.

Approach. We run our thread identification approach on the full dataset of extracted chat messages (i.e., 6,605,248 messages from 709 chatrooms). We conduct an evaluation of the proposed thread identification approach by selecting a statistically representative random sample of 384 threads (Sample_{Eval}) from the identified threads (i.e., 708,294 threads from 709 chatrooms) for manual investigation. Sample_{Eval} is selected with a confidence level of 95% and a confidence interval of 5%. For each identified thread in Sample_{Eval}, the first and third author of this paper (i.e., the evaluators) manually assess the accuracy of the thread identification by looking at the identified thread along with messages that are posted before and after the identified thread. As such, the manual evaluation can also capture the messages missed by the thread identification approach (i.e., the false negatives). We evaluate the accuracy of our approach by calculating the precision, recall and F1-score of Sample_{Eval}.

Precision can be defined as the probability that an object is relevant given that it is returned by the system[26]. As shown in equation 3, we use precision to identify the number of messages of a thread that are correctly identified (TP) over the the number of the wrongly identified messages (FP), based on our manually labeled dataset (Sample_{tuning}).

$$P = \frac{TP}{TP + FP} \tag{3}$$

Recall is defined as a probability that a relevant object is returned by a system [26]. As shown in equation 4, we use recall to identify the number of messages of a thread that are correctly identified (TP) over the number of the missed messages (FN) based on our manually labeled dataset (Sample_{tuning}).

$$R = \frac{TP}{TP + FN} \tag{4}$$

F1-score is the weighted average of the precision and the recall which includes the impact of the false positive as well as false negatives. Hence, we use F1-score to evaluate the overall accuracy of our approach for our manually labeled dataset (Sample_{tuning}), as shown in equation 5.

$$F1 = 2\frac{P.R}{P + R} \tag{5}$$

To access the agreement between the evaluators, we use Fliess' Kappa [22]. Fliess' Kappa is a statistic to measure agreement among two or more evaluators for categorical items (e.g., if a message belongs to a thread or not). A higher value of Fliess' Kappa means a strong agreement between the evaluators, with a maximum possible value of 1 showing complete agreement.

Results:

The content of 81% of the analyzed threads is correctly identified by the proposed thread identification approach. We calculated the precision, recall, and F1-Score for the selected sample. We observe that our approach achieves the precision score of 0.86, recall of 0.74, and F1-Score of **0.81**. This result shows that our approach can find the threads from the chatrooms with good F1-Score. We also calculate the Fliess' Kappa to be 0.93, which shows a strong agreement between the two evaluators. After the evaluation, we performed an error analysis to identify the scenarios where our algorithm incorrectly labels messages as part of a thread or misses messages which are part of a thread. For example, if a developer communicates with two developers in two threads T_1 and T_2 at the same time, our algorithm incorrectly labels the responses to one developer as part of the thread of the other developer. However, in this scenario, the concerned messages are labeled in both the threads T_1 and T_2. As a result, we do not lose any valuable information in a respondent's messages. In addition, our algorithm is not able to label the messages outside the sliding window. Usually, it is only one message that is missed by our algorithm. Either the asker or the respondent posts this message after some time from their initial discussion.

62.7% of the posted questions triggered developer discussions. Our approach identified 708,294 threads that contain 6,605,248 messages from 709 chatrooms. Using our approach, we observe that 420,857 messages are not involved in any thread discussion. In the next research questions, we further examine the responded and not responded threads and try to identify the significant features.

Summary of RQ1

Threads can be identified from the text messages in chatrooms using metrics from users, content, and back-and-forth communication. The evaluation of our proposed automatic labeling approach shows that our approach accurately identifies discussion threads with 0.81 F-Score.

4.2 RQ2: What makes a question getting responses in developer chatrooms?

Motivation. In RQ1, we evaluate our proposed approach to automatically identify threads in chatrooms. Based on the results of the thread identification approach, we observe that a large number of queries (approximately 40%) remain unanswered. The high rate of unanswered queries could hinder the problem resolution process for the developers who seek answers and possibly make serious issues to be unnoticed by the project maintainers. Therefore, in this research question, we study the relationship between the characteristics of the chat messages initiating a thread, and the developer's response behavior, such as the speed of getting a response. The result of this analysis can help project maintainers design general guidelines for developers to compose queries with more likelihood of faster engagement from other developers. As chatrooms belong to different domains, the response behaviors can be different from a chatroom to another. Hence, we also study the patterns of response behaviors (i.e., clusters of chatrooms that have similar response behaviors).

Approach. In this section, we study the relationship between the characteristics of messages initiating threads, and the response behaviors from other developers. In particular, we study two different types of response behaviors: 1) the posted messages that engage other developers to respond (Behavior$_{engaged}$), and 2) the speed of the thread discussion either fast or slow (Behavior$_{speed}$). Our approach consists of three steps: 1) feature collection, 2) model building, and 3) model analysis. We describe the details of each step as follows.

Step 1: Features collection: In the first step, we collect the features that may have a relationship with the response behaviors. We identify two categories of features: *A) message features* that describe the posted messages; and *B) user features* that characterize the authors of the messages. Table 2 lists the collected features, along with their types (i.e., categorical or numerical), and the rationale for the selection of each feature.

Step 2: Model building: In this step, we build models to analyze the relationship between the collected features and the response behaviors (i.e., Behavior$_{engaged}$ and Behavior$_{speed}$) as follows.
a) Removing correlated and redundant features: To avoid the possibility of correlated features interference with our interpretation of the models, we remove the highly correlated features [59]. We use the Spearman rank correlation test (cut-off value for p is 0.7) to calculate the correlation between the features [30]. We run `varclus` package from `Hmisc R` [34] to construct a hierarchical overview of the inter-feature relationship. We find that the GitHub features that includes *issues, commits, reviews, pull requests*, and *active GitHub contributor* are highly correlated. Hence, we remove the *commits, issues, pull requests* and *reviews* features and we retained the *active GitHub contributor* feature.
b) Data selection for two models: We study two different behaviors of responses in chatrooms: (1) analyzing whether the posted messages get responses (Behavior$_{engaged}$) and (2) analyzing messages with fast or slow responses (Behavior$_{speed}$). As the goal of both behaviors is different, the data selected for both models are also different.
Data for Behavior$_{engaged}$ analysis. Our goal is to analyze the features that determine whether the message posted in a chatroom gets a response. For this purpose, we include all the messages with responses and messages without responses. For the messages with responses, we have a total of 708,294 messages with responses as these messages initiated the identified 708,294 threads. Based on RQ1 results, we find 408,257 messages without responses, as these messages did not trigger any threads. Hence, we collected features for a total of 1,116,551 messages.
Data for Behavior$_{speed}$ analysis. We analyze the messages which get fast or slow responses. For this analysis, we have to use only the messages with responses (i.e., 708,294 messages). We first calculate the time difference between the first response and the initial asker's message in a thread and then calculate the four quantiles of the time difference. Wang *et al.* [68] analyzed whether a question gets a fast response in Q&A forums. Wang *et al.* [68] identified fast and slow responses by dividing the response time in the Q&A dataset into four quantiles and considering the 1^{st} and 4^{th} quantile of time differences as slow and fast responses respectively. We use a similar approach as our 1^{st} and 4^{th} quantiles contain the slow responses (i.e., 177,073 messages) and the fast responses (i.e., 177,073 messages) making a total of 354,146 messages in the dataset for the behavior$_{speed}$ model.

c) Building the mixed-effect models: As the threads are identified from different chatrooms, every chatroom may have different response behavior. To study the response behaviors with the chatroom context into consideration, we use a mixed-effect model [62]. A mixed-effect model includes two types of features, explanatory features, and context features. The explanatory features (user features and message features) are used to explain the data (i.e., chat messages), while context features refer to the chatroom ID. A mixed-effect model shows the relationship between the outcome (i.e., message responded or not in Behavior$_{engaged}$ and discussion fast or not in Behavior$_{speed}$) and the explanatory features (user features and message features), while taking into consideration the context variables (chatrooms). It is

important to note here that we construct two separate models for Behavior$_{engaged}$ (message responded or not) and Behavior$_{speed}$ (discussion fast or not). In our study, we construct the mixed-effect models using the `glmer` function of the `lmer` package of R [6].

Step 3: Analyzing the constructed models: The discriminative power of the model measures the ability of the model to discriminate between the goal (i.e., messages getting a response and fast discussion). We calculate the discriminative power using the Area Under Curve (AUC) [29]. A plot of true positives against the false positives for different thresholds is done using Receiver Operator Curve (ROC). AUC value ranges from 0-1, 0 being the worst performance, 0.5 random guessing performance, and 1 is the best performance [29].

We use Wald statistics [36] to estimate the relative contribution (χ^2) to understand the impact of the user features and the message features. A higher value of χ^2 shows a high impact of the feature on the performance of the model [30]. We used `car` package [24] in R which provides the implementation of the `Anova` to calculate the Wald χ^2.

Step 4: Identifying developer's response patterns: To identify clusters of chatrooms that share a common response behavior (Behavior$_{engaged}$ and Behavior$_{speed}$), we follow the same approach as Hassan *et al.* [30]. In particular, we build a logistic regression model using `glm` function provided in `stats R` package [23]. The logistic regression model shows the likelihood of the message getting a response and fast discussion based on user and message features. We construct a model for each chatroom. Then, we investigated each model by extracting *a) the percentage of importance* and *b) sign of slope* for each feature.

a) Percentage of importance: To calculate the percentage of importance, we first calculate the relative contribution (χ^2) of each feature using Wald statistics. Then, we calculated the percentage of χ^2 for each feature to the total χ^2 for all the features for each chatroom.

b) Sign of slope: To examine the direction of the relationship between features and response behaviors, we identify the sign of the slope for each model.

