Learning to Describe Solutions for Bug Reports Based on Developer Discussions

Sheena Panthaplackel[1], **Junyi Jessy Li**[2], **Milos Gligoric**[3], **Raymond J. Mooney**[1]
[1]Department of Computer Science
[2]Department of Linguistics
[3]Department of Electrical and Computer Engineering
The University of Texas at Austin
spantha@cs.utexas.edu, jessy@austin.utexas.edu
gligoric@utexas.edu, mooney@cs.utexas.edu

Abstract

When a software bug is reported, developers engage in a discussion to collaboratively resolve it. While the solution is likely formulated within the discussion, it is often buried in a large amount of text, making it difficult to comprehend and delaying its implementation. To expedite bug resolution, we propose generating a concise natural language description of the solution by synthesizing relevant content within the discussion, which encompasses both natural language and source code. We build a corpus for this task using a novel technique for obtaining noisy supervision from repository changes linked to bug reports, with which we establish benchmarks. We also design two systems for generating a description *during* an ongoing discussion by classifying when sufficient context for performing the task emerges in real-time. With automated and human evaluation, we find this task to form an ideal testbed for complex reasoning in long, bimodal dialogue context.

1 Introduction

Software bugs in open-source projects are reported through issue tracking systems like GitHub Issues. When a bug is reported, a discussion is initiated among developers to collectively resolve it (Noyori et al., 2019). The *bug resolution* process is often strenuous and time-consuming, involving extended deliberations (Liu et al., 2020b) among multiple participants (Kavaler et al., 2017), spanning long periods of time (Kikas et al., 2015). Although a solution often emerges within the discussion (Arya et al., 2019), this can easily get lost in a large amount of text (Liu et al., 2020b). Wading through a long discussion to determine whether a solution has been suggested, comprehending it, and then implementing it can be daunting, especially for developers who are not closely following the discussion (Arya et al., 2019; Tan et al., 2020). Consequently, the resolution can be delayed.

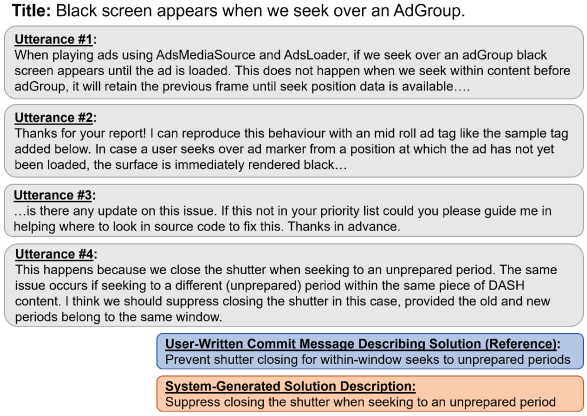


Figure 1: ExoPlayer bug report discussion with user-written and system-generated solution descriptions.

As developers scan through the long discussion, it is desirable to have an automated system that guides them to more easily absorb information relevant towards implementing the solution. We propose automatically generating a concise natural language description of the solution by synthesizing relevant content as it emerges in the discussion. For example, as the discussion in Figure 1 progresses, the cause of the bug is identified as the shutter getting closed "when seeking to an unprepared period" and a solution emerges: "suppress closing the shutter in this case, provided the old and new periods belong to the same window." Our task aims to describe this solution: *Prevent shutter closing for within-window seeks to unprepared periods*.

To study this task, we build a corpus from bug report discussions on GitHub Issues. The changes made within the code base to resolve the bug are often linked to the bug report in the form of a commit or pull request. We develop a novel approach to ob-

Findings of the Association for Computational Linguistics: ACL 2022, pages 2935 - 2952

tain noisy supervision for the solution description from the associated commit message or pull request title which describe the bug-resolving changes in natural language. To control for noise, we apply filtering techniques. The dataset and code are publicly available for research use.[1]

With this data, we set benchmarks for generating solution descriptions, conditioned on the discussion. From the long context, a model must learn to tease out and condense information relevant to the solution. Handling long context is critical for tasks with dialogue as input, since the input grows rapidly with the number of interactions. Additionally, the context entails technical text, with natural language and source code often appearing in the same sentence (Li et al., 2018). So, deducing information from the context to articulate a meaningful description requires complex reasoning. We explore generation models including transformer models (Vaswani et al., 2017) and PLBART (Ahmad et al., 2021), which was pretrained on large quantities of code and technical text. We evaluate with automated metrics and human evaluation.

Furthermore, we investigate integrating our task into a real-time setting. An informative description can be generated only if there is sufficient context about the solution, so we must wait until this context is available. In Figure 1, generation should be performed only after utterance #4 is made in the discussion. Since the solution is not formulated until that point, there is insufficient context to reliably generate a description before then. We design two methods for integrating a classifier that determines *when* to generate with a generation model: (1) a *pipelined* system with independently trained classification and generation models; (2) a *joint* system that is simultaneously trained for both tasks.

By monitoring progress and later chiming into the discussion with a solution description, this combined system lays the groundwork for future work on developing an intelligent dialogue agent which participates in discussions to facilitate more efficient bug resolution. While there is growing interest in building tools to support development activities such as code summarization (Iyer et al., 2016; Ahmad et al., 2020), comment updating (Panthaplackel et al., 2020b), and commit message generation (Loyola et al., 2017), dialogue systems have been largely understudied in this domain. We consider our work as a step towards building more dialogue-based AI tools for software development.

[1] https://github.com/panthap2/describing-bug-report-solutions

2 Problem Setting

As shown in Figure 1, when a user reports a bug, they state the problem in the *title* (e.g., “Black screen appears when we seek over an AdGroup") and initiate a discussion by making the first *utterance* (U_1), which usually elaborates on the problem. Other participants join the discussion at later time steps through utterances ($U_2 ... U_T$), where T is the total number of utterances. Throughout the discussion, developers discuss various aspects of the bug, including a potential solution (Arya et al., 2019). We propose the task of generating a concise description of the solution (e.g., “Prevent shutter closing for within-window seeks to unprepared periods") by synthesizing relevant content within the title and sequence of utterances ($U_1, U_2 ...$).

3 Data

Following prior work on other tasks (Kavaler et al., 2017; Panichella et al., 2021), we mine issue reports corresponding to open-source Java projects from GitHub Issues. Issue reports can entail feature requests as well as bug reports. In this work, we focus on the latter. We identify bug reports by searching for “bug" in the labels assigned to a report and by using a heuristic for identifying bug-related commits (Karampatsis and Sutton, 2020).

3.1 Data Collection

A bug report is organized as an event timeline, recording activity from when it is opened to when it is closed. From comments that are posted on this timeline, we extract utterances which form the *discussion* corresponding to a bug report, ordered based on their timestamps. We specifically consider bug reports that resulted in code (or documentation) fixes (Nguyen et al., 2012). These changes are made through *commits* and *pull requests*, which also appear on the timeline. Changes made in a commit or pull request are described using natural language, in the corresponding commit message (Loyola et al., 2017; Xu et al., 2019) or pull request title (Kononenko et al., 2018; Zhao et al., 2019). In practice, commit messages and pull request titles are written after code changes. However, like contemporary work (Chakraborty and Ray, 2021), we treat them as a proxy for solution descriptions to drive bug-resolving code changes.

	Train	Valid	Test	Total
Projects	395 (330)	145 (111)	134 (104)	412 (344)
Examples	9,862 (4,664)	1,232 (599)	1,234 (593)	12,328 (5,856)
# Commit messages	4,520 (2,355)	410 (234)	386 (189)	5,316 (2,778)
# PR titles	5,342 (2,309)	822 (365)	848 (404)	7,012 (3,078)
Avg T	3.9 (4.5)	3.8 (4.4)	4.0 (4.4)	3.9 (4.5)
Avg t_g	2.9 (3.4)	2.9 (3.4)	3.2 (3.6)	2.9 (3.4)
Avg utterance length (#tokens)	68.4 (75.6)	74.8 (84.3)	70.2 (75.7)	69.2 (76.5)
Avg title length (#tokens)	10.6 (10.6)	11.2 (11.0)	11.5 (11.3)	10.7 (10.7)
Avg description length (#tokens)	9.1 (10.5)	8.9 (9.9)	9.1 (10.1)	9.1 (10.4)

Table 1: Data statistics. In parentheses, we show metrics computed on the filtered subset.

