Empirical Software Engineering (2025) 30:45
https://doi.org/10.1007/s10664-024-10594-x

Explainable automated debugging via large language model-driven scientific debugging

Sungmin Kang[1] · **Bei Chen**[2] · **Shin Yoo**[1] · **Jian-Guang Lou**[2]

Accepted: 12 November 2024

Abstract

Automated debugging techniques have the potential to reduce developer effort in debugging. However, while developers want rationales for the provided automatic debugging results, existing techniques are ill-suited to provide them, as their deduction process differs significantly froof human developers. Inspired by the way developers interact with code when debugging, we propose Automated Scientific Debugging (AutoSD), a technique that prompts large language models to automatically generate hypotheses, uses debuggers to interact with buggy code, and thus automatically reach conclusions prior to patch generation. In doing so, we aim to produce explanations of how a specific patch has been generated, with the hope that these explanations will lead to enhanced developer decision-making. Our empirical analysis on three program repair benchmarks shows that AutoSDperforms competitively with other program repair baselines, and that it can indicate when it is confident in its results. Furthermore, we perform a human study with 20 participants to evaluate AutoSD-generated explanations. Participants with access to explanations judged patch correctness more accurately in five out of six real-world bugs studied. Furthermore, 70% of participants answered that they wanted explanations when using repair tools, and 55% answered that they were satisfied with the Scientific Debugging presentation.

Communicated by: Andrea Stocco, Matteo Biagiola, Vincenzo Riccio, Foutse Khomh, Nicolás Cardozo, Dongwan Shin

Sungmin Kang participated in this work during an internship at Microsoft Research Asia.

This article belongs to the Topical Collection: *Special Issue on Innovations in Software System Testing with Deep Learning*

✉ Shin Yoo
shin.yoo@kaist.ac.kr

Sungmin Kang
sungmin.kang@kaist.ac.kr

Bei Chen
beichen1019@126.com

Jian-Guang Lou
jlou@microsoft.com

[1] KAIST, Daejeon, South Korea

[2] Microsoft Research Asia, Beijing, China

Published online: 18 December 2024

Keywords Automated Program Repair · Machine Learning

1 Introduction

Automated debugging techniques, such as Fault Localization (FL) or Automated Program Repair (APR), aim to help developers by automating the debugging process in part. Due to the significant amount of developer effort that goes into debugging (Zeller 2009), automated debugging is a research topic of significant interest (Liu et al. 2020): many papers are published every year (Monperrus 2020), and the field is mature enough to see adoption by industry (Kirbas et al. 2021; Marginean et al. 2019).

Regarding the practical adoption of these techniques, a body of literature surveying developer expectations on automated debugging has consistently highlighted that, as much as strong performance on software engineering tasks is important, so is supporting information that helps developers judge the results. For example, Kochhar et al. (2016) perform a study of developer expectations on fault localization, and find that more than 85% of developers agree that the ability to provide rationale is important. Further, Kirbas et al. (2021) note that some developers responded negatively to automated program repair results, citing that they would come "out of the blue". Such findings suggest that strong automated debugging results may not be acceptable on their own, and may need supporting information that helps *explain* the results.

Despite the consistent request for explainable processes for automated results, to the best of our knowledge explainable automated debugging techniques can be difficult to come by. For example, in the living review of APR compiled by Monperrus updated in August 2022 (Monperrus 2020), the word 'explain' appears only in one position paper (Monperrus 2019), revealing that the critical research on how to explain repair suggestions to developers is under-explored. We argue that this is in part because existing automated debugging techniques reason in starkly different ways to humans. Whereas existing automated debugging techniques will reduce a search space (Jiang et al. 2018) and try multiple solutions to find results that are correlated with the location and fix of a bug (Moon et al. 2014), human developers will generally utilize debuggers and `print` statements to interact with the buggy code, understand its behavior and in turn make a patch based on such observations (Siegmund et al. 2014). That is, the reasoning traces (Lim et al. 2009) of existing automated debugging processes are so different from those of developers, that suggesting them may contribute little to the understanding of a generated patch.