We use the extracted key features for all the models of the chatrooms as input to the clustering implementation of `kmeans` function that is provided in `stats R` package [23]. To identify the optimal number of clusters, we started with the two clusters (the minimum number of clusters) and an increased number of clusters until the new clusters are sub-cluster of the previous run.

Results:

Inactive chatroom participants are more likely to get a response. Table 6 shows the significance of the user and the message features with respect to the questions getting responses. The *active chatroom participant* feature accounts for the largest χ^2 value in the model which indicates that *active participant* contributes the most to our model (Behavior$_{engaged}$). However, the negative relationship represents that the possibility of getting a response increases for less active chatroom participants in the chatrooms. A recent study by Chhabra *et al.* [15] also shows that the inactive askers are responsible for the extraction of useful contributions from active users. Due to the fact that a lot of developers visit chatrooms only to ask questions while more active developers are involved in responding to the posted questions. Surprisingly, the experience in GitHub (represented by the *active GitHub contributor* feature) has the least significant effect on getting a response. In the following discussion, we only investigate the most significant features that have an effect on the possibility of getting a response (based on χ^2), and the rest can be interpreted from Table 6. For example, code snippets and edited questions have a positive impact on getting a response. In addition, CLI (readability) has a negative impact on getting a response, which means that easy-to-read questions are most likely to get responded.

Table 6. Results of the mixed-effect model for questions getting responses or not (Behavior$_{engaged}$) - sorted by χ^2 descendingly.

Factor	Coef.	χ^2	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	$3.83e^{+00}$	690	$<2.2e^{-16}$	***	↘
active chatroom participant	$-4.06e^{-01}$	5751	$<2.2e^{-16}$	***	↘
URLs	$-2.12e^{-01}$	1699	$<2.2e^{-16}$	***	↘
user mentions	$2.59e^{-01}$	1261	$<2.2e^{-16}$	***	↗
message singularity	$6.44e^{-02}$	538	$<2.2e^{-16}$	***	↗
code snippets	$1.22e^{-01}$	288	$<2.2e^{-16}$	***	↗
edited	$-1.24e^{-01}$	175	$<2.2e^{-16}$	***	↘
CLI	$4.06e^{-04}$	37	$<1.4e^{-09}$	***	↗
issue mentions	$-2.11e^{-01}$	15	$<8.6e^{-05}$	***	↘
weekdays	$-1.87e^{-02}$	7	0.010	*	↘
daytime	$1.18e^{-02}$	5	0.018	*	↗
active GitHub contributor	$1.15e^{-02}$	4	0.038	*	↗
lexicons	$1.29e^{-04}$	2	0.212		↗

$^+$Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Including URLs in the questions has a negative impact on getting responses. The result of the mixed-effect model of the Behavior$_{engaged}$ shows that *URLs* have a significant impact on getting a response. Prior studies show the importance of including the URLs in raised questions to provide more details about the raised questions [11] [50][63]. To investigate further, we select a statistically representative random sample of 96 threads (with a confidence level of 95% and a confidence interval of 10%) with URLs. We find multiple cases where askers post questions that contain URLs with little details, and the posted question did not get a response. For example, a developer asked *"anyone seen this issue before (external link)"*. One possible explanation for these questions being unanswered is that developers may be hesitant to open such links which do not contain any details associated with them. However, we do not conclude that askers should not include URLs in their posted messages. We further investigate the inclusion of URLs in the messages with clear details about the included URLs. For example, a developer asked *"Hey. I am using a Wordpress theme called Clarion on my website which is based on the Gantry framework. I am currently struggling to get the layout to work on the homepage. Can anyone help? (external link) The main section under the header has an extra column on the right which I can't get rid of?"*. In this example, we observe that the asker includes specific details along with the URL, which is easier to understand for the respondents. Hence, we recommend that askers should include the appropriate details while including the links (if needed).

We also investigate the messages where askers include all the details in the messages. For example, an asker posted a message *"Hi, Is there any way to set the* `$core-font-size` *and* `$menu-font-size` *in the custom.scss ? or how can I set them? Overriding the blueprint base.yaml ?Thank you"*. We can observe from this example that concise details included in the posted message increase the chances of getting a response.

Table 7. Results of the mixed-effect model for fast discussions (Behavior$_{speed}$) - sorted by χ^2 descendingly.

Factor	Coef.	χ^2	$Pr(<\chi^2)$	Sign.$^+$	Relationship
(Intercept)	-2.77e^{+00}	864	<2.2e^{-16}	***	↗
weekdays	9.64e^{-01}	4607	<2.2e^{-16}	***	↗
active chatroom participant	-7.14e^{-01}	4437	<2.2e^{-16}	***	↘
message singularity	-3.82e^{-01}	2976	<2.2e^{-16}	***	↘
URLs	-6.60e^{-01}	1501	<2.2e^{-16}	***	↘
lexicons	-1.53e^{-02}	1451	<2.2e^{-16}	***	↘
daytime	3.60e^{-01}	1297	<2.2e^{-16}	***	↗
user mentions	-3.44e^{-01}	577	<2.2e^{-16}	***	↘
code snippets	-1.87e^{-01}	158	<2.2e^{-16}	***	↘
edited	-2.59e^{-01}	152	<2.2e^{-16}	***	↘
active GitHub contributor	2.27e^{-05}	35	3.1e^{-09}	***	↗
issue mentions	-6.76e^{-01}	18	2.2e^{-05}	***	↘
CLI	-3.59e^{-04}	13	0.0002	***	↘

$^+$Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Messages containing user mentions have a positive effect on getting a response. The model results for Behavior$_{engaged}$ also represent a significant relation between user mentions in getting a response. The developers (askers) who ping other developers in the posted messages have a high chance of getting a response.

Unique questions posted by inactive askers during weekdays are more likely to have a quick discussion. Table 7 shows the significant features with regards to the fast discussions in chatrooms. We observe that weekdays and inactive chatroom participants are the most impactful features on Behavior$_{speed}$. In particular, the questions asked on weekdays and from inactive developers have a significant effect on making the discussions fast. This is understandable because developers are less likely to be active on the weekends and more active on the weekdays [25][60].

Other notable features with a significant impact on the fast discussion are unique questions, URLs, and the number of words in messages. We observe that the unique questions may result in a longer discussion than the average discussion time, as sometimes respondents ask for more clarification and try to understand the described scenario. We also find that including URLs in the messages might take more time than the average discussion time because respondents need to visit the URLs first, and hence discussions take more time. Finally, we observe that longer messages can take a longer time to discuss because they include a lot of details.

We identify general guidelines for chatroom askers to prepare high-quality questions as follows:

- Write concise questions with code snippets (if needed) and provide the details that are easy to read.
- Include URLs in your questions with sufficient details, avoid posting only the URLs in a message without a descriptive context.
- Edit the original questions if the respondent asks for clarification. Adding missing details can help other developers correctly understand and resolve the raised questions.

We identify the implications for project maintainers as follows:

- Provide customized guidelines to askers by providing examples of the well-crafted questions in the chatrooms.

Table 8. Summary of the identified patterns of responses. For each pattern, we present the percentage of importance of the collected features.

Feature	Behavior$_{engaged}$ patterns				Behavior$_{speed}$ patterns		
	P1(113)	P2(83)	P3(77)	P4(26)	P1(110)	P2(80)	P3(72)
active GitHub contributor	3%(-)	5%(+)	3%(+)	2%(-)	5%(+)	6%(-)	7%(+)
active chatroom participant	**26%(+)**	8%(+)	**53%(-)**	12%(+)	10%(+)	12%(+)	**55%(-)**
CLI	4%(-)	3%(-)	3%(+)	2%(+)	1%(-)	4%(+)	1%(-)
code snippets	7%(+)	6%(-)	11%(+)	3%(-)	8%(+)	2%(+)	3%(-)
daytime	6%(+)	3%(-)	2%(+)	5%(-)	**29%(-)**	**22%(+)**	2%(+)
edited	2%(-)	6%(+)	1%(-)	5%(+)	3%(+)	3%(-)	5%(+)
issue mentions	1%(+)	4%(+)	1%(-)	2%(+)	1%(-)	1%(-)	2%(+)
lexicons	**21%(-)**	**23%(+)**	6%(+)	7%(-)	**22%(+)**	11%(+)	8%(-)
message singularity	11%(+)	**17% (-)**	5%(+)	2%(-)	2%(+)	**20%(-)**	5%(+)
URLs	5%(+)	9%(+)	6%(+)	4%(-)	4%(+)	9%(+)	3%(+)
user mentions	9%(+)	10%(+)	7%(+)	**54%(+)**	11%(+)	6%(+)	4%(-)
weekdays	5%(-)	6%(+)	2%(-)	2%(+)	4%(-)	4%(+)	4%(+)

- Develop a chatbot that recommends the most relevant threads to a newly posted question or suggests the best respondents who answered similar questions. Hence, askers can mention the recommended developers to get a response.
- Address the most recurring questions by grouping them as FAQs in order to reduce the number of unanswered and repeated questions.

Different chatrooms have a different possibility of getting a response and fast discussions. Our analysis shows that the random intercept of both mixed effect models varies across different chatrooms, meaning that every chatroom has a different possibility of getting a response. The result indicates that if a question is posted in a chatroom, every chatroom may behave differently. Moreover, the discussion time varies across the different chatrooms, based on the random intercept analysis. The discriminative power (AUC) of questions getting a response (Behavior$_{engaged}$) is 0.75, while AUC for fast discussion time (Behavior$_{speed}$) is 0.93. As the chatrooms have a different possibility for response behaviors, we perform a deeper investigation to identify the patterns of the response behaviors.