Furthermore, we extract the position of a commit or pull request on the timeline, relative to the utterances in the discussion. We consider this as the point at which a developer acquired enough information about the solution to implement the necessary changes and describe these changes with the corresponding commit message or pull request title. So, if the implementation is done immediately after U_g on the timeline, then we take this position t_g as the "gold" time step for when sufficient context becomes available to generate an informative description of the solution. This leads to examples of the form *(Title, U_1...U_T, t_g, description)*.

We disregard issues with multiple commit messages/PR titles, so there is at most one example per issue. This is because the reason for needing multiple sets of changes is not clear (e.g., the solution could be implemented in parts or the first solution may have been incorrect and it is later corrected).[2]

3.2 Handling Noise

Due to significant noise in large online code bases like GitHub and StackOverflow, automatically extracted data from these sources is typically filtered both for more effective supervision and for more accurate evaluation (Panthaplackel et al., 2020a; Allamanis et al., 2016; Hu et al., 2018; Fernandes et al., 2019; Iyer et al., 2016; Yao et al., 2018; Yin et al., 2018). Upon studying the data, we also deemed filtering to be necessary. First, we apply simple heuristics to reduce noise, which we discuss in more detail in Appendix A. From this, we obtain the examples that are primarily used for training and evaluation in this work, which we refer to as the *full dataset*. Next, we identify three sources of noise that are more difficult to control with simple heuristics and use techniques described below to quantify them and build a *filtered subset* of the full dataset that is less noisy. This subset is used for more detailed analysis of the models that are discussed in the paper, and we find that training on this subset leads to improved performance (§4.3).

Generic descriptions: Commit messages and pull request titles are sometimes generic (e.g., *"fix issue."*) (Etemadi and Monperrus, 2020). To limit such cases, we compute normalized inverse word frequency (NIWF), which is used in prior work to quantify specificity (Zhang et al., 2018). The filter excludes 1,658 examples in which the reference description's NIWF score is below 0.116 (10th percentile computed from the training data).

Uninformative descriptions: Instead of describing the solution, the commit message or pull request title sometimes essentially re-states the problem (which is usually mentioned in the title of the bug report). To control for this, we compute the percentage of unique, non-stopword tokens in the reference description which also appear in the title. The filtered subset excludes 3,552 additional examples in which this percentage is 50% or more.

Discussions without sufficient context: While enough context is available to a developer to implement a solution at t_g, this context may not always be available in the discussion and could instead be from their technical expertise or external resources. For instance, in the discussion in the footnote[3], only a stack trace and personal exchanges between developers are present. From the utterance before the PR, "Or PM me the query that failed" suggests that an offline conversation occurred. Since relevant content is not available in such cases, it is unreasonable to expect to generate an informative description. We try to identify such examples with an approach (Nallapati et al., 2017) for greedily constructing an extractive summary based on a reference abstractive summary. The filtered subset excludes 1,262 more examples for which a summary could not be constructed (i.e., there is no relevant sentence that is extracted from the context). After applying all three filters, we have 5,856 examples.

[2]However, since such examples could be useful for future work, they are available in the data we release.

[3]https://github.com/prestodb/presto/issues/14567

		1	2	3	4
Full	Title	73.0	88.9	94.0	96.1
	$U_1 ... U_{t_g}$	54.7	87.6	95.0	97.6
	Title + $U_1 ... U_{t_g}$	47.9	82.0	91.2	94.8
Filtr.	Title	82.3	95.6	98.4	99.4
	$U_1 ... U_{t_g}$	49.9	87.4	95.1	97.8
	Title + $U_1 ... U_{t_g}$	47.5	86.0	94.5	97.5

Table 2: Percent of novel unigrams, bigrams, trigrams, and 4-grams in the reference description, with respect to the title, $U_1 ... U_{t_g}$, and title + $U_1 ... U_{t_g}$. High percentages show that generating solutions is an abstractive task.

3.3 Preprocessing

Since text in this domain can contain code tokens, we *subtokenize* (e.g., snake_case → snake case, camelCase → camel case) in the title, utterances, and description. We retain inlined code (on average 5.7 tokens/utterance); however, we remove code blocks and embedded code snippets (with markdown tags), as done in prior work (Tabassum et al., 2020; Ahmad et al., 2021). Capturing meaning from large bodies of code often requires reasoning with respect to the abstract syntax tree (Alon et al., 2019) and data and control flow graphs (Allamanis et al., 2018). However, markdown tags are not always used to identify code (Tabassum et al., 2020), and consequently, we observe some instances of larger code blocks within utterances that cannot be easily removed. We do not use source code files within a project's repository and leave it to future work to incorporate large bodies of code. We discard URLs and mentions of GitHub usernames from utterances. From the description, we remove references to issue and pull request numbers.

3.4 Partitioning

The dataset spans bug reports from April 2011 - July 2020. We partition based on the timestamp of the commit or pull request associated with a given example. Namely, we require all timestamps in the training set to precede those in the validation set and those in the validation set to precede those in the test set. Partitioning with respect to time ensures that we are not using models trained on future data to make predictions in the present, more closely resembling the real-world scenario (Nie et al., 2022). Dataset statistics are shown in Table 1.

4 Generating Solution Descriptions

We first generate informative solution descriptions in a static setting, in which we leverage the oracle context from the discussion (i.e., the title and $U_1 ... U_{t_g}$). From Table 1, the average length of a single utterance is ∼70 tokens while the average description length is only ∼9 tokens. Therefore, this task requires not only effectively selecting content about the solution from the long context (which could span multiple utterances) but also synthesizing this content to produce a concise description. Following See et al. (2017), we compute the percent of novel n-grams in the reference description with respect to the input context in Table 2. The high percentages underline the need for an *abstractive* approach, rather than an *extractive* one which generates a description by merely copying over utterances or sentences within the discussion.[4] Furthermore, this task requires complex, bimodal reasoning over the discussion, encompassing both natural language and source code.

4.1 Models

We benchmark various models for this task. To represent the input in neural models, we insert <TITLE_START> before the title and <UTTERANCE_START> before each utterance.

Copy Title: Though the bug report title usually only states a problem, we observe that it sometimes also puts forth a possible solution, so we evaluate how well it can serve as a solution description.

S2S + Ptr: We consider a transformer encoder-decoder model (Vaswani et al., 2017) in which we flatten the context into a single input sequence. Generating the output often requires incorporating project-specific out-of-vocabulary tokens from the input, so we support copying with a pointer generator network (Vinyals et al., 2015).

Hier S2S + Ptr: Inspired by hierarchical approaches for dialogue response generation (Serban et al., 2016), we consider a hierarchical variant of the S2S + Ptr model with two separate encoders: one for representing an individual utterance, and one for representing the whole discussion. We provide implementation details in Appendix B.

PLBART: Ahmad et al. (2021) proposed PLBART, which is pretrained on a large amount of code from GitHub and software-related natural language from StackOverflow, using BART-like (Lewis et al., 2020) training objectives. With fine-tuning, PLBART achieves state-of-the-art performance on many program and language understanding tasks. We fine-tune PLBART on our training set and evaluate its ability to comprehend bug report discus-

[4] We observe very low performance with extractive approaches, as shown in Appendix C.

	Model	BLEU	METEOR	ROUGE
Full	Copy Title	14.4$^{\|}$	13.1	24.4§
	S2S + Ptr	12.6	9.8	25.0‡
	Hier S2S + Ptr	12.4	9.6	24.1§
	PLBART	**16.6**	**14.5**	**28.3**
	PLBART (F)	14.2$^{\|}$	12.3	25.1‡
Filtr.	Copy Title	10.0*†	8.3	16.6
	S2S + Ptr	10.2*	7.5	20.1
	Hier S2S + Ptr	9.9†	7.4	19.6
	PLBART	**12.3**‡	9.9	21.1
	PLBART (F)	**12.3**‡	**10.2**	**21.9**

Table 3: Automated metrics. S2S + Ptr and Hier S2S + Ptr scores are averaged across 3 trials. Differences that are *not* statistically significant are indicated with matching symbols.

sions and generate descriptions of solutions.[5] Note that PLBART has a 1024 token limit. We use left truncation to keep the most recent content.

PLBART (F): Since PLBART is pretrained on a large amount of data, we can afford to reduce the fine-tuning data. So we fine-tune on only the filtered subset of the training set (cf. §3.2), to investigate whether fine-tuning on this "less noisy" sample can lead to improved performance.

4.2 Results: Automated Metrics

We use text generation metrics, BLEU-4 (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), and ROUGE-L (Lin, 2004). We compute statistical significance with bootstrap tests (Berg-Kirkpatrick et al., 2012) with $p < 0.05$. Results are in Table 3. On the full test set, PLBART significantly outperforms other models, demonstrating the value of pretraining on large amounts of data. PLBART (F) underperforms PLBART on the full test set. On the filtered subset, it either beats or matches PLBART.