As a step towards automated debugging techniques that can generate explanations that help developers, we propose AutoSD, which represents a novel pipeline to combine bug-related information into generate legible explanations for bugs along with patches. To do so, AutoSDleverages Large Language Models (LLMs) and a debugger interface to automatically emulate the Scientific Debugging (SD) process for developers proposed by Zeller (2009), which suggests to formulate and verify hypotheses about the bug in the process of debugging. In line with this, AutoSDprompts an LLM to automatically generate hypotheses about what is causing the bug, along with a debugger script that would test the hypotheses. AutoSDthen executes the suggested debugger command and provides the LLM with the result; based on this, the LLM finally decides whether the hypothesis was met, and predicts if the debugging process is done, or additional investigation is required. The intermediate debugging text generated as a result can naturally be presented as an *explanation* describing how AutoSDreached its conclusion. Emulating Scientific Debugging has ideal properties for

explainable debugging: notably, as existing work identifies that developers use the principles of Scientific Debugging to debug even without formal training (Siegmund et al. 2014), the explanations could help inform or augment the thought process of developers.

We empirically evaluate AutoSDby first evaluating it on three program repair benchmarks. Our results indicate AutoSDcan achieve competitive repair results to non-explainable APR techniques. In terms of practical usage, precision is an important factor (Xiong et al. 2017); we find that for cases when AutoSDindicates it had collected enough information for debugging, repair performance is in fact higher, potentially reducing developer inspection effort. As language models become more capable, the repair performance of AutoSDrapidly increases as well, demonstrating the potential of AutoSD. We further perform a user study on Python developers involving 20 participants, including six professional developers, under a realistic APR application setting: reviewing patches for acceptance. Our results demonstrate that the debugging traces generated by AutoSDenhance developer accuracy in terms of assessing whether the patch is correct for 83% of the real-world bugs studied, while the amount of time in which developers judged the patch was roughly constant; these results suggest that humans benefit from the automatically generated patch explanations of AutoSD. Furthermore, 70% of participants responded that they would see explanations as an important factor when using APR tools, and 55% were satisfied with the Scientific Debugging formulation of AutoSD.

Our demonstration that LLMs can emulate loosely-structured workflows such as Scientific Debugging, which were originally intended for humans, and consequently improve explainability, has a greater implication for many software engineering processes as well. Our work can also be seen as an combining recent advances in LLM research, in which LLMs are combined with traditional tool use, with existing software engineering processes - AutoSDcombines tools that developers often use (symbolic debuggers), which subsequently allows LLMs to debug in a human-like manner. This opens the possibility that other software processes designed for humans in mind, such as test scripts for manual testers (Haas et al. 2021) or code review processes (Rigby and Bird 2013), may also be performed in an explainable manner, improving the usability of automated software engineering tools in multiple areas.

Overall, our contribution may be summarized as:

- We identify that explainable automated debugging may be achieved by LLMs emulating developer processes, and as a demonstration propose AutoSD, which uses LLMs to emulate Scientific Debugging (Zeller 2009);
- We perform a comprehensive set of empirical analyses on three APR benchmarks, demonstrating that AutoSDcan achieve significant APR performance while also generating explanations and indicating confidence in its results as a natural byproduct of its patch-generation process;
- We conduct a developer study on AutoSD, based on a realistic scenario of patch review, and demonstrate explanations from AutoSDcan aid developers in decision-making;
- We further solicit feedback from users regarding repair explanations, presenting a guideline for future improvement of AutoSDexplanations.

The remainder of the paper is organized as follows. We introduce the technical background to our work in Section 2, and our technique AutoSDin Section 3. The evaluation setup and research questions are provided in Section 4, and the empirical results based on these experiments are presented in Section 5. Threats and limitations are discussed in Sections 6, and 7 concludes.

2 Background

This section provides the motivation and background for our work.