We identified four different patterns of respondents to the raised questions. Percentage of the importance of each feature in the four clusters are described in Table 8. The four clusters are described as follows:

(1) **Respondents who mainly respond to an active asker who posts short messages.** In the first cluster that contains 113 chatrooms, respondents respond to the active askers who post short messages. The mean value for the importance of active members is +26%, while the length of messages is -21%. We manually investigate the messages from the cluster and find that active askers ask about different features of the project and get responses from respondents.

Implications for askers. Askers should write short and concise messages in the chatrooms.

(2) **Respondents who mainly respond to long and non-unique messages.** In the second cluster that contains 83 chatrooms, respondents respond to long but simple messages. The mean value for the importance of the length of the message is +23%, and for the message, the singularity is -17%. Our manual investigation shows that the messages or questions in the cluster are usually descriptive, an asker explains the scenario and the solutions already tried. However, the questions are related to each other and are not unique.
Implications for chatroom askers. Askers should focus on explaining the problems and adding further details if needed.

(3) **Respondents who mainly respond to inactive members.** In the third cluster that contains 77 chatrooms, respondents respond to the inactive askers. The mean value for the importance of the active asker feature is -53%. The respondents respond to the messages posted by the askers. It is encouraging to see that chatrooms in this cluster tend to help the beginners and novice developers.
Implications for chatroom askers. Askers should not hesitate to ask well-crafted questions if they are new to the chatrooms.

(4) **Respondents who mainly respond to mentions.** In the fourth cluster that contains 26 chatrooms, the askers mention respondents who frequently respond to the queries in the chatroom. The mean value for the importance of the mentions is +54%.
Implications for chatroom askers. Askers should review recent discussions and analyze the developers who are more active in responding to questions and mention such developers when asking the questions.

We find three clusters that represent the common patterns for developers (respondents) involving in fast discussions based on the associated features. Percentage of the importance of each feature in the three clusters are described in Table 8 and can be explained as follows:

(1) **Respondents who mainly participate in quick discussions on long messages during non-office hours.** In the first cluster that contains 110 chatrooms, askers and respondents have quick discussions on long messages from 05:00 p.m. to 09:00 a.m. The mean value for the importance of the daytime is -29%, and the length of the message is +22%. We further investigated the messages in the cluster and find that the asker posts their queries after 05:00 p.m. and gets a quick discussion. Respondents in the cluster respond to the questions at that time faster than during the daytime.
Implications for chatroom askers. Askers should ask long questions during non-office hours in the chatrooms.

(2) **Respondents who mainly participate in quick discussions during office hours on simple questions.** In the second cluster that contains 80 chatrooms, askers and respondents have quick discussions on simple messages from 09:00 a.m. to 05:00 p.m. The mean value for the importance of the daytime is +22%, and the singularity of the message is -20%. As opposed to the first cluster, respondents in the second cluster tend to have a quick response to the queries during office hours. We notice that the questions posted by askers are simple and easy to understand.
Implications for chatroom askers. Askers should ask simple questions during office hours for a quick discussion in the chatrooms.

(3) **Respondents who mainly participate in quick discussions with inactive members.** In the third cluster that contains 72 chatrooms, respondents have quick discussions with inactive askers of the chatrooms. The mean value for the importance of the active chatroom participant is -55%.

Implications for chatroom askers. New askers in the chatrooms should ask concise questions to get a quick response.

Summary of RQ2

Providing details in a message rather than external links are more likely to get a response. We also find that questions posted by inactive askers are more likely to be responded by the members of the chatroom. Each chatroom has different behavior in terms of responding to the queries posted. In particular, we identified four patterns of respondents. Understanding the nature of respondents (i.e., response patterns) can help project maintainers maintain guidelines for askers to shape their questions in a way that can lead to a higher probability of getting responses. Askers can also be aware of the response behaviors of the chatroom in terms of the time of asking a question, speed of response, and complexity of the question.

4.3 RQ3: What features show an association with the resolution outcome of the discussion threads?

Motivation. In the previous RQ, we study the features that are associated with the response behaviors (i.e., the engagement from other developers, and the speed of interactions). However, getting responses from other developers on a raised question does not necessarily lead to the question to be properly answered or resolved. Understanding the features that may impact the resolution outcome of a raised question is important for many reasons. First, writing questions that are more likely to get resolved would attract answers with higher quality, which boosts the reputation of the associated project. This, in turn, is likely to encourage experts to spend more time in the chatroom, seeking questions that challenge their expertise.

Approach. In this research question, we build a model to study the association between the message features (e.g., the discussed topic in the message), and the resolution outcome and the type of the threads. The resolution outcome represents whether the issue raised is resolved. The type of resolutions indicates how developers resolve the raised issue (e.g., suggesting a tutorial to follow and fixing the code). Since it is challenging to automatically identify the outcome and the type of resolution, we manually identify resolution type and outcome using a statistically significant sample of the identified threads. In addition to the features described in Section 3 (i.e., *message features and user features*), we used the manually identified topics as well to build the model. Our approach involves the following three steps.

1) Selecting a statistically representative random sample of 384 discussion threads. In RQ1, we identify 708,294 threads from the chatrooms. Manually labeling all the identified threads is a time-consuming and tedious process. Due to this reason, we select a statistical sample from the identified threads. We use a confidence interval of 5% and a confidence level of 95%, which results in 384 threads. We use the randomly selected 384 threads ($Sample_{threads}$) for manually labeling the topics and identifying the resolution of the studied threads.

2) Identifying the topic, resolution type, and resolution outcome of every thread. The goal of manually labeling the threads is two-fold, first is to identify the resolution outcome and type of the thread and second to examine the topic associated with the thread. We aim to answer the following two questions.

- *What is the resolution outcome and type of a discussion thread?* In this question, we first investigate if there is enough evidence that the raised question is resolved. We use phrases like *"Thanks, that works"*, *"Sure, I will try this"* to confirm whether the thread is resolved. We also looked for a detailed and comprehensive answer

Table 9. Topic categories of the discussion threads in the studied chatrooms.

Category Name & Description	Count	Example
Phase A: Beginner-Level Questions		
1) Getting started with a project		
Description: Threads discussing beginner-level questions, asking about the initial learning material.	23	**Asker::** *"Hi, I'm trying to get started with Angular 2.0 Sorry, but all the typescript in the get started guide seems like just noise. Are the examples without typescript?"* **Respondent:** *"here is the getting started guide using pure JS "*
Phase B: Raising How-to Questions		
2)How to install & configure		
Description: Threads discussing the problems of installation and configuration.	26	**Asker:** *"Windows 10/Vagrant/Virtualbox/ Homestead. Has anyone successfully gotten Mix working in this environment? I'm having the hardest time installing and configuring them"* **Respondent:** *"this might help (NodeJs on Windows) doesn't need to sync, vagrant already symlinks your folders "*
3) How to implement a coding task		
Description: Threads discussing about the implementation guidance (e.g., asking for a specific project feature implementation).	161	**Asker:** *"I am not able to get the default value = 'Please Select', when the dropdown is null "* **Respondent:** *"try true: null , don't put them into strings "*
4) How to solve a bug		
Description: Threads discussing about code with bugs, Askers post their bugs and usually respondents fixes the code.	69	**Asker:** *"hi, a list ,its content as below:"* `val a = List(""aa"",'1'),(""bb"",'2'))` *"I expected the following output: "* `List(List((""aa"",'1')),List((""cc"",3)))` *"now i using this method:"* `a.splitAt(a.size-1)` *"what method I should use to get my expected value?"* **Respondent:** `a.map(t => List(t._1, t._2))`
5) How to build & compile		
Description: Threads discussing the issues that happen while building/compiling the project code.	31	**Asker:** *"Hi guys, every time I run 'sbt publishLocal' for ensime-sbt, it would publish a non-deterministic version like '1.12.4+20161210-1158-SNAPSHOT', then I need to change my project plugins.sbt to use it, can I make this version string fixed instead of using the current timestamp?"* **Respondent:** *"you could set 'version := "whatever"' locally and that'll win"*
Phase C: Engaging in Advanced Discussions		
6) Enhancing your code		
Description: Threads discussing about enhancing the code/project such as improving the maintainability of their code.	6	**Asker:** *" hello everyone, need some advice about localStorage/sessionsstorage what is better ?this ngStorage or this Local Storage,I want to make my storage efficient."* **Respondent:** *"performance wise I would say the latter. ngStorage works with a watcher and events, and it actually compares itself by creating deep copies of itself"*
7) Discussing project intricacies		
Description: Threads discussing the details of the project, usually project features.	63	**Asker:** *"What is 'Portal' for? Is it related to overlays? Is it like a mini-router you control dynamically? What are some of it's uses* **Respondent:** *"its sort of a mini-router that you can control dynamically, you can use it for dialogue overlays. portal"*

provided by the respondent. For the resolved questions, we identify the resolution type (e.g., the answer provides fixes of the raised coding issues).

- *What is the discussion topic of a thread?* In this question, we look at the topic of a discussion thread. We assign a label to each thread that describes the topic of the discussion (e.g., installation).

To calculate the accuracy of the identified resolution outcomes and topics of the threads, two evaluators (the first and the third author of this paper) manually labeled the randomly selected Sample threads (Sample$_{threads}$). To assess the agreement between the evaluators, we used Fliess' Kappa [22], as explained in RQ1.

We group the topic labels into topic categories. The topic categories help understand the diverse range of topics discussed in the chatrooms and reflect the difficulty of the questions.