Performance drops across models between the full and filtered test sets. The relatively high performance of the naive Copy Title baseline shows that simply copying or rephrasing the title performs well in many cases, particularly for the full test. The filtered subset is designed to remove uninformative reference descriptions that merely re-state the problem, as illustrated in Table 2 with filtered reference descriptions having higher percentages of novel n-grams, with respect to the title. Nonetheless, keywords relevant to the solution are often also in the title, so the Copy Title baseline still achieves reasonable scores on the filtered subset. Although automated metrics provide some signal, they emphasize syntactic similarity over semantic similarity. So, we conduct human evaluation.

[5]We focus on PLBART rather than vanilla BART because it achieves higher performance, as shown in Appendix D.

Model	Full	Filtered
Copy Title	8.1	6.0
S2S + Ptr	1.3*	1.2†
Hier S2S + Ptr	1.3*	1.2†
PLBART	11.9	10.5
PLBART (F)	**33.1**‡	**39.5**
All Poor	20.0	22.1
Insufficient Context	31.9‡	25.6

Table 4: Human evaluation results: Percent of annotations for which users selected predictions made by each model. This entails 160 annotations for the full test set, 86 of which correspond to examples in our filtered subset. Differences that are *not* significant are indicated with matching superscripts.

4.3 Results: Human Evaluation

Evaluators first read the title and discussion ($U_1 ... U_{t_g}$). For each example, they are shown predictions from the 5 models discussed in Section 4.1. From these, they must select one or more that are most informative towards resolving the bug. If all candidates are uninformative, they select a separate option: "All candidates are poor." There is also another option to indicate that there is insufficient context about the solution (§3.2), making it difficult to evaluate candidate descriptions. They also write a rationale for their response.

Since annotation requires not only technical expertise, but also high cognitive load and time commitment, it is hard to perform human evaluation on a large number of examples with multiple judgments per example. Similar to Iyer et al. (2016), we resort to having each example annotated by one user to obtain more examples. We recruited 8 graduate students with 3+ years of programming experience and familiarity with Java. They are not active contributors, so they will likely select the option of insufficient context more often than if they had a deeper understanding of the various software projects. However, it is difficult to conduct a user study at a similar scale with contributors. Nonetheless, there are developers aiming to become first-time contributors for a particular project (Tan et al., 2020). Our study better aligns with this use case.

Each user annotated 20 examples, leading to annotations for 160 unique examples in the full test set. In Table 4, we show that PLBART (F) substantially outperforms all other models, with users selecting its output 33.1% of the time. Even though the title typically only states a problem, users selected it 8.1% of the time. From rationales that users were asked to write, we found that there were cases in which the title not only posed the problem but also offered a solution. Users rarely preferred the output of S2S + Ptr and Hier S2S + Ptr as they

Model	Title ↓		$U_1...U_{t_g}$ only ↑	
	1	2	1	2
Copy Title	100.0	100.0	0.0	0.0
S2S + Ptr	64.8	37.1	31.6	25.3
Hier S2S + Ptr	60.3	34.2	38.7	26.1
PLBART	80.8	77.7	31.0	41.4
PLBART (F)	36.9	28.4	**52.8**	**42.3**
Reference	**32.7**	**22.2**	38.8	25.4

Table 5: Percent of unigrams and bigrams in the prediction (or reference) which appear in the title and in $U_1..U_{t_g}$ only (excluding the title), on the CS subset.

Model	Prediction
Copy Title	black screen appears when we seek over an ad group .
S2S + Ptr	fix black ads
Hier S2S + Ptr	fix seeking in ad tag
PLBART	suppress closing shutter when seeking over an ad group
PLBART (F)	suppress closing the shutter when seeking to an unprepared period
Reference	prevent shutter closing for within - window seeks to unprepared periods

Table 6: Model outputs for the example shown in Figure 1.

usually just rephrased the problem. PLBART also appears to be re-stating the problem in many cases; however, less often than other models.

Though we see similar trends across the full test set and the filtered subset, all models except PLBART (F) tend to perform worse on the filtered subset, as previously observed on automated metrics. Also, the average number of cases with insufficient context is lower for the filtered subset, confirming that we are able to reduce such cases through filtering. We find the results on the filtered data to align better with human judgment. By fine-tuning on the filtered training set, PLBART (F) learns to pick out important information from within the context and generate descriptions which reflect the solution rather than the problem.

4.4 Analysis

Of the 160 annotated examples, users found 109 to have sufficient context about the solution. We consider this the *context-sufficient subset (CS)*, which we will release for future research. To analyze how models exploit the provided context, we measure the percent of n-grams in the prediction which overlap with the title as well as $U_1...U_{t_g}$ (excluding n-grams already in the title) in Table 5. PLBART (F)'s predictions tend to have less n-gram overlap with the title and more overlap with the utterances. This suggests that this model predicts fewer uninformative descriptions which merely re-state the problem mentioned in the title and instead focuses on other content from the utterances.

In Table 6, we show model outputs for the example in Figure 1. SeqToSeq and Hier S2S + Ptr essentially rephrase aspects of the problem, which are described in the title. Both PLBART and PLBART (F) capture the solution, with PLBART (F) providing more information. When there is sufficient context, 62.4% of the time, either PLBART or PLBART (F) generates output that is informative towards bug resolution. While this demonstrates that fine-tuning this large, pretrained model on our data can be useful in supporting bug resolution in on-line discussions to some extent, it also shows that there is opportunity for improvement.

We manually inspected PLBART (F)'s outputs and associated user rationales. We observe that the model tends to perform better when the solution is clearly stated in 1-3 consecutive sentences (Table 7 (1) and (2)). When more complex synthesis is needed, it sometimes stitches together tokens from the input incorrectly (Table 7 (3)). Next, although the model picks up on information in the context, sometimes, it draws content from an elaboration of the problem from within the discussion rather than a formulation of the solution (Table 7 (4)). This demonstrates that it still struggles to disentangle content relevant to the solution from that about the problem. It also sometimes struggles to generate meaningful output when in-lined code is present, highlighting the challenge in bimodal reasoning about code and natural language (Table 7 (5)).

5 Supporting Real-Time Generation

Generating an informative description requires sufficient context about the solution being available in the discussion. In a real-time setting, this context is likely not immediately available but rather emerges as the discussion progresses, and we must wait until it becomes available to generate a solution description. However, the time step at which it becomes available (t_g) is not known beforehand, so we must instead predict it (t_p) in order to perform generation *during* ongoing discussions. For this, we consider classifying whether sufficient context is available upon each new utterance. In Figure 1, the solution is formulated in U_4, so the correct behavior is to predict the negative label at $t = 1, 2, 3$ and the positive label at $t = 4$. Once the positive label is predicted at t_p[6], the description is generated, conditioned on the title and $U_1...U_{t_p}$. We develop

[6]Classifications are not made at $t > t_p$. We leave generating at multiple time steps for future work.

	Title	PLBART (F)	Reference
(1)	Issue with dex: OIDC server is not available at the 'quarkus.oidc.auth-server-url' URL	fix trailing slash in auth - server url	strip trailing forward slash from oidc url
(2)	InvalidDataTypeException: UDATA contains value larger than Integer.MAX_VALUE DDR issue decoding lookswitch	fix bug in byte code dumper when tableswitch instruction precedes tableswitch instruction	fix interpretation of switch instructions in byte code dumper
(3)	Worldmap viewport changes when switching between dashboard pages	don ' t refresh widget grid when worldmap loses viewport	define key prop for map visualization to update map on dimension change
(4)	Workaround comments exist in opengrok-indexer/pom.xml file while the related issues are already fixed.	fix jflex - de / jflex # 705 (comment)	use jflex 1.8.2
(5)	Why subscribe with single action for onNext design to crush if error happened?	1 . x : fix subscription . subscribe ( ) to return observable . empty ( ) 2 . x : fix subscription . subscribe ( ) to return observable . empty ( )	fixed sonar findings

Table 7: Output of PLBART (F) for a sample of examples in the test set. Derived from: https://github.com/quarkusio/quarkus/issues/10227, https://github.com/eclipse-openj9/openj9/issues/9294, https://github.com/Graylog2/graylog2-server/issues/7997, https://github.com/oracle/opengrok/issues/3172, https://github.com/ReactiveX/RxJava/issues/637.

two systems for integrating classification with a generation model: *pipelined* and *joint trained*.