2.1 Explainable Automated Debugging

Automated debugging has a long history, with research often being done on the topics of fault localization (Moon et al. 2014; Jones et al. 2002; Li et al. 2019) and automated program repair (Gazzola et al. 2019). As described before, while the technical complexity and performance of automated debugging techniques has been increasing (Jiang et al. 2023b), including the use of LLMs for APR (Jiang et al. 2023a; Xia et al. 2022), empirical work on explaining results for developer consumption has been difficult to identify. In addition to Monperrus' living review on APR having only one paper mentioning explanations (Monperrus 2020), Winter et al. (2022) find 17 human studies evaluating APR, of which none involved explanations directly from an APR tool; Kochhar et al. (2016) survey fault localization techniques at the time, and find two techniques that could provide explanations of their results (Sun and Khoo 2013; Mariani et al. 2011); unfortunately, both papers did not have human studies.

This contrasts to the growing body of literature showing that, to adopt automated debugging techniques in practice, 'explanations' for the results would be welcome. Developers have stated their desire for explanations in multiple occasions: along with the findings of Kochhar et al. (2016) mentioned earlier, a developer study on expectations for APR by Noller et al. (2022) notes that "the most commonly mentioned helpful output from an APR tool is an *explanation* ... including its *root cause*". Developer expectation is particularly important because when automated debugging has been adopted by industry, automatically generated patches are consistently reviewed by developers. At Meta, the APR system is connected to the internal code review platform (Marginean et al. 2019); at Bloomberg, Kirbas et al. (2021) write that "Bloomberg's view was that full automation was far from ideal", and they subject APR patches to be reviewed by a software engineer. This is also reflected in Noller et al.'s results that "full developer trust requires a manual patch review".

A promising way to present developers with explanations could be to show the reasoning trace (Lim et al. 2009) of a tool, i.e. how an automated debugging tool came to recommend a certain line for FL or a certain patch for APR. Unlike post-hoc explanation techniques such as commit message generation (Jiang et al. 2017), reasoning traces can answer critical questions that a developer may have, such as 'why this patch?'; indeed, research in Human-Computer Interactions (HCI) have indicated that explanations should strive to be capable of answering *why* an approach gave a certain result (Lim et al. 2009).

We suggest that ideally, such explanations and reasoning traces would provide two critical factors: (i) new information about the situation, so that developers can learn something real about the situation by reading the explanation, and (ii) a new perspective about the situation, so that developers can interpret the new information that is presented by the explanation. In fact, prior research on developer debugging practice supports the need of each of these components. As Böhme et al. (2017) note, developers spend a significant amount of time gathering information about the bug, such as internal program values. Meanwhile, it is also apparent from the literature that developers are not consumers of raw data; instead, they will formulate higher-level 'hypotheses' on why the bug is happening to interpret the data, as evidenced by a multitude of prior work (Siegmund et al. 2014; Alaboudi and LaToza 2020; Layman et al. 2013). As such, providing both information and perspective is important in generating an explanation seeks to be helpful for developers.

However, current automated debugging techniques are ill-suited to generate helpful explanations for their results under these criteria, as they fail to provide at least one of these factors. Using a common classification of APR techniques (Goues et al. 2019) as an example[1], generate-and-validate (G&V) techniques (Gazzola et al. 2019) (which includes learning-based techniques (Jiang et al. 2018; Xia and Zhang 2022; Zhu et al. 2021)) will generate variants of the buggy code until a test passes. As their deduction process is simply enumerating changes and trying them one by one, the process runs without regard to any 'hypothesis'. Semantics-based APR techniques such as Mechtaev et al. (2016) use variable values as inputs to Satisfiability Modulo Theory (SMT) solvers to more effectively search within a patch space; thus they are not inherently identifying any 'hypothesis' either. This is not to say these techniques are ineffective at fixing bugs - numerous work on APR shows that existing APR techniques can fix a wide array of bugs. Rather, we argue that because their reasoning trace is so different from humans, it is difficult to make a satisfactory explanation of their results. On the other hand, one way to make satisfactory explanations would be to develop an automated debugging technique that deduces in a similar way to humans, to make the decision-making process transparent (Dam et al. 2018). While technically similar, the generation of explanations is what distinguishes our technique from ChatRepair (Xia and Zhang 2023): while we allow LLMs to interact with a debugger in a formatted way that imitates human debugging, in the hopes of actually generating explanations helpful to developers, ChatRepair provides test execution result feedback to the LLM, and is thus closer to an extension of the generate-and-validate concept (Martinez and Monperrus 2019) in APR.