3) Analyzing the effect of features on the resolution outcome and type. We construct a mixed-effect model for resolution outcome based on Sample$_{threads}$. Then, we analyze the discriminative power (AUC) of the model. Besides, we use Wild statistics test [36] to estimate the relative contribution(χ^2), to understand the impact of all the studied features on resolution outcome.

Results:

We observe that developers discuss seven topics in the studied chatrooms. Table 9 shows the identified topics based on analyzing Sample$_{threads}$. Our focus for the topic identification process is to categorize the relevant threads for developers and project maintainers. We found that almost 94% of the threads are useful for either developers or project maintainers. In particular, 24 out of 384 threads are labeled as non-informative and are not part of any topic. These non-informative threads include discussions about birthday greetings, game buying discussion, and laptop specifications discussions. Hence, we excluded such 24 non-informative threads from our analysis. It should be noted that the discussion thread can belong to multiple topics. Thus, we label such discussion threads with all the discussed topics.

For the 360 informative threads, we group the identified topics into three categories that reflect the difficulty level of the posted questions: *A) Beginner-level questions, B) Raising how-to Questions and C) Engaging in advanced discussions.* In phase A, beginner-level developers ask questions about the tutorials or any learning material about the project. In phase B, questions and discussion are mostly raised by the developers who gain more knowledge about the project (e.g., *"Hi. What to do with CakeException: MSBuild: Could not locate executable? I'm just trying to execute msbuild"*). In phase C, developers discuss advanced topics related to improving the project, as well as discussing different features in the project. The Fliess' Kappa is 0.88, which shows a strong agreement between the evaluators.

80% of developers post and discuss How-To questions in the studied chatrooms. We observe that 80% (i.e., 287 out of 360) of the informative threads belong to phase B, where developers ask "How-to" questions. Zagalsky *et al.* [71] show that the R community raises "How-to" questions more frequently than other questions in Stack Overflow and other online platforms. In our studied threads, developers raise questions about the installation, configuration issues, bugs, compilation issues in phase B. However, 60% of the threads in phase B discuss the implementation details. Askers either want to know how they can incorporate project features into their application or present their ideas and ask for the related features of the project to use.

We also find that 90% of the threads in phase C are related to the intricacies of the project. In particular, the askers involved in the threads associated with phase C want to grasp the concepts behind the newly added features of the project. We also observe that the project intricacies topic includes developer discussions about the pull request reviews, and discusses the issues reported in GitHub.

Table 10. The identified resolution types of the studied discussion threads.

Category Name & Description	Count	Example
Resolved Thread Types		
1) Suggest Reading *Description*: The respondent suggests the asker read about a particular topic and shares a link to a blog, documentation, or an article.	106	**Asker:**: *"Hi. What to do with CakeException: MSBuild: Could not locate executable? I'm just trying to execute msbuild"* **Respondent:** *"I would suggest using the VSWhere tool, as where 2017 is installed isn't predictable anymore MS build VS Support "*
2) Provide textual explanation *Description*: The respondent replies to the asker by answering the posted question; this does not include any code fixes or suggested reading.	149	**Asker:** *"Was even SQL Server tested by Jenkins? I was under the impression of no. I believe if the "modified factory" succeeds in loading the dlls, the tests and other things should succeed too. That much can be checked by downloading the Nuget packages"* **Respondent:** *"since we are not touching the .SQL files, there is no point on the test each one of the DB engines... any one of those can validate the whole SQLUtils, since the code dont fork to specific implementation and the whole code is (very well) implemented on top of ADO abstractions"*
3) Propose code fixes *Description*: The respondent takes a look at the code posted by the askers, and fixes the code or points out the bug in their code.	68	**Asker:** *"I am not able to get the default value = 'Please Select' when the dropdown is null "* **Respondent:** *"try true: null , don't put them into strings "*
4) Propose project improvements *Description*: Upon considering the issue posted by the asker, the respondent suggests a project improvement, this can either be an improvement in the documentation or creating an issue in GitHub.	9	**Asker:** *"i've managed to get cas 3.5 and 4.0 running - the documentation is not very clear. i'm generally clueless about java/maven so what can i do to learn more about that so i can be more useful in configuring/managing CAS"* **Respondent:** *"If u find particular areas in the docs where something is unclear please let us know and we'll try to clarify as best as we can."*
Unresolved Thread Types		
5) No follow-up from the asker *Description*: The respondent looks for more clarification after the asker's initial query, but the asker becomes unresponsive, due to which the thread is unresolved.	15	**Asker:** *"I have a sidebar on my page that I want to stuff with data when items on the page are selected. I have my data in a ngrx store. I was curious what the best way was to stuff or feed dynamic content to the element/component"* **Respondent:** *"Can you share the sample structure of data and Can you get away with ngIf"*
6) No follow-up from respondent *Description*: The respondent asks for clarification and the asker provides the requested clarifications. Then, the respondent becomes unresponsive, which makes the thread unresolved.	33	**Asker:** *"for some reason I get 8 error screenshots for 100% passed test image "* **Respondent:** *"any chance you have error handling "* **Asker:** *"umh.. you mean this? -> logLevel: 'error' "*
7) Referral to other developers *Description*: The respondent is not aware of the query posted by the asker, rather he/she refers the asker to contact other developers.	4	**Asker:** *"Could you help me please? I downloaded dash engine from github a tried to compile sample project. But it throws error. (Dash/dash.lib(output) Error 42: Symbol Undefined coverage. Am I building a wrong target in Mono-D"* **Respondent:** *"I don't have a lot of experience building with Mono-D, so if you have any other issues, with building, ping @userX"*

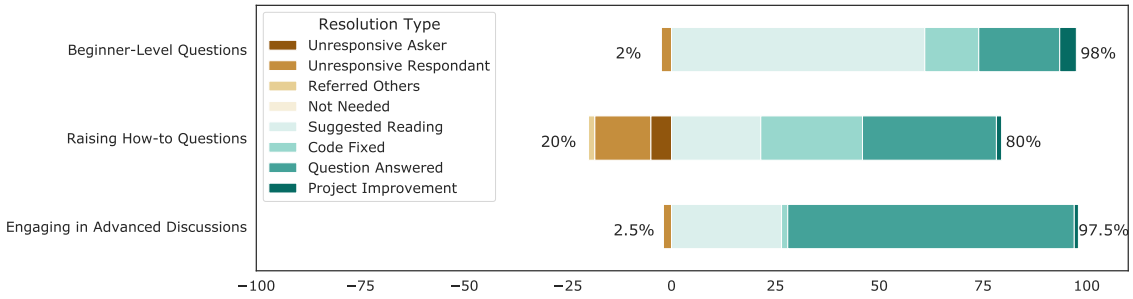

Fig. 4. The percentage of resolved or unresolved threads in each topic category. The right side (i.e., shades of green) presents the resolved and left side (i.e., shades of brown) presents the unresolved threads).

To further investigate threads that discuss code reviews, we searched for the threads with the text “pull request | PR" and the text “review" in their content. We find that the number of such threads is 2,081. We identified a statistically representative random sample of 92 threads (with a confidence level of 95% and a confidence interval of 10%). Our analysis shows that 81% of the studied threads discuss the pull requests review related to the projects. We also find that 245 out of 709 chatrooms have such messages. However, we notice that only 41 chatrooms have more than 10 messages related to code reviews. The obtained results show that the frequency of threads that discuss the pull request review is different from chatroom to another.

To identify the duplicate threads, we analyzed a statistically representative random sample of 94 threads. For every asker message, we searched for duplicate (identical) messages in the same chatroom. We found that the raised questions in 3 threads (3% of the studied threads) are duplicate with other messages in the chatroom. On further investigation, we found that askers post messages. If the askers do not get any response, they post the message again, and eventually, other developers respond to the asker.

80% of the studied threads are resolved. We observe that 323 threads out of the 384 studied threads are resolved. The two possible outcomes of a thread are resolved or unresolved. Within each resolution outcome, we identify the different resolution types, i.e., how the issue in the thread is resolved for the resolved threads, or what prevented the issue from being resolved for the unresolved threads. We list the identified resolution types in Table 10. In our labeling, we find four unique types of resolution in the threads and three types of unresolved threads.

Respondents provide textual explanations and suggest readings in resolving the studied threads. We observe that providing a textual explanation (149 out of 323 resolved threads) is the most common type of resolving the raised questions. We also find that in the resolved discussions, respondents usually suggest external reading (106 out of 323 resolved threads), such as the project documentation, articles, blogs, or sharing links to Q/A websites where a similar query is resolved. An interesting fact that we notice is when an asker posts a code query related to a bug, in more than 80% of the cases, respondents fix the asker’s code.

90% of the unresolved threads are due to no follow-up by the askers or the respondents. We observe that for 48 (out of 52) of the unresolved threads, the askers or the respondents have no follow-up (i.e., do not proceed with the discussion). For example, the asker or the respondent asks for clarification about the posted messages, and no response is provided by the author of the posted message.

Advanced discussions and beginner-level questions are more likely to be resolved. Figure 4 shows the analysis of topic categories and their corresponding resolution types. The right side of the plot shows the threads that

Table 11. Results of the mixed-effect model for threads getting resolved or not - sorted by $< \chi^2$ descendingly.