5.1 Pipelined System

We design an independent classifier built on PLBART's encoder. When a new utterance U_t is made in the discussion, we encode the context so far (the title and all utterances up to and including U_t). We take the final hidden state, e_t, as the context representation at t, which we feed e_t through a 3-layer classification head and apply softmax to classify whether or not sufficient context is available. We train to minimize cross entropy loss. At test time, we use the already trained PLBART (F) model to generate a solution description with context available at t_p.

5.2 Joint System

We initialize an encoder-decoder model from PLBART with an additional classification head (§5.1). The encoder is shared among the two tasks. When classifying whether sufficient context about the solution is available, there is likely specific solution-related content that contributes to predicting the positive label. So, classification may enhance encoder representations, improving content selection for generating solution descriptions.

Furthermore, having sufficient context correlates with whether it can be used to generate an *informative* description. So, the informativeness of a description that can be generated with the available context can provide signal for classifying whether that context is sufficient. Additionally, if sufficient context was not previously available at $t-1$ but becomes available at t, we expect an improvement in the informativeness of the descriptions generated at the two time steps. We represent these descriptions with the final decoder states at the two time steps, d_{t-1} and d_t. We concatenate e_t, d_{t-1}, and d_t to form the input into the classification head. For training loss, we sum the generation and classification losses across time steps $t_1 ... t_g$. Sufficient context for generation may not be available at $t < t_g$, so we mask generation loss for earlier time steps.

5.3 Evaluation Setup

We train on filtered data since we found this to improve performance. At test time, a system can generate a solution description at $t_p \leq t_g$, or it can fail to predict the positive label before or at t_g. After a commit/PR for fixing the bug is made at t_g, the state of the discussion changes, with possible mentions of the solution that is implemented. Since using this as context to generate a solution description can be considered "cheating," we do not make predictions for time steps after t_g. We treat this as the system *refraining* from generating after not finding sufficient context.

5.4 Results: Automated Metrics

The pipelined and joint systems refrained from generating 33.3-35.4% and 36.4-39.8% of the time respectively. We present automated metrics for the remaining cases in Table 8. We find that $t_g - t_p$ is between 1.69 and 1.85 for the pipelined system and between 1.81 and 1.97 for the joint system. While a system should wait until sufficient context is available, sometimes, the last couple utterances before the implementation do not add context about the solution but are personal exchanges (e.g., "Thanks",

		$t_p \leq t_g$	$t_g - t_p$	BLEU	METEOR	ROUGE
Pipelined	Full	$@t_p$	1.69	$14.3^{\ddagger}$	$12.4^{\S}$	$25.1^{\P}$
		$@t_g$	-	**14.4**‡	**12.5**§	**25.3**¶
	Filtr.	$@t_p$	1.85	12.5^{*}	10.1	21.7
		$@t_g$	-	**12.6***	**10.5**	**22.3**
Joint	Full	$@t_p$	1.81	13.1	11.4	$22.4^{\dagger}$
		$@t_g$	-	**13.2**	**11.7**	**22.5**†
	Filtr.	$@t_p$	1.97	11.7	9.5	19.3
		$@t_g$	-	**11.9**	**9.9**	**19.7**

Table 8: Automated metrics for combined systems when $t_p \leq t_g$. We compare the generated description $@t_p$ with that if the system had generated $@t_g$. Differences that are *not* statistically significant are indicated with matching superscripts.

		$t_g - t_p$	BLEU	METEOR	ROUGE
Full	Pipelined	2.09	**14.4**	**12.4**	**24.8**
	Joint	**1.86**	12.9	11.3	22.3
Filtr.	Pipelined	2.16	**12.4**	**10.0**	**21.0**
	Joint	**2.03**	11.4	9.2	18.7

Table 9: Performance at t_p on examples for which both systems predicted $t_p \leq t_g$ (614 of full and 304 of filtered test sets). All differences are statistically significant.

"I'll open a PR"). So, generating slightly before t_g is acceptable in some cases. Moreover, despite generating early in some cases, the generated output $@t_p$ achieves comparable performance to that $@t_g$, with respect to the generation metrics (BLEU, METEOR, and ROUGE).

Note that the numbers are not directly comparable across the two systems since the exact subset of examples for which $t_p \leq t_g$ varies between the two. In Table 9, we present results for the subset of examples for which both systems predict $t_p \leq t_g$. The joint system achieves lower average error ($t_g - t_p$) for classification while the pipelined system performs better on generation metrics.

5.5 Results: Human Evaluation

We also do human evaluation, for which we recruited 6 graduate students with 3+ years of Java experience. Each user evaluated outputs of the two systems for 20 random examples from the filtered test set. Users are given the same information as Section 4.3. If the system refrained from generating, we ask them if there is sufficient context about the solution at any time step $t \leq t_g$. Otherwise, we show them the generated description and ask if there is sufficient context about the solution at t_p and also to rate the informativeness of the description on a Likert scale: 1: incomprehensible, completely incorrect, irrelevant; 2: generic, rephrasing problem; 3: includes some useful information but does not capture the solution; 4: partially captures solution; 5: completely captures solution.

In the cases that the system generated a description, users found there to be sufficient context at t_p 39.0% and 33.8% of the time for the pipelined and joint systems, with average informativeness being 3.3 for both. This suggests that when sufficient context is available, these systems generate descriptions which can be useful for bug resolution.

Because a real-time system must act at a given time step agnostic to future activity, classifying *when to generate* is challenging. It should defer generation to later time steps if the optimal context is not available. Generating too early can result in output that is generic and re-states the problem. For the cases in which the system generated a description *without* sufficient context at t_p, the average informativeness ratings were 2.2 (pipelined) and 2.0 (joint). However, deferring generation for too long by expecting more context to emerge later also poses a risk. After the solution has already been implemented, it is too late for a generated description to be useful towards resolving the bug. In the cases that the pipelined and joint systems refrained from generating, there was sufficient context about the solution 34.2% and 37.0% of the time respectively.

Despite the pipelined and joint systems having nuanced differences, we find them to perform similarly. Through our evaluation of these systems, we demonstrate room for improvement, particularly for the classification component in determining the optimal time step for generation. We leave it to future work to develop more intricate end systems.

6 Related Work

Bug report summarization: To help developers gather information from bug reports, there is interest in automatic bug report summarization. Approaches for this are designed to generate holistic summaries of bug reports, with a summary being

25% of the length of the bug report (Liu et al., 2020b). We instead aim to generate a concise description that captures a specific aspect of the bug report. Next, bug report summaries are not widely available, so approaches for this task rely on unsupervised techniques (Li et al., 2018; Liu et al., 2020b) or supervision from a small amount of data (Rastkar et al., 2014; Jiang et al., 2016). Our approach for obtaining noisy supervision allows us to train supervised models on a large amount of data. Bug report summarization is a post hoc task, done after the bug has been resolved, to help developers address related bug reports in the future. In contrast, our goal is to help resolve the present bug report, so our system must learn *when* to perform generation during an ongoing discussion. Approaches for bug report summarization have been predominantly extractive whereas ours is abstractive. While we are interested in how bug report summarization techniques fair on our task, their implementations are not publicly available.

Commit message generation: Unlike the task of automatically generating commit messages to describe code changes that have already been made (Loyola et al., 2017; Xu et al., 2019), our system aims to generate natural language descriptions that can drive code changes.

Response triggering: Classifying when to generate a description relates to chatbots learning to respond at an appropriate time (Liu et al., 2020a) in dyadic conversations. The goal is to avoid interrupting a user who splits up an utterance across multiple turns. We instead consider multi-party dialogue in which an agent should wait until a specific type of content emerges in the discussion. Bohus and Horvitz (2011) studied turn-taking decisions in spoken dialogue systems, using audio-visual features, while ours is a text-based system.

Dialogue + software: We view our work as a step towards building a dialogue agent for streamlining software bug resolution. There has been minimal work in building interactive systems for this domain, with the exception of a few for tasks like query refinement (Zhang et al., 2020) and code generation (Chaurasia and Mooney, 2017; Yao et al., 2019). Wood et al. (2018) recently built a dialogue corpus through a "Wizard of Oz" experiment to study the potential of a Q&A assistant during bug fixing. Lowe et al. (2015) developed a dialogue corpus based on Ubuntu chat logs to study Q&A assistants for technical support. In contrast, our dataset is designed for building a collaborative agent that participates in multi-party conversations rather than one which answers directed questions.