2.2 Scientific Debugging

To align APR reasoning traces more closely to those of human developers, we must know how developers debug in practice. Previous work on developer debugging patterns provide glimpses into how debugging is actually done.

Early work on developer debugging found that there was a "gross descriptive model" that developers followed, in which developers formulated hypotheses, then verified whether the hypotheses are true (Gould 1975). A formal version of this process was named *Scientific Debugging* by Zeller (2009), who advocated for developers to maintain a debugging log consisting of an iteration of the following items:

- Hypothesis: a tentative description that explains the bug and is consistent with the known observations;
- Prediction: an expected outcome if the hypothesis is true;
- Experiment: a means of verifying the prediction;
- Observation: the result of an experiment;
- Conclusion: a judgement of the hypothesis, based on the observation.

Siegmund et al. (2014) found that even without formal training in debugging techniques, all developers surveyed would roughly follow the 'hypothesis formulation, then verification' process of scientific debugging, where the developer will formulate a hypothesis about what the bug is, then observe actual execution results via debuggers or logging to verify the hypothesis. Thus, Scientific Debugging can be seen as a formal way of describing the dominant developer thought process when fixing bugs, and thus we seek to emulate this process to make an explanation when generating APR results.

[1] The explainability of FL is discussed in the Appendix.

2.3 Large Language Models

In this paper, we seek to emulate the Scientific Debugging process via Large Language Models (LLMs). We believe LLMs are capable of emulating Scientific Debugging for the following reasons. First, they have shown increasingly strong performance on question-answering benchmarks that involve reasoning (Brown et al. 2020; OpenAI 2023), which also makes it possible that they would be capable of predicting whether a hypothesis is met, and which hypothesis to investigate next. While it would be difficult to manually gather a large amount of data that contains debugging traces in the Scientific Debugging format, LLMs have also been demonstrated to be capable of few-shot or zero-shot problem solving: that is, given a few examples or simply a description of the task to be solved in the form of a natural-language *prompt*, they are capable of doing the task (Brown et al. 2020). This capability improves with Reinforcement Learning with Human Feedback (RLHF) training (Ouyang et al. 2022), which the main LLM of our task (ChatGPT of OpenAI) was trained on. Finally, the interaction with code that Scientific Debugging asks for requires the use of external tools. When using 'Chain-of-Thought' (CoT) prompting (Wei et al. 2022), LLMs appear capable of using the results of external tools to improve their performance as well (Yao et al. 2022; Gao et al. 2022). As a result, we believe that LLMs are well-positioned to emulate the Scientific Debugging process, and thus generate reasoning traces complete with actual execution results that would provide an intelligible guide to developers as to how the patch occurred.

3 Automated Scientific Debugging

The overall process of our approach is presented in Fig. 1. To start, the prompt containing relevant information is generated (Fig. 1 A): this consists of a detailed explanation of what Scientific Debugging is, and a description of the debugging problem itself, so that AutoSDcan proceed with the following steps. With the initial prompt prepared, AutoSDgenerates a hypothesis on what is wrong with the code or how it can be fixed, along with the concrete experiment that would validate such a hypothesis, using an LLM (Fig. 1 B). The experiment script will be passed to a background debugger/code executor process, which runs the script and returns the actual result (Fig. 1 C). Based on the observed information, AutoSDdecides whether the hypothesis was verified or not using an LLM (Fig. 1 D); depending on the conclusion, AutoSDeither starts with a new hypothesis or opts to terminate the debugging process and generate a fix. When the interaction with the code is over, AutoSDgenerates a bug fix based on the gathered information (Fig. 1 E). Unlike other automated program repair techniques we are aware of, as a result of steps (B - D) AutoSDcan provide a *rationale* of how a particular fix was generated, which can then be provided to the developer upon request.