Factor	Coef.	χ^2	$Pr(< \chi^2)$	Sign.$^+$	Relationship
(Intercept)	3.25	6.89	0.008	**	↗
topic	8.29	7.88	0.019	**	↗
edited	1.63	2.41	0.119		↗
TF-IDF	0.32	1.82	0.176		↗
active chatroom participant	0.33	0.93	0.334		↗
active GitHub contributor	-0.30	0.75	0.385		↘
URLs	-0.40	0.60	0.435		↘
CLI	0.00	0.43	0.511		↗
daytime	-0.17	0.25	0.614		↘
lexicons	0.00	0.02	0.878		↘
user mentions	0.08	0.01	0.915		↗
weekdays	-0.02	0	0.968		↘
code snippets	0.00	0	0.987		↗

$^+$Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

are resolved, and the left side shows the unresolved threads. It is evident from the figure that phase A (i.e., getting started) and Phase C (i.e., project intricacies and enhancing code) threads have a higher resolution percentage (98%) than phase B questions. Although phase B has lower % (80%) of resolution than other categories, respondents provide details to resolve the raised questions. Project maintainers can leverage the provided question-answer pairs in the threads discussions to enrich the documentation for their project (e.g., improving the FAQ of their project).

The topic category of a thread discussion is significantly associated with the resolution outcome of the thread. Table 11 shows topic category is the most influential feature. Editing messages is another interesting feature in the model. Usually, the respondents responding to questions ask for clarification and an asker either posts a new message to clarify or edit the posted message. We observe that editing the messages makes it more likely for the thread to be resolved, adding missing information in questions help respondents answer better. We further investigated the threads which are edited and find that 25 out of 26 edited threads get resolved. We examine the threads associated with the edited messages and observe that respondents ask for clarification, and an asker adds the information in the original message by editing the message. Writing a well-articulated question with the needed details can improve the chatroom platforms and help respondents better support the projects related queries.

Summary of RQ3

Chatrooms developers mostly post *How-to questions*, which includes installation, compilation, implementation inquires, and solving bugs. Around 80% of the studied threads posted in the chatrooms are resolved. In particular, advanced and beginner-level questions are highly likely to be resolved. The examination of the message features that are associated with a question being resolved shows that the topic of the question and edited questions have a significant effect in resolving threads.

5 IMPLICATIONS

In this section, we discuss the implications of our study for developers using chatrooms and developers maintaining the open-source software projects. We provide recommendations to the developers to improve the quality of the interaction within the development chatrooms and better maintain the associated projects.

Chatroom askers: Askers always want to get fast and efficient responses to their queries. To obtain the desired result, askers can follow guidelines to post their questions effectively. We list here the main observed guidelines.

1) Write self-contained questions without a URL (if possible). Respondents are hesitant in visiting such links due to which queries are not responded. Prior studies have shown the importance of including URLs in the raised questions to provide more details about the raised questions [11] [50][63]. We examine the impact of adding URLs in the posted messages and provided a detailed investigation of the results of RQ2. The investigation concludes that the URLs should only be included when necessary and should coincide with the proper details.

2) Visit the FAQs and recently asked questions. Our results indicate that asking similar questions in the chatrooms may result in fewer responses (e.g., the chatrooms in pattern 2). However, it is very difficult to go through all of the previous messages, but a quick overview can help reduce the queries for similar questions. A searchable summary of the existing discussions can help greatly in reducing the number of similar questions and would increase the possibility of getting more unseen questions.

3) Include code snippets and reproducibility details. Askers are recommended to add their code and error messages in the posted messages, which make the messages more understandable for other developers. The example (from an angular chatroom) below represents the asker adding the code and error along with the detail of what is the rationale of the scenario in the posted message. Adding such details in a message can greatly help in getting a faster response. Prior studies also show that including the code snippets with appropriate context can increasing the change of getting a response [63].

> *"Hi, anybody testing their app with karma-typescript? I can't get global angular-mock functions to work, here is my code*
>
> ```
> describe('AdminController', () => {
> beforeEach(() => module('app'));
>
> it('loads data', () => {
> expect(1).toEqual(1);
> });
> });
> ```
>
> *here is the error"*
>
> ```
> TypeError: ngMock_1.module is not a function
> at <Jasmine>
> at UserContext.<anonymous> (public/app/admin/admin.spec.ts:6:22 <- public/app/admin/admin.spec.js:6:46)
> at <Jasmine>
> ```

Project maintainers: As a project maintainer (i.e., the developer contributing to the linked GitHub project by committing code that fixes the reported issues or reviewing the committed code) of the popular project, it is impossible to go through and respond to all of the discussions happening in chatrooms.

1) The value of our thread identification approach. Our thread identification approach can help project maintainers to get a quick overview of discussion threads. Project maintainers can also get assistance in terms of getting to know the

topics that are discussed by developers. In turn, this can help maintainers in prioritizing the features to implement and issues to resolve.

2) Spot the modules of documentation for improvements. Project maintainers can also use identified threads to improve the documentation of the project by including the repeated questions about project features. We also observe that 30% of the resolved threads include a suggestion from respondents in the form of external links. The external links include blogs and a Q&A forum link where a similar query is asked. Project maintainers can use the suggested readings information to improve the documentation of the projects. The example below shows the suggested reading for the support of a tool for the project. The suggestions can be used to improve the support of the project and improve the documentation of the project.

> **Asker:** *"Hi. What to do with CakeException: MSBuild: Could not locate executable ? I'm just trying to execute msbuild: without*
> `ToolVersion = MSBuildToolVersion.VS2017`
> *it doesn't work, it seems that Cake picks up some old msbuild which fails on my projects".*
>
> **Respondent:** *"I would suggest using the VSWhere tool, as where 2017 is installed isn't predictable anymore".(link)*

3) Tailor the guidelines for posting questions to be suitable with the response style in your project. We observe that developers in every chatroom have a certain development style; project maintainers can tailor the guidelines of their project to fit in with the response style/pattern.

4) Use similar threads to form FAQs. The questions asked in Q&A forums and chatrooms more often can be included in the project as Frequently Asked Questions (FAQ). The results also indicate that askers include code snippets in their questions. Respondents resolve the raised questions by fixing the posted code. The code snippets can be used as an example while explaining the relevant component in the project documentation. We recommend that researchers can use the existing similar question detection approaches [8] to identify similar threads in a chatroom.

5) Include a chatbot in your chatroom. Gitter provides an open-source API[7] which enables the project maintainers to include a chatbot in their chatroom. Project maintainers can use the chatbot to recommend the most relevant threads to a newly posted question or suggest the best respondents who answered similar questions.

6) Identify the discussion duration and other related analysis. Project maintainers can use the features used in our analysis to extend our study for multiple objectives. For example, project maintainers can investigate different aspects of the identified threads, such as analyzing the correlation between the extracted features with the thread duration, the number of messages in a thread, and the number of participants in threads.

6 RELATED WORK

In this section, we present the related work on analyzing social interaction platforms, thread disentanglement, asking high-quality questions, and identifying the topics of the posted questions.

[7] https://github.com/Odonno/gitter-bot-how-to

6.1 Analyzing developer social interaction platforms

Prior studies analyze the interaction between developers using different social interaction platforms (e.g., chatrooms and mailing lists) [13] [16] [17] [33] [48] [61] [71]. For example, Shihab *et al.* [61] investigate the developer IRC meetings from two large open-source projects. The focus of the study is along three dimensions: meeting participants, content, and style. Results show that core members of the project do not actively participate in the meetings. Topics of the meetings are related to the project in general and specific to some release. The information extracted from the content can be used to identify the discussion among the members. Ibrahim *et al.* [33] study the development of mailing lists and identify the main features that encourage developers to contribute to mailing lists. Personalized models are developed to predict the discussions that developers would participate in. Ibrahim *et al.* find that the content of the discussion, the length of discussion, and developer contribution activities are the most significant contributing factors for developers to participate in discussions. The nature of mailing lists and IRC meeting channels are different from the chatrooms because chatrooms are informal and contain irrelevant messages to the discussion due to multiple participants and entangled discussions.

Di Sorbo *et al.* [17] use Natural Language Parsing to qualify the content of the email discussions between developers according to their purpose. The study identifies the key intents in the developer emails, such as feature requests, opinion asking, problem discovery, solution giving, and information giving. The identified content can be used to improve the documentation of the project. A follow-up study by Di Sorbo *et al.* [16] present an approach to classify the content of the emails according to their purpose and identify the email fragments that can be used for maintenance tasks. The approach is able to distinguish between the feature requests, problem discovery, and other related intents from the content of the development emails.

Recently, Chatterjee *et al.* [13] manually analyze 400 threads and find that Q&A chat messages provide similar information as can be found on Q&A forums (i.e., a question is raised and multiple answers are provided). Chatterjee *et al.*'s study also reflects the importance of analyzing the threads (instead of analyzing a random selection of messages) to identify topics and extract Q&A. However, Chatterjee *et al.*'s study does not propose an automatic thread identification approach rather manually examine 400 threads.

Studies on social interaction platforms focus more on qualitative features such as user activity, the raised topics, and keyword analysis without analyzing the discussion threads.

6.2 Thread disentanglement

Chatrooms have multiple participants who participate in different discussions at the same time. Disentangling threads in the multi-participant chatroom is a challenge and several studies try to identify messages that belong to the same thread [1] [20] [21] [40] [58] [67]. Shen *et al.* [58] use a single pass clustering approach to identify entangled threads from chat messages between students and teachers. The approach starts with the first message of the cluster and assigns the following message to an existing cluster if a certain threshold is exceeded. Messages are represented as a vector space model with Term Frequency-Inverse Document Frequency (TF-IDF). The approach uses the similarity of vectors for each message along with the sentence types and pronouns to calculate the probability of the messages belonging to the thread. The 16 text streams used in the evaluation have an average of 102.8 messages and have achieved 0.61 F-score with eight as the window size. Wang and Oard [67] propose a clustering algorithm that comprises of two steps for every incoming message. The first step performs the single-pass clustering (i.e., assigning the message to the cluster if there exists a strong relationship). The second step does another pass for all the clusters and renews the associations

Table 12. Summary of prior studies which identify threads from text streams.