7 Conclusion

We presented the novel task of generating concise natural language solution descriptions to guide developers in absorbing information relevant towards bug resolution from long discussions. We established benchmarks for this task using a dataset that we constructed with supervision derived from commit messages and pull request titles. Through automated and human evaluation, we demonstrated the utility of these models and also highlight their shortcomings, to encourage more research in exploring ways to address these challenges. We also simulated a real-time setting through two approaches for combining a generation model with a classification component for determining when sufficient context for generating an informative description emerges in an ongoing discussion. We believe this lays the groundwork for future work on building a dialogue agent that participates in bug report discussions to foster efficient resolution.

Acknowledgements

We would like to thank Tanya Goyal, Prasoon Goyal, Adrian Benton, and Eunsol Choi for early feedback on this work. We would also like to thank reviewers for their detailed comments and suggestions. This work was supported by NSF grant IIS-1850153, the Bloomberg Data Science Fellowship and a Google Faculty Research Award.

Ethics Statement

Our work aims to expedite bug resolution by mobilizing developers and guiding them in absorbing content in long discussions that is relevant towards implementing the solution. Through this, we hope to reduce the life span of software bugs and vulnerabilities that can significantly disrupt everyday operations. Our system is designed to *assist* developers and should not be considered as a replacement for the critical reasoning that is needed during bug resolution. Over-relying on this system to always alert developers when a solution has been recommended could have the opposite effect of causing delays in bug resolution for cases that the system is unable to handle. Additionally, if developers choose to rely solely on the system's generated description and ignore the discussion context, the solutions

they implement could potentially be incomplete or incorrect, if the system's output misses important details. Instead, developers should use the generated output to guide their focus and understanding as they read through the discussion.

To build our system, we used data from GitHub, in accordance with its acceptable use policy, and no additional permission was required. Namely, the policy states: "Researchers may use public, non-personal information from the Service for research purposes, only if any publications resulting from that research are open access."[7] We use only publicly available data and use it only for research purposes. Additionally, the data we used to train and evaluate models (and publicly release) does not contain personal information (e.g., usernames of users who authored utterances and linked mentions). We require that any future work using our dataset must abide by GitHub's official policy as well. For evaluation, we conducted human evaluation, for which participants willfully volunteered to be part of the study. They were not compensated for their participation.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *ACL*, pages 4998–5007.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *NAACL*, pages 2655–2668.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *ICLR*.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *ICML*, pages 2091–2100.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *ICLR*.

Deeksha Arya, Wenting Wang, Jin L. C. Guo, and Jinghui Cheng. 2019. Analysis and detection of information types of open source software issue discussions. In *ICSE*, page 454–464.

[7] https://docs.github.com/en/github/site-policy/github-acceptable-use-policies

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for MT evaluation with improved correlation with human judgments. In *Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72.

Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An empirical investigation of statistical significance in NLP. In *EMNLP*, pages 995–1005.

Dan Bohus and Eric Horvitz. 2011. Multiparty turn taking in situated dialog: Study, lessons, and directions. In *SIGDIAL*, pages 98–109.

Saikat Chakraborty and Baishakhi Ray. 2021. On multimodal learning of editing source code. In *ASE*, pages 443–455.

Shobhit Chaurasia and Raymond J. Mooney. 2017. Dialog for language to code. In *IJCNLP*, pages 175–180.

Günes Erkan and Dragomir R. Radev. 2004. Lexrank: Graph-based lexical centrality as salience in text summarization. *Journal of Artificial Intelligence Research*, 22(1):457–479.

Khashayar Etemadi and Martin Monperrus. 2020. On the relevance of cross-project learning with nearest neighbours for commit message generation. In *ICSE Workshops*, page 470–475.

Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured neural summarization. In *ICLR*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC*, pages 200–210.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *ACL*, pages 2073–2083.

Oskar Jarczyk, Blazej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. 2014. Github projects. Quality analysis of open-source software. In *SocInfo*, pages 80–94.

He Jiang, Jingxuan Zhang, Hongjing Ma, Najam Nazar, and Zhilei Ren. 2016. Mining authorship characteristics in bug repositories. *Science China Information Sciences*, 60:012107.

Rafael-Michael Karampatsis and Charles Sutton. 2020. How often do single-statement bugs occur? the manysstubs4j dataset. In *MSR*, page 573–577.

David Kavaler, Sasha Sirovica, Vincent Hellendoorn, Raul Aranovich, and Vladimir Filkov. 2017. Perceived language complexity in github issue discussions and their effect on issue resolution. In *ASE*, pages 72–83.

Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. 2015. Issue dynamics in github projects. In *International Conference on Product-Focused Software Process Improvement*, page 295–310.

Oleksii Kononenko, Tresa Rose, Olga Baysal, Michael Godfrey, Dennis Theisen, and Bart de Water. 2018. Studying pull request merges: A case study of shopify's active merchant. In *ICSE: Software Engineering in Practice Track*, pages 124–133.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *ACL*, pages 7871–7880.

Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. 2018. Unsupervised deep bug report summarization. In *ICPC*, page 144–155.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81.

Che Liu, Junfeng Jiang, Chao Xiong, Yi Yang, and Jieping Ye. 2020a. Towards building an intelligent chatbot for customer service: Learning to respond at the appropriate time. In *SIGKDD*, page 3377–3385.

Haoran Liu, Yue Yu, Shanshan Li, Yong Guo, Deze Wang, and Xiaoguang Mao. 2020b. BugSum: Deep context understanding for bug report summarization. In *ICPC*, page 94–105.

Yang Liu and Mirella Lapata. 2019. Text summarization with pretrained encoders. In *EMNLP-IJCNLP*, pages 3730–3740.

Ryan Lowe, Nissan Pow, Iulian Serban, and Joelle Pineau. 2015. The Ubuntu dialogue corpus: A large dataset for research in unstructured multi-turn dialogue systems. In *SIGDIAL*, pages 285–294.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. In *ACL*, pages 287–292.

Ramesh Nallapati, Feifei Zhai, and Bowen Zhou. 2017. Summarunner: A recurrent neural network based sequence model for extractive summarization of documents. In *AAAI*, page 3075–3081.

Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2012. Multi-layered approach for recovering links between bug reports and fixes. In *FSE*, pages 63:1–63:11.

Pengyu Nie, Jiyang Zhang, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2022. Impact of evaluation methodologies on code summarization. In *ACL*, page (To Appear).

Yuki Noyori, Hironori Washizaki, Yoshiaki Fukazawa, Keishi Ooshima, Hideyuki Kanuka, Shuhei Nojiri, and Ryosuke Tsuchiya. 2019. What are good discussions within bug report comments for shortening bug fixing time? In *International Conference on Software Quality, Reliability and Security*, pages 280–287.

Sebastiano Panichella, Gerardo Canfora, and Andrea Di Sorbo. 2021. "won't we fix this issue?" qualitative characterization and automated identification of wontfix issues on GitHub. *Information and Software Technology*, 139:106665.

Sheena Panthaplackel, Milos Gligoric, Raymond J. Mooney, and Junyi Jessy Li. 2020a. Associating natural language comment and source code entities. In *AAAI*.

Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020b. Learning to update natural language comments based on code changes. In *ACL*, pages 1853–1868.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *ACL*, pages 311–318.

Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2014. Automatic summarization of bug reports. *TSE*, 40(4):366–380.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *ACL*, pages 1073–1083.

Iulian V. Serban, Alessandro Sordoni, Yoshua Bengio, Aaron Courville, and Joelle Pineau. 2016. Building end-to-end dialogue systems using generative hierarchical neural network models. In *AAAI*, page 3776–3783.

Jeniya Tabassum, Mounica Maddela, Wei Xu, and Alan Ritter. 2020. Code and named entity recognition in StackOverflow. In *ACL*, pages 4913–4926.

Xin Tan, Minghui Zhou, and Zeyu Sun. 2020. A first look at good first issues on github. In *ESEC/FSE*, page 398–409.

Yuqing Tang, Chau Tran, Xian Li, Peng-Jen Chen, Naman Goyal, Vishrav Chaudhary, Jiatao Gu, and Angela Fan. 2020. Multilingual translation with extensible multilingual pretraining and finetuning. *ArXiv*, abs/2008.00401.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *NeurIPS*, volume 30.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *NeurIPS*, pages 2692–2700.

Andrew Wood, Paige Rodeghero, Ameer Armaly, and Collin McMillan. 2018. Detecting speech act types in developer question/answer conversations during bug repair. In *ESEC/FSE*, page 491–502.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *IJCAI*, pages 3975–3981.

Ziyu Yao, Yu Su, Huan Sun, and Wen-tau Yih. 2019. Model-based interactive semantic parsing: A unified framework and a text-to-SQL case study. In *EMNLP*, pages 5447–5458.

Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A systematically mined question-code dataset from Stack Overflow. In *WWW*, pages 1693–1703.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from Stack Overflow. In *MSR*, pages 476–486.

Neng Zhang, Qiao Huang, Xin Xia, Ying Zou, David Lo, and Zhenchang Xing. 2020. Chatbot4QR: Interactive query refinement for technical question retrieval. *TSE*.

Ruqing Zhang, Jiafeng Guo, Yixing Fan, Yanyan Lan, Jun Xu, and Xueqi Cheng. 2018. Learning to control the specificity in neural response generation. In *ACL*, pages 1108–1117.

Guoliang Zhao, Daniel Alencar da Costa, and Ying Zou. 2019. Improving the pull requests review process using learning-to-rank algorithms. *Empirical Software Engineering*, 24:2140–2170.

A Data Cleaning

We focus on closed bug reports from the top 1,000 Java projects (in terms of number of stars), as a way of identifying well-maintained projects (Jarczyk et al., 2014). We require there to be at least two distinct "actors" in the discussion, in which the actor can either be a developer who makes an utterance in the discussion or an actor who implements the solution through a commit or pull request. We discard examples in which the reference description is identical to the title (disregarding stopwords), as these are cases in which either the reference description only states the problem and is uninformative or the title already puts forth a solution (in which case a generated description would not be useful). We remove examples with commits or pull requests which simultaneously address multiple bug reports.

We mined 141,389 issues (from 770 of the top 1,000 projects). After applying heuristics, we get 35,010 (from 525 projects), which will be released. Of these, 16,899 pertain to bugs and 18,111 pertain to non-bugs. From the 16,899 bug-related issues, we focus on the 12,328 issues with a single commit message/PR title. We explain our reasoning for discarding examples linked to multiple commits and/or pull requests in Section 3.1. However, such examples (which are available in the data we release) can be useful for supporting generating descriptions at multiple time steps in future work.

From an example's description, we remove references to issue and pull request numbers, as they do not contribute to the meaning and are instead used as identifiers for organizational purposes.

B Details of Hier S2S + Ptr Model

We encode U_t using a transformer-based encoder and feed the contextualized representation of its first token (<UTTERANCE_START>) into the RNN-based discussion encoder to update the *discussion state*, s_t. When encoding U_t, we also concatenate s_{t-1} to embeddings, to help the model relate U_t with the broader context of the discussion. Note that we treat the title as U_0 in the discussion. This process continues until U_{t_g} is encoded, at which point all accumulated token-level hidden states are fed into a transformer-based decoder to generate the output.

Unlike the S2S + Ptr model which is designed to reason about the full input at once, this approach reasons step-by-step, with self-attention in the utterance encoder only being applied to tokens within the same utterance. Since the input context for this task is often very large, we investigate whether it is useful to break down the encoding process in this way. We also equip this model with a pointer generator network.

C Additional Generation Baselines

We considered additional baselines; however, since they were performing much lower than other approaches (on wide statistically significant margins), we chose to exclude them from the main paper. We briefly describe these baselines below.

C.1 Extractive Baselines

Supervised Extractive: Using a greedy approach for obtaining noisy extractive summaries (Nallapati et al., 2017), we train a supervised extractive summarization model, similar to (Liu and Lapata, 2019).

LexRank: We use LexRank (Erkan and Radev, 2004), an unsupervised graph-based extractive summarization approach. We extract 1 sentence with threshold 0.1.

$\mathbf{U_1}$ (Lead 1): This entails simply taking the first sentence of the first utterance, intended to simulate the Lead-1 baseline that is commonly used in summarization.

$\mathbf{U_1}$ (Lead 3): This entails simply taking the first 3 sentences of the first utterance, intended to simulate the Lead-3 baseline that is commonly used in summarization.

$\mathbf{U_{t_g}}$: Since some part of the solution is often mentioned within U_{t_g}, we copy this utterance.

$\mathbf{U_{t_g}}$ (Lead 1): Since the length of an utterance is quite different than that of a description (Table 1), we extract only the lead sentence of U_{t_g}.

$\mathbf{U_{t_g}}$ (Lead 3): For the reason stated above, we also apply the Lead-3 baseline to this utterance.

$\mathbf{U_{t_g}}$ (Last sentence): Rather than extracting the lead sentence, we extract the last sentence of U_{t_g}.

$\mathbf{U_{t_g}}$ (Last 3 sentences): Rather than extracting the lead 3 sentences, we try extracting the last 3 sentences of U_{t_g}.

C.2 Retrieval Baselines

Retrieval (Title-Title): Using TF-IDF, we compute cosine similarity between the test example's title and titles in the training set, to identify the closest training example, from which we take the description.

	Model	BLEU	METEOR	ROUGE-1	ROUGE-2	ROUGE-L
Full	Supervised Extractive	0.537	0.536	0.807	0.010	0.767
	LexRank	2.252	1.851	2.629	0.061	2.470
	U_1 (Lead 1)	4.793	6.537	10.077	2.534	8.752
	U_1 (Lead 3)	3.085	7.955	9.778	2.303	8.687
	U_{t_g}	2.842	5.425	7.426	1.363	6.712
	U_{t_g} (Lead 1)	4.028	4.453	7.736	1.451	6.889
	U_{t_g} (Lead 3)	3.189	5.692	8.153	1.504	7.359
	U_{t_g} (Last sentence)	3.475	3.480	6.089	0.930	5.476
	U_{t_g} (Last 3 sentences)	3.234	5.082	7.525	1.287	6.787
	Retrieval (Title-Title)	6.866	4.497	11.517	1.281	10.748
	Retrieval (Title-Desc)	8.763	6.167	15.965	2.426	14.776
	Project Retrieval (Title-Title)	7.442	4.709	11.501	1.49	10.943
	Project Retrieval (Title-Desc)	9.118	6.299	14.949	2.232	14.089
	Copy Title	14.358	13.142	27.361	11.539	24.427
	S2S + Ptr	12.583	9.838	27.589	4.258	25.024
	Hier S2S + Ptr	12.365	9.564	26.785	3.672	24.084
	PLBART	**16.551**	**14.484**	**31.564**	**11.549**	**28.295**
	PLBART (F)	14.188	12.302	27.443	8.349	25.128
Filtr.	Supervised Extractive	0.711	0.653	1.084	0.005	1.029
	LexRank	2.442	1.946	2.843	0.066	2.637
	U_1 (Lead 1)	4.951	6.207	9.881	1.938	8.553
	U_1 (Lead 3)	3.055	7.907	9.890	1.875	8.777
	U_{t_g}	2.899	6.045	8.081	1.507	7.346
	U_{t_g} (Lead 1)	4.406	4.808	8.424	1.507	7.590
	U_{t_g} (Lead 3)	3.356	6.257	8.894	1.681	8.060
	U_{t_g} (Last sentence)	3.515	3.961	6.547	1.046	5.868
	U_{t_g} (Last 3 sentences)	3.345	5.722	8.200	1.460	7.448
	Retrieval (Title-Title)	6.117	3.727	9.546	0.711	8.965
	Retrieval (Title-Desc)	6.998	4.542	12.082	1.257	11.410
	Project Retrieval (Title-Title)	6.646	4.195	9.603	1.273	9.255
	Project Retrieval (Title-Desc)	7.593	5.064	11.895	1.638	11.328
	Copy Title	9.962	8.291	18.538	4.943	16.641
	S2S + Ptr	10.168	7.521	21.846	2.278	20.116
	Hier S2S + Ptr	9.893	7.369	21.562	2.131	19.649
	PLBART	**12.319**	9.877	23.419	5.452	21.097
	PLBART (F)	12.266	**10.218**	**23.786**	**5.712**	**21.857**

Table 10: Comparing models in main paper with low-performing baselines for generating solution descriptions. Scores for Supervised Extractive are averaged across three trials.

Retrieval (Title-Desc): Using TF-IDF, we compute cosine similarity between the test example's title and *descriptions* in the training set, to identify the closest training example, from which we take the description.

Project Retrieval (Title-Title): Using TF-IDF, we compute cosine similarity between the test example's title and titles *for the same project* in the training set, to identify the closest training example, from which we take the description.

Project Retrieval (Title-Desc): Using TF-IDF, we compute cosine similarity between the test example's title and descriptions for the same project in the training set, to identify the closest training example, from which we take the description.