3.1 Constructing the Input Prompt

To construct the initial prompt, as in the example presented in Fig. 1 A, we first manually wrote a detailed description of Scientific Debugging that explains what hypotheses, predictions, experiments, observations, and conclusions are, along with multiple examples for each category, so that the LLM can generate an intelligible reasoning trace. The full description can be found in the Appendix; here, we describe the aspects of the description critical for the pipeline of AutoSDin detail. For one, concrete examples of experiments are provided, to allow the LLM to predict appropriate experiment scripts: composite debugger commands

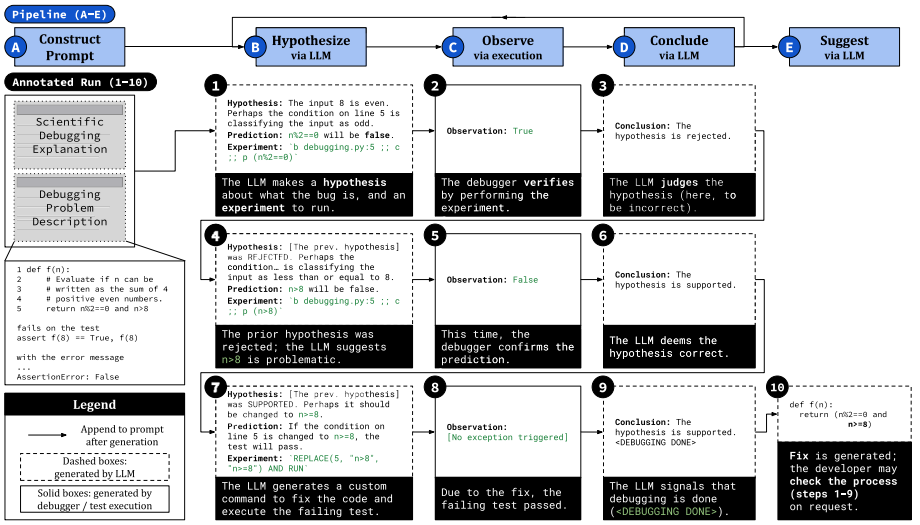


Fig. 1 The pipeline and a real example run of AutoSD, with annotations in black boxes and lightly edited for clarity. Given a detailed description of the scientific debugging concept and a description of the bug (A), AutoSDwill generate a hypothesis about what the bug is and construct an experiment to verify, using an LLM (B), actually run the experiment using a debugger or code execution (C), and decide whether the hypothesis is correct based on the experiment result using an LLM (D). The hypothesize-observe-conclude loop is repeated until the LLM concludes the debugging or an iteration limit is reached; finally, a fix is generated (E), with an *explanation* (white boxes from (1) to (9)) that the developer may view. The experiments generated in (1) and (4) are valid Python debugger commands, with `b` signifying the setting of a breakpoint, `c` signifying running the code until a breakpoint, and `p` being a command to print the value of an expression. The experiment in (7) is a valid code-editing DSL command, as described in the main text

(consisting of setting a breakpoint, running code, and printing a value) and a Domain-Specific Language (DSL) that we define to allow edit-and-execute commands are given. The Backus-Naur form definition of the DSL is provided in Fig. 2. The prompt explains the DSL, specifically that the following commands are available: `REPLACE(line, old_expr, new_expr)` that changes an expression at `line`, `ADD(line, new_expr)` that adds a new statement above `line`, and `DEL(line, old_expr)` that allows deletion of any expression within a line, including the entire line. Multiple commands can be joined with the `AND` connector, and finally the bug-revealing test can be executed after modification via the `RUN` command. In addition to experiment commands, the prompt instructs to predict the `<DEBUGGING DONE>` token (`<DONE>` for short in the rest of the paper) if enough information to discern the patch has been gathered, so that we can gauge how confident AutoSDis in its patch. The prompt is detailed enough so that our default LLM, ChatGPT, can follow the instructions zero-shot, i.e., without a concrete demonstration of the full process. On