Study	Venue-Year	Dataset description	Accuracy (F-score)
Shen *et al.* [58]	ACM SIGIR - 2006	16 text streams with 102.8 average messages in each stream.	0.61
Adams and Martell [1]	ICSC - 2008	200 conversations.	0.67
Wang *et al.* [67]	Human Language Technologies - 2009	800 lines of annotated text messages.	0.72
Elsner and Charniak [20]	Computational Linguistics - 2010	2,000 messages.	0.71
Elsner and Charniak [21]	Computational Linguistics - 2011	20,000 lines of messages.	0.79
Mayfield *et al.* [40]	Computational Linguistics - 2012	15 conversations.	0.71
Current Study	-	11,049,802 messages posted in Gitter.	**0.81**

among messages. Wang *et al.* evaluate their approach using 800 lines of annotated text messages. Due to the social and temporal contexts considered in Wang *et al.*'s approach, the evaluation shows that the approach achieved an F-score of 0.72.

Elsner and Charniak [20] use a correlation clustering approach to identify threads from chats. Correlation clustering finds a set of clusters with the maximum agreements between pairs in the clusters and a maximum disagreement between pairs of different clusters. The approach uses the maximum-entropy classifier to determine if two messages are related. A total of around 2,000 messages are annotated to evaluate the approach and have achieved 0.71 F-Score. Another approach by Elsner and Charniak [21] use coherence models to identify threads from chat messages. Elsner and Charniak's approach uses a tabu search to find messages belonging to the same thread. They conducted two different experiments, one for disentangling a single message (assuming the rest of the chat log has the correct structure) and the second experiment for disentangling the whole chat logs, which performed worse due to biases in their model. The results of the two conducted experiments generate incorrect threads. Running the approach on approximately 20,000 lines of messages, an accuracy of 79.3 is achieved. Mayfield *et al.* [40] propose a two-pass algorithm, the first-pass labels the sentences using a negotiation framework, and the second pass groups these sentences using a cluster classifier. The experiment is conducted on 15 conversations with an accuracy of 0.71. Adams and Martel [1] use connectivity matrices to build a parent-child relationship between the messages. The approach first creates a similarity matrix and then creates a directed graph based on the threshold. Then, the approach uses hypernym augmentation, time-distance penalization, and nickname augmentation to determine whether a message belongs to a thread. Results show that the time-distance penalization has the largest impact on improving the accuracy of the algorithm. The approach is evaluated on the teenagers' chatroom corpus of around 200 conversations and achieved an F-score of 0.67. Table 12 summarizes prior studies which identify threads from text streams.

The aforementioned approaches use multiple techniques that are costly and achieve low F-score. Moreover, the aforementioned approaches are evaluated on a small and already annotated corpus of conversations. Prior work assumes that every message has to be a part of a thread. However, in developer chatrooms (as shown in our study), 37.3% of the raised/posted questions do not trigger a response. No existing studies focus on the developer chatrooms as the discussion in the chatrooms is technical, and every chatroom has a different context. Our approach can identify threads by using simple heuristics such as user involvement, text similarity, and back-and-forth communication. We have applied our approach to a large corpus of 6,605,248 messages and achieved a good F-score of 0.81.

6.3 Asking high-quality questions

Prior studies examine the questions asked in Q&A platforms (e.g., Stack Overflow) to determine the key characteristics in getting an answer [9] [10] [11] [42] [50] [51] [63] [71]. Q&A platforms provide standard rules for accepting and voting the answers which help determine the best answer and indirectly help identify the characteristics of high-quality questions (i.e., questions with a high probability of getting answers). Treude *et al.* [63] analyze the Stack Overflow (SO) questions to see the types of questions that are asked in SO, the questions that are answered, and the questions that are unanswered. The study shows that "How-to" questions are the most frequently asked questions. The results also indicate that concrete questions with appropriate code snippets and URLs are likely to get answers. The type of raised questions is not the only factor in determining whether questions get an answer. In addition, the profiles of the askers, day and time at which questions are asked, and the length of the posed questions impact the likelihood of getting answers.

Calefato *et al.* [9] investigate four stack exchange websites to find factors for asking high-quality questions. Calefato *et al.* find that the presentation of the raised questions, in terms of concise details and relevant code snippets, can increase the likelihood of getting responses. The posting time of a question is also an important factor in getting responses. Another study by Calefato *et al.* [11] empirically investigate the impact of the presentation quality and posting time on the success of the posted questions in Stack Overflow. The study provides evidence-based guidance for askers to increase the probability of getting answers. The inclusion of code snippets, URLs, and concise content of the posted question is highly recommended. The study also shows that there exist low and high-efficiency hours in terms of responding to developers.

Ponzanelli *et al.* [51] provide an approach to improve the detection of low-quality questions to improve the content of Q&A websites. Ponzanelli *et al.*'s approach identifies low-quality questions by extracting different features of the posted questions, such as the body length, the URL count, the average term entropy, and the number of up-votes and down-votes of the posted question. The study can identify low-quality questions that can help improve the question reviewing process in Stack Overflow. Another study by Ponzanelli *et al.* [50] presents an approach to classify questions to bad and good questions based on their quality. The prominent features for indicating good/bad questions include body length, title length, email count, URLs count, tags count, and other readability features. Ponzanelli *et al.*'s work shows that including URL is a good practice, and too long questions tend to be rated as bad questions.

Zagalsky *et al.* [71] report a qualitative study to determine how the R community creates and organizes the knowledge in Stack Overflow questions and R mailing list. Based on results, authors suggest the askers ask the questions in the appropriate channels which keeping in mind the rules and standards described by the channel. The study also encourages developers to post well-crafted questions by providing good background, including the URLs, to provide the relevant details rather than attachments, and including the links to the already examined resources. Chhabra *et al.* [15] investigate the Ortega hypothesis, which states that a large number of novice users (i.e., users who are not well-qualified) are instrumental to the progress of any system. The study includes low and high contributing users in the Stack Overflow questions. The study shows that the questions posted by masses (i.e., less experienced users) provide a useful contribution to the system as such questions are responsible for extracting information from more experienced users.

Prior studies analyze Q&A platforms to extract the key features for asking high-quality questions. Our work aims to provide guidelines for developers asking questions in the chatrooms. Our study confirms and extends prior work as we find similar guidelines (e.g., inclusion of URLs and code snippets), and we provide customized guidelines for every chatroom based on the characteristics of every chatroom.

6.4 Topic identification of the posted questions

Recent studies analyze the topics discussed in the Q&A forums [3] [4] [5] [56] [70]. Bandeira *et al.* [3] analyze the microservices community questions and answers from the Stack Overflow dataset. The authors use Latent Dirichlet Allocation (LDA) to generate the topics that are discussed in 1,043 Stack Overflow posts. The results show that the most popular discussion in microservices posts is related to Netflix Eureka (13%). However, the most popular concept in the micro-services domain (i.e., blue/green deployment) is not identified as the discussed topics. Finally, the analysis shows that the high discussion rate in Stack Overflow does not reflect the popularity of a certain topic in the community. Bangash *et al.* [4] perform LDA on more than 28 thousand Stack Overflow posts and identify 44 topics specific to machine learning developers. The most common topics include algorithms, classification, and training datasets. Further analysis shows that developers lack introductory knowledge about deep learning and often have less feedback from the community. The study also recommends appropriate tags for the posts which have irrelevant tags.

Yang *et al.* [70] conduct a large scale study on the Stack Overflow questions related to security. The authors use LDA tuned using a genetic algorithm to identify the topics and investigate the popularity and difficulty of the topics. The popular five categories include web security, mobile security, cryptography, software security, and systems security. Most questions related to web security discussions include managing user passwords, hashing algorithms, generating signatures, and the SQL injection attacks as the most popular topics. Barua *et al.* [5] present a methodology to analyze the textual content of Stack Overflow posts using LDA. The study defines various metrics to quantify the topics and investigate their changes over time. The result shows that mobile development is more prominent compared to web development. In mobile development, Barua *et al.* find that the questions related to both Android and iPhone platforms are on the rise. In web development, PHP is gaining popularity, and the .NET framework is slowly declining. Rosen *et al.* [56] examine around 13 million posts from Stack Overflow. The authors use LDA to determine the mobile-related topics from the Stack Overflow posts. The most popular topics include app distribution, mobile APIs, data management, and sensors. The study also concludes that mobile-related questions are difficult to answer than non-mobile questions. Additionally, the study compares the topics among Android, iPhone, and Windows platforms and identifies the common topics as well as the specific topics for each platform.

Prior studies apply automatic approaches (e.g., using LDA) to identify the discussion topics of the raised questions in Q&A platforms. In our study, we need a deeper analysis of the chatroom discussion than extracting the discussed topics, as we study whether the raised questions get resolved and what are the different resolution types. In addition, the question body in the Q&A platforms is comprehensive. However, as compared to chatrooms, the posted messages are short, informal, and the discussion is expanded over multiple messages. Hence, in RQ3, we perform a qualitative study on a statistically representative random sample of 384 threads to gain an in-depth understanding of the discussed topics and how the posed questions are resolved.

7 THREATS TO VALIDITY

This section addresses the threats to the validity of our approach.

Threats to conclusion validity concern the relation between the treatment and the outcome. The threats to conclusion validity include the errors introduced by processing the developer's messages. In particular, chatroom messages are written by developers in an informal text format. Hence, they may contain incorrect words. To mitigate this issue, we use SymSpell [14] to automatically fix any spelling errors in the processed messages. However, our approach achieves 0.81 accuracy, which impacts the obtained results in RQ2 and RQ3 (as the proposed approach may

automatically classify messages to their corresponding threads). The F1-score indicates that our approach wrongly identifies around 20% of the messages as part of the threads.