C.3 Baseline Results

We present baseline results in Table 10. In addition to the metrics used in the main paper, we report ROUGE-1 and ROUGE-2. All of these baselines substantially underperform models presented in the main paper, especially the Supervised Extractive model. We believe this model performs so poorly due to noise in the supervision and because the extracted summaries are longer and structured differently than the reference descriptions in our dataset. Additionally, there are many examples in which the model does not select a single sentence from the input, resulting in the prediction being the empty string. LexRank also performs poorly in terms of automated metrics against the reference description. This unsupervised approach aims to identify a “centroid" sentence that summarizes the full input context and is not designed to specifically focus on solution-related context.

All baselines that extract a whole utterance or sentences from specific utterances perform poorly, demonstrating the need for content selection from the broader context and content synthesis rather than relying on simple heuristics to produce a description of the solution. We find that the retrieval baselines tend to achieve higher scores, as retrieved

	Model	BLEU	METEOR	ROUGE-1	ROUGE-2	ROUGE-L
Full	mBART base (randomly initialized)	9.978	6.976	17.000	2.498	15.744
	mBART large	15.251	12.503	28.522	9.520	26.109
	BART base	14.226	11.522	26.957	8.864	24.746
	PLBART	**16.551**	**14.484**	**31.564**	**11.549**	**28.295**
Filtr.	mBART base (randomly initialized)	8.819	6.151	14.870	2.011	13.574
	mBART large	11.663	9.233	22.295	5.159†	20.458
	BART base	10.820	8.583	21.247	5.055†	19.537
	PLBART	**12.319**	**9.877**	**23.419**	**5.452**	**21.097**

Table 11: Comparing performance of BART-based models. Training/fine-tuning is done with our full training set. Differences that are *not* statistically significant are shown with matching symbols.

descriptions are from the same distribution as the reference descriptions. However, these numbers are still much lower than those in the main paper.

D BART Models

We use PLBART (Ahmad et al., 2021), which was pretrained on large amounts of code from GitHub and software-related natural language from StackOverflow. Compared to other pretrained models, fine-tuning PLBART achieves higher performance for various NL+code tasks, including code summarization, code generation, code translation, and code classification. Since our task also requires reasoning about code and technical text, we choose PLBART over other pretrained models in our work. We present automated metrics for PLBART and PLBART (F) in Table 3. The average length of PLBART's output is 9.0 and 8.6 tokens on the full and filtered test sets respectively, while it is 9.3 and 9.4 for PLBART (F).

For completion, we compare against BART-based models which are not pretrained on code or technical text. First, we consider mBART base (multilingual BART) (Tang et al., 2020), which is the underlying architecture of PLBART. Without pretraining (randomly initializing the same architecture), performance is very low, as shown in Table 11. The publicly released pretrained mBART model, which is pretrained on non-technical natural language, does not use the base architecture but rather large. We also fine-tune this model on our training set but find that it achieves lower performance than PLBART. Finally, we compare against BART base (Lewis et al., 2020), which is also pretrained on non-technical natural language. Again, this model underperforms PLBART. Because PLBART's performance is higher, we choose to focus on this model in our work.

E Human Evaluation Setup

In the user study, users are shown the title of the bug report, all utterances up till (and including) U_{t_g}, and the reference description in our dataset for the given example. We choose to provide this as a manual suggestion to help guide users in better understanding a bug report, for a software project with which they have minimal familiarity. However, we state in our instructions that this is merely provided for reference and is not necessarily the exact and only valid answer.

Next, we show them up to 5 model predictions and ask them to "select the one(s) which add(s) the most amount of useful information that will help resolve the bug, beyond just re-stating the problem itself." Note that these are presented in random order (per example), without any identifying information about the underlying models that generated them. We explain that we consider a description to be informative if it provides content that will be useful towards *fixing* the issue, beyond just rephrasing the problem itself. And we encourage users to select candidates based on content that is informative, rather than focusing on exact phrasing. If all candidates appear to be poor (completely unrelated to the resolving the bug, uninformative, incomprehensible, or plain wrong), users are asked to select another option: "All candidates are poor." If there is no useful information towards resolving the bug in the context and they are unable to evaluate candidate descriptions, they are asked to select another option: "The context does not have any useful information for resolving the bug." They must also justify their selection by writing a brief rationale.

This is a challenging task, as it requires reading through and reasoning about a large amount of text to evaluate each example. To prepare annotators, we first present a set of training examples and a training video in which we demonstrate how the task should be completed.

F Analyzing CS Subset

The CS subset consists of 109 examples from the test set spanning 45 projects, with average $T = 4.1$ and $t_g = 3.2$. We present automated metrics for

Model	BLEU	METEOR	ROUGE
Copy Title	12.6	12.2¶	22.1
S2S + Ptr	11.6	8.9	23.1
Hier S2S + Ptr	12.0	9.0	22.9
PLBART	**14.6**	**13.2**	**26.0**
PLBART (F)	14.2	12.3¶	25.1

Table 12: Automated metrics for generation on CS subset. Differences that are *not* statistically significant are indicated with matching symbols.

this subset in Table 12. Results are analogous to the full test set, except that the numbers are generally lower for all models other than for PLBART (F), which achieves consistent performance. PLBART (F) slightly underperforms PLBART on automated metrics overall. However, this is because these metrics are computed against the single reference description, which could diverge from how the solution is formulated in the discussion since the developer could have written an uninformative/generic description. To do more fine-grained analysis, in Figure 2, we plot automated metrics for varying percentages of token overlap between the reference description and $U_1 ... U_{t_g}$ (excluding tokens already present in the title which have been used to state the problem). Higher overlap suggests that the reference description draws more content from within the discussion. For higher percentages, PLBART (F) generally achieves higher scores against the reference than PLBART and all other models, indicating that this model is better at gathering information from within the discussion. In Table 13, we supplement the n-gram analysis from Section 4.4.

G Classification Performance

To benchmark performance on the classification task for determining when sufficient context is available for generating an informative description, we consider some simple baselines. We observe that there are many cases in which $t_g = 1, 2$, i.e., the solution is implemented immediately after the first or second utterance. So, we include the FIRST baseline which always predicts a positive label at $t = 1$, and SECOND which predicts negative at $t = 1$ and positive at $t = 2$, if $t_g \geq 2$ (otherwise it never predicts positive).

We include the RAND (uni) baseline which progresses through the discussion, randomly deciding between the positive and negative label after each utterance, based on a uniform distribution. We also include RAND (dist), which instead uses the probability distribution of labels at the example-level estimated from the filtered training set (pos = $\frac{1}{N}\sum_{n=1}^{N}\frac{1}{t_g}$=0.510, neg = 0.490). Results are averaged across 3 trials. We present results in Table 14.

H Reproducibility Checklist

H.1 Validation Performance

We report performances on the full validation set. Results for the generation task are in Table 15.

I Hyperparameters

All neural models were implemented using PyTorch. For S2S + Ptr and Hier S2S + Ptr, we use a batch size of 8, an initial learning rate of 3e-05, and a dropout rate of 0.2. Our transformer models have 4 encoder and decoder layers, 4 heads in multi-head attention, a hidden size of 64, and feedforward hidden size 256. We use Adam as the optimizer and have a learning rate scheduler with gamma 0.95 which decays after an epoch if the validation loss has not improved. We use early stopping with patience 5 during training.

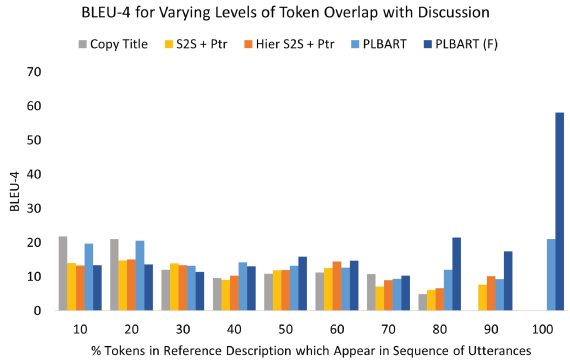
For classification, the classification head consists of a linear layer (dimension 768), followed by a tanh non-linear layer, and a final linear projection layer (dimension 2). When computing cross entropy loss for classification, we weight the positive and negative labels using the inverse of the class proportion to handle class imbalance (1.70 and 0.71 respectively). For the joint model, loss for a given example is computed as follows, with $\lambda_1 = 0.8, \lambda_2 = 0.2$ (tuned on validation data).

$$L = \lambda_1 L_{gen}(t_g) + \lambda_2 \sum_{t=1}^{t=t_g} L_{class}(t)$$

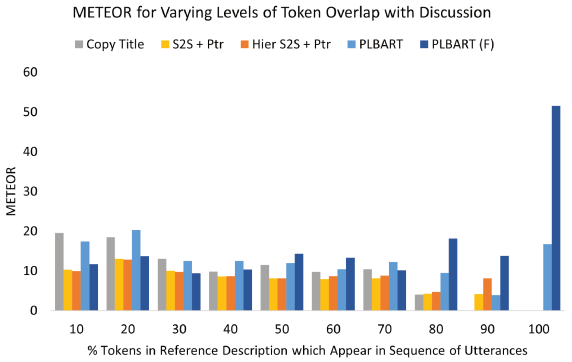
.

I.1 Tuning

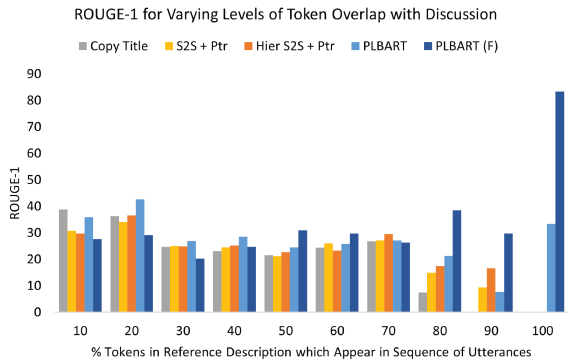
For S2S + Ptr and Hier S2S + Ptr, hyperparameters are tuned manually. For batch size, we consider {8,16,32}, learning rate {1e-03, 1e-04, 3e-05}, dropout {0.1, 0.2, 0.4, 0.5, 0.6}, encoder/decoder layers {2, 4, 6, 8}, number of heads {2, 4, 8}, hidden sizes {32, 62, 128}, and feedforward dimensions {128, 256, 512}. These hyperparameters are tuned on validation data, using the text generation metrics mentioned in Section 4.2 for generation. For tuning, we do not do grid search but rather compare performances between models trained with identical configurations, with the exception of a single parameter. Therefore, the number of hyperparameter tuning runs scales linearly. We ran



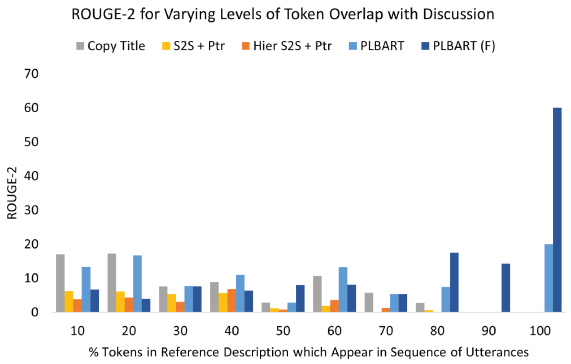
(a) BLEU-4



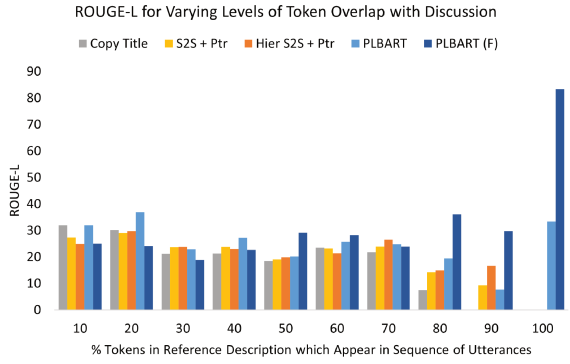
(b) METEOR



(c) ROUGE-1



(d) ROUGE-2



(e) ROUGE-L

Figure 2: Metrics for CS subset, with buckets corresponding to the % of tokens in reference description which also appear in $U_1 ... U_{t_g}$ (disregarding title tokens). Bucket 10 corresponds to [0, 10)%, 20 corresponds to [10, 20)%, etc.

	Model	Title ↓				$U_1...U_{t_g}$ only ↑			
		1	2	3	4	1	2	3	4
Full	Copy Title	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
	S2S + Ptr	65.6	34.4	39.3	46.5	28.6	24.9	27.0	25.0
	Hier S2S + Ptr	60.2	33.9	41.1	50.4	37.4	27.9	28.3	29.2
	PLBART	79.3	75.0	72.5	71.7	30.7	34.8	34.6	**39.9**
	PLBART (F)	43.2	37.4	38.3	43.1	**47.1**	**38.1**	**35.6**	37.2
	Reference	**35.1**	**30.9**	**33.5**	**37.7**	34.5	22.2	22.2	25.3
Filtered	Copy Title	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
	S2S + Ptr	64.5	33.8	39.1	38.3	29.4	25.3	23.8	0.0
	Hier S2S + Ptr	58.4	33.3	39.3	45.7	40.4	28.4	30.0	29.2
	PLBART	76.9	73.4	71.1	70.4	34.0	37.0	36.3	**41.2**
	PLBART (F)	38.4	33.9	35.2	40.7	**51.0**	**40.0**	**36.6**	38.1
	Reference	**23.7**	**18.6**	**18.4**	**16.3**	40.1	22.8	21.4	23.0
CS	Copy Title	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
	S2S + Ptr	64.8	37.1	38.5	22.5	31.6	25.3	33.1	25.0
	Hier S2S + Ptr	60.3	34.2	37.9	28.3	38.7	26.1	29.2	0.0
	PLBART	80.8	77.7	72.8	70.3	31.0	41.4	37.0	**50.0**
	PLBART (F)	36.9	28.4	30.8	34.1	**52.8**	**42.3**	**39.4**	45.0
	Reference	**32.7**	**22.2**	**26.2**	**35.6**	38.8	25.4	23.1	27.1

Table 13: Percent of unigrams, bigrams, trigrams and 4-grams in the prediction (or reference) which appear in the title and in $U_1..U_{t_g}$ only (excluding the title). Lower is better for the title and higher is better for $U_1..U_{t_g}$ only.

		First	Second	Rand (uni)	Rand (dist)	Pipelined	Joint
Full	(↑) $\mathbf{t_p} \leq \mathbf{t_g}$	100.0%	70.5%	76.0%	77.1%	66.7%	60.2%
	(↓) $\mathbf{t_g} - \mathbf{t_p}$	2.2	2.1	2.2	2.2	1.7	1.8
Filtr.	(↑) $\mathbf{t_p} \leq \mathbf{t_g}$	100.0%	76.2%	79.4%	80.1%	64.6%	63.6%
	(↓) $\mathbf{t_g} - \mathbf{t_p}$	2.6	2.4	2.5	2.5	1.9	2.0

Table 14: Percent of time $t_p \leq t_g$ and for these particular cases, the mean absolute error between t_g and t_p.

Model	BLEU-4	METEOR	ROUGE-1	ROUGE-2	ROUGE-L
Copy Title	15.223	13.645	28.088	12.322	25.341
S2S + Ptr	12.896	10.241	27.757	4.571	25.921
Hier S2S + Ptr	12.758	10.119	28.722	3.934	25.275
PLBART	**16.924**	**14.979**	**32.152**	**12.124**	**29.623**
PLBART (F)	15.059	13.057	29.107	9.111	26.710

Table 15: Scores for generation @ t_g on the 1,232 examples in the full validation set.

Model	Train	Eval	Epoch
S2S + Ptr	2:56:19	0:01:12	52.0
Hier S2S + Ptr	4:47:34	0:01:22	51.0
PLBART (fine-tuning)	0:32:07	0:00:25	11.0
PLBART (F) (fine-tuning)	0:26:08	0:00:28	15.0
Pipelined system (classifier only)	2:12:48	0:02:09	12.0
Jointly trained combined system	6:25:01	0:15:06	22.0

Table 16: Average training time, inference time, and number of epochs. Format for time is H:M:S.

each configuration once. For PLBART-based models, we use the same configurations as the scripts released by Ahmad et al. (2021).

I.2 Random Seeds

For the randomly initialized models, random seeds are set according to the timestamp, and we average results across 3 trials. For S2S + Ptr, the seeds were: 1620001129, 1620001158, and 1620004022. For Hier S2S + Ptr, the seeds were: 1620001125, 1620001159, and 1620004024.

J Statistical Significance Testing

We compute statistical significance using bootstrap tests (Berg-Kirkpatrick et al., 2012) with $p < 0.05$ and 10,000 samples of size 5,000 each.

J.1 Running Times

Table 16 reports average training time, inference time, and # epochs for the various models considered in this work. The PLBART-based models were trained/fine-tuned on NVIDIA DGX GPUs (32 GB) and all other models were trained and evaluated using on GeForce GTX Titan GPUs (8 GB). All models used single-GPU training.