To mitigate the error of manual identification of threads, the first and the third author of this paper independently identify threads form the chatroom messages. The measured Fliess' Kappa is 0.81, which shows a strong agreement between the evaluators. The high value of the obtained inter-rater agreement indicates that both authors identified the same threads from the investigated messages. However, we are not the project maintainers of the studied GitHub projects, and our results could be biased with our knowledge. Thus, we recommend that future studies can complement our analysis of the identified threads through developer surveys.

Threats to internal validity concern our selection of subject projects and analysis methods. The thread identification depends on the chatrooms selected. To minimize the threat of our results being biased towards specific chatrooms, we select a large number of chatrooms that are distributed across different categories. In addition, we filter the chatrooms with fewer messages to make sure we select only the chatrooms with the probability of having threads.

The features identified for mixed-effect models are effective for only the period of the data selected (i.e., creation of chatrooms till March 2019). A time span expanding over different boundaries might result in different features significance. The threads can be of different lengths in terms of the number of messages. We used a sliding window of four messages and kept moving the sliding window if new messages are found. In cases where the messages from the same thread are apart by more than the messages in the window, our approach is not able to capture such messages.

For identifying the message singularity, we use a window of last 30 days to check for similar questions. A different window can result in the different calculations for the message singularity. Similarly, for calculating the participation, we use the last 30 days window as the activity of the developer might differ from month to month. A quarterly or yearly window can yield a different representation of the participation of the developer. The evaluation for thread identification in RQ1 done by the two evaluators (i.e., the first and third author) might miss some of the false negatives (i.e., messages that are part of the thread but not identified by the approach). To reduce the threat, we provided the evaluators with 15 messages before the first identified message and 15 messages after the identified message. Any message (part of the identified thread) beyond this limit is not captured by the evaluators.

As raised by prior studies [35] [69], it is essential to identify and resolve the different aliases that developers use in open-source project discussions. Similarly, we fix the disambiguation of the identity problem in discussion threads as follows. Instead of using the heuristics proposed by [35] [69], which is necessary for mailing lists and other similar datasets, we used GitHub API to retrieve the GitHub accounts of the askers/respondents. In Gitter, every message is linked to a single username to reveal the ownership of the message. We verified that every Gitter user has a different GitHub account. Hence, we are able to confirm the identities of the developers who post messages in Gitter. Our approach is similar to the method proposed by Avelino *et al.* [2] to solve the disambiguation in GitHub users. Similar to other existing approaches, our approach is unable to identify the cases when the developer has multiple GitHub accounts.

Prior studies use Latent Dirichlet Allocation (LDA) to identify the higher-level concepts in a corpse of documents [4] [31] [47] [70]. To identify discussion threads, we initially used the LDA to group the messages based on the discussion topic. We used the optimal number of topics when applying the LDA to our dataset. We found that the identified threads are not representative of the actual discussions. In other words, the accuracy of our thread identification approach was decreased by considering the LDA as a feature to group similar messages of a discussion thread. This might be due to the small size of the text presented in chatroom messages. In addition, the goal of each chatroom is different, which

makes it challenging to use one theme of topics to identify the discussion threads in all chatrooms. Hence, we did not use LDA in the process of identifying similar threads.

Prior work proposed metrics such as conciseness and completeness to evaluate the quality of the generated release notes [43]. For the evaluation of the identified threads, we provided two authors of the paper with a statistically representative random sample of 384 threads. The main objective of the manual validation is to examine whether all the relevant messages are added to a thread (i.e., avoiding false-negative cases), and no irrelevant messages are wrongly labeled (i.e., preventing false-positive cases). Prior approach evaluates the completeness of the generated release notes by measuring the percentage of release notes that are correctly generated. In our context, this definition has an issue as missing a single message in a thread makes it incomplete. Hence, we used the F1-score to evaluate the proportion of the missed parts of a thread instead of measuring the percentage of the correctly identified threads. Regarding the conciseness, the aim of the study is not to identify the quality of the responses. In particular, we do not remove any less meaningful messages from the threads. Hence, we do not use conciseness in our study.

Threats to external validity address the possibility of generalizing our results. In this study, we analyze Gitter because Gitter is one of the most popular chatroom platforms for open source projects in addition to the fact that the messages are available. Slack is a popular chatroom platform among developers (especially within organizations and teams) to discuss ideas and communicate with each other. The Slack chatrooms data can be used to extend our findings to other platforms.

8 CONCLUSION

Open-source software developers use chatrooms to communicate with the community. Developer chatrooms are used to ask queries, design ideas, and solve problems raised by peers. In this paper, we perform an in-depth analysis of 6,605,248 messages of 709 chatrooms from Gitter. Our main findings can be summarized as follows:

(1) We propose an algorithm to identify threads from chat messages with 0.81 F1-Score.
(2) We study the significant features which impact the response behavior of the respondents. We observe that questions posted by inactive askers are more likely to be responded by the members of the chatroom. We also find that providing details in a message rather than external links are more likely to get a response.
(3) We identify different patterns of respondents responding to questions and raised topics in the chatrooms. The identified patterns of respondents can help project maintainers set up guidelines for their chatrooms, which makes askers write their questions in a suitable way for each chatroom.
(4) We label a statistically significant random sample of threads to identify topics, resolution types, and resolution outcomes of the threads. We find that the topic of the posted questions and the edited questions have a significant effect on resolving the thread. We also observe that around 80% of the studied threads posted in the chatrooms are resolved. Hence, we recommend that project maintainers can leverage the provided question-answer pairs in the threads discussions to enrich the documentation for their project (e.g., improving the FAQ of their project).

To ease the replication of our study, we shared our dataset and the scripts for processing chatroom messages in our replication package[8].

REFERENCES

[1] Paige H Adams and Craig H Martell. 2008. Topic identification and extraction in chat. In 2008 IEEE international conference on Semantic computing. IEEE, 581–588.

[8] https://gitlab.com/e.osamaehsan/gitteranalysis/ username: **t_user** password: **Test123+**

[2] Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. 2019. On the abandonment and survival of open source projects: An empirical investigation. In 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE, 1–12.

[3] Alan Bandeira, Carlos Alberto Medeiros, Matheus Paixao, and Paulo Henrique Maia. 2019. We need to talk about microservices: an analysis from the discussions on StackOverflow. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, 255–259.

[4] Abdul Ali Bangash, Hareem Sahar, Shaiful Chowdhury, Alexander William Wong, Abram Hindle, and Karim Ali. 2019. What do developers know about machine learning: a study of ML discussions on StackOverflow. In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, 260–264.

[5] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. 2014. What are developers talking about? an analysis of topics and trends in stack overflow. Empirical Software Engineering 19, 3 (2014), 619–654.

[6] Douglas Bates, Martin Maechler, Ben Bolker, Steven Walker, Rune Haubo Bojesen Christensen, Henrik Singmann, Bin Dai, Fabian Scheipl, and Gabor Grothendieck. [n.d.]. Package 'lme4'. ([n. d.]).

[7] Jason Bengel, Susan Gauch, Eera Mittur, and Rajan Vijayaraghavan. 2004. Chattrack: chatroom topic identification using classification. In International Conference on Intelligence and Security Informatics. Springer, 266–277.

[8] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2009. Frequently asked questions in bug reports. Technical Report. University of Calgary.

[9] Fabio Calefato, Filippo Lanubile, MR Merolla, and N Novielli. 2015. Success factors for effective knowledge sharing in community-based question-answering. In Proc. of the International Forum on Knowledge Asset Dynamics (IFKAD 2015). 1431–1441.

[10] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. 2016. Moving to stack overflow: Best-answer prediction in legacy developer forums. In Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement. 1–10.

[11] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. 2018. How to ask for technical help? Evidence-based guidelines for writing questions on Stack Overflow. Information and Software Technology 94 (2018), 186–207.

[12] Fabio Calefato, Filippo Lanubile, and Nicole Novielli. 2019. An empirical assessment of best-answer prediction models in technical Q&A sites. Empirical Software Engineering 24, 2 (2019), 854–901.

[13] Preetha Chatterjee, Kostadin Damevski, Lori Pollock, Vinay Augustine, and Nicholas A Kraft. 2019. Exploratory study of Slack Q&A chats as a mining source for software engineering tools. In Proceedings of the 16th International Conference on Mining Software Repositories. IEEE Press, 490–501.

[14] Sym Spell Checker. [n.d.]. Sym Spell Checker. https://github.com/wolfgarbe/SymSpell. (Last accessed: August 2019).

[15] Anamika Chhabra and SRS Iyengar. 2019. Investigating Ortega Hypothesis in Q&A portals: An Analysis of StackOverflow. arXiv preprint arXiv:1911.02376 (2019).

[16] Andrea Di Sorbo, Sebastiano Panichella, Corrado A Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald Gall. 2016. DECA: development emails content analyzer. In 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C). IEEE, 641–644.

[17] Andrea Di Sorbo, Sebastiano Panichella, Corrado A Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C Gall. 2015. Development emails content analyzer: Intention mining in developer discussions (T). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 12–23.

[18] T3N Digital. [n.d.]. Gitter: A chat for GitHub users. https://t3n.de/news/gitter-github-chat-509383/. (Last accessed: August 2019).

[19] Mariam El Mezouar, Feng Zhang, and Ying Zou. 2018. Are tweets useful in the bug fixing process? An empirical study on Firefox and Chrome. Empirical Software Engineering 23, 3 (2018), 1704–1742.

[20] Micha Elsner and Eugene Charniak. 2010. Disentangling chat. Computational Linguistics 36, 3 (2010), 389–409.

[21] Micha Elsner and Eugene Charniak. 2011. Disentangling chat with local coherence models. In Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies. 1179–1189.

[22] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. Psychological bulletin 76, 5 (1971), 378.

[23] Documentation for R stats package. [n.d.]. STAT package. https://cran.r-project.org/web/packages/STAT/index.html. (Last accessed: August 2019).

[24] John Fox, Sanford Weisberg, Daniel Adler, Douglas Bates, Gabriel Baud-Bovy, Steve Ellison, David Firth, Michael Friendly, Gregor Gorjanc, Spencer Graves, et al. 2012. Package 'car'. Vienna: R Foundation for Statistical Computing (2012).

[25] Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. [n.d.]. An empirical study of the long duration of continuous integration builds. Empirical Software Engineering ([n. d.]), 1–38.

[26] Cyril Goutte and Eric Gaussier. 2005. A probabilistic interpretation of precision, recall and F-score, with implication for evaluation. In European Conference on Information Retrieval. Springer, 345–359.

[27] Latifa Guerrouj, Shams Azad, and Peter C Rigby. 2015. The influence of app churn on app success and stackoverflow discussions. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE, 321–330.

[28] Mark Handel and James D Herbsleb. 2002. What is chat doing in the workplace?. In Proceedings of the 2002 ACM conference on Computer supported cooperative work. ACM, 1–10.

[29] James A Hanley and Barbara J McNeil. 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology 143, 1 (1982), 29–36.

[30] Safwat Hassan, Chakkrit Tantithamthavorn, Cor-Paul Bezemer, and Ahmed E Hassan. 2018. Studying the dialogue between users and developers of free apps in the google play store. Empirical Software Engineering 23, 3 (2018), 1275–1312.

[31] Abram Hindle, Christian Bird, Thomas Zimmermann, and Nachiappan Nagappan. 2012. Relating requirements to implementation via topic analysis: Do topics extracted from requirements make sense to managers and developers?. In 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, 243–252.

[32] Pieter Hooimeijer and Westley Weimer. 2007. Modeling bug report quality. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 34–43.

[33] Walid M Ibrahim, Nicolas Bettenburg, Emad Shihab, Bram Adams, and Ahmed E Hassan. 2010. Should I contribute to this discussion?. In 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, 181–190.

[34] Frank E Harrell Jr. [n.d.]. Harrell Miscellaneous. https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf. (Last accessed: August 2019).

[35] Erik Kouters, Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ Van Den Brand. 2012. Who's who in Gnome: Using LSA to merge software repository identities. In 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, 592–595.

[36] Francine Lafontaine and Kenneth J White. 1986. Obtaining any Wald statistic you want. Economics Letters 21, 1 (1986), 35–40.

[37] Bin Lin, Alexey Zagalsky, Margaret-Anne Storey, and Alexander Serebrenik. 2016. Why developers are slacking off: Understanding how software teams use slack. In Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion. ACM, 333–336.

[38] Edward Loper and Steven Bird. 2002. NLTK: the natural language toolkit. arXiv preprint cs/0205028 (2002).

[39] Willy JR Martin, Bernard PF Al, and Piet JG Van Sterkenburg. 1983. On the processing of a text corpus: From textual data to lexicographical information. Lexicography: Principles and practice (1983), 77–87.

[40] Elijah Mayfield, David Adamson, and Carolyn Penstein Rosé. 2012. Hierarchical conversation structure prediction in multi-party chat. In Proceedings of the 13th Annual Meeting of the Special Interest Group on Discourse and Dialogue. Association for Computational Linguistics, 60–69.

[41] Douglas R McCallum and James L Peterson. 1982. Computer-based readability indexes. In Proceedings of the ACM'82 Conference. ACM, 44–48.

[42] Qing Mi, Yujin Gao, Jacky Keung, Yan Xiao, and Solomon Mensah. 2017. Identifying Textual Features of High-Quality Questions: An EmpiricalStudy on Stack Overflow. In 24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017. IEEE Computer Society, 636–641.

[43] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2014. Automatic generation of release notes. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 484–495.

[44] Ehsan Noei, Feng Zhang, and Ying Zou. 2019. Too Many User-Reviews, What Should App Developers Look at First? IEEE Transactions on Software Engineering (2019).

[45] Christiane Nord. 2005. Text analysis in translation: Theory, methodology, and didactic application of a model for translation-oriented text analysis. Number 94. Rodopi.

[46] Aditya Pal, Shuo Chang, and Joseph A Konstan. 2012. Evolution of experts in question answering communities. In sixth international AAAI conference on weblogs and social media.

[47] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshynanyk, and Andrea De Lucia. 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In 2013 35th International Conference on Software Engineering (ICSE). IEEE, 522–531.

[48] Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. 2014. How developers' collaborations identified from different sources tell us about code changes. In 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 251–260.

[49] Joël Plisson, Nada Lavrac, Dunja Mladenic, et al. 2004. A rule based approach to word lemmatization. Proceedings of IS-2004 (2004), 83–86.

[50] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, and Michele Lanza. 2014. Understanding and classifying the quality of technical forum questions. In 2014 14th International Conference on Quality Software. IEEE, 343–352.

[51] Luca Ponzanelli, Andrea Mocci, Alberto Bacchelli, Michele Lanza, and David Fullerton. 2014. Improving low quality stack overflow post detection. In 2014 IEEE international conference on software maintenance and evolution. IEEE, 541–544.

[52] Gitlab Issue report Gitter. [n.d.]. Measure monthly active users. https://gitlab.com/gitlab-org/gitter/webapp/issues/2000. (Last accessed: August 2019).

[53] Juliana Reyes. [n.d.]. These devs delved into the world of Backbone.js and were blown away by what they found. https://technical.ly/philly/2015/02/06/backbone-js-marionette-open-source-squareknot/. (Last accessed: August 2019).

[54] Peter C Rigby and Ahmed E Hassan. 2007. What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list. In Proceedings of the fourth international workshop on mining software repositories. IEEE Computer Society, 23.

[55] Stephen Robertson. 2004. Understanding inverse document frequency: on theoretical arguments for IDF. Journal of documentation 60, 5 (2004), 503–520.

[56] Christoffer Rosen and Emad Shihab. 2016. What are mobile developers asking about? a large scale study using stack overflow. Empirical Software Engineering 21, 3 (2016), 1192–1223.

[57] Stack Share. 2019. Gitter vs RocketChat vs Slack. What are the differences. https://stackshare.io/stackups/gitter-vs-rocketchat-vs-slack#stats. (Last accessed: August 2019).

[58] Dou Shen, Qiang Yang, Jian-Tao Sun, and Zheng Chen. 2006. Thread identification in dynamic text message streams. In Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 35–42.

[59] Martin Shepperd, David Bowes, and Tracy Hall. 2014. Researcher bias: The use of machine learning in software defect prediction. IEEE Transactions on Software Engineering 40, 6 (2014), 603–616.

[60] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken-ichi Matsumoto. 2010. Predicting re-opened bugs: A case study on the eclipse project. In 2010 17th Working Conference on Reverse Engineering. IEEE, 249–258.

[61] Emad Shihab, Zhen Ming Jiang, and Ahmed E Hassan. 2009. Studying the use of developer IRC meetings in open source projects. In 2009 IEEE International Conference on Software Maintenance. IEEE, 147–156.

[62] Tom AB Snijders, Roel J Bosker, et al. 1999. An introduction to basic and advanced multilevel modeling. Sage, London. WONG, GY, y MASON, WM (1985): The Hierarchical Logistic Regression. Model for Multilevel Analysis, Journal of the American Statistical Association 80, 5 (1999), 13–524.

[63] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. 2011. How do programmers ask and answer questions on the web?(NIER track). In Proceedings of the 33rd international conference on software engineering. 804–807.

[64] David C Uthus and David W Aha. 2013. Multiparticipant chat analysis: A survey. Artificial Intelligence 199 (2013), 106–121.

[65] Fernanda B Viégas and Judith S Donath. 1999. Chat circles. In Proceedings of the SIGCHI conference on Human Factors in Computing Systems. ACM, 9–16.

[66] David Vronay, Marc Smith, and Steven Drucker. 1999. Alternative interfaces for chat. In Proceedings of the 12th annual ACM symposium on User interface software and technology. ACM, 19–26.

[67] Lidan Wang and Douglas W Oard. 2009. Context-based message expansion for disentanglement of interleaved text conversations. In Proceedings of human language technologies: The 2009 annual conference of the North American chapter of the association for computational linguistics. Association for Computational Linguistics, 200–208.

[68] Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. 2018. Understanding the factors for fast answers in technical Q&A websites. Empirical Software Engineering 23, 3 (2018), 1552–1593.

[69] Igor Scaliante Wiese, José Teodoro da Silva, Igor Steinmacher, Christoph Treude, and Marco Aurélio Gerosa. 2016. Who is who in the mailing list? Comparing six disambiguation heuristics to identify multiple addresses of a participant. In 2016 IEEE international conference on software maintenance and evolution (ICSME). IEEE, 345–355.

[70] Xin-Li Yang, David Lo, Xin Xia, Zhi-Yuan Wan, and Jian-Ling Sun. 2016. What security questions do developers ask? a large-scale study of stack overflow posts. Journal of Computer Science and Technology 31, 5 (2016), 910–924.

[71] Alexey Zagalsky, Daniel M German, Margaret-Anne Storey, Carlos Gómez Teshima, and Germán Poo-Caamaño. 2018. How the R community creates and curates knowledge: an extended study of stack overflow and mailing lists. Empirical Software Engineering 23, 2 (2018), 953–986.

[72] Guoliang Zhao, Daniel Alencar da Costa, and Ying Zou. 2019. Improving the pull requests review process using learning-to-rank algorithms. Empirical Software Engineering (2019), 1–31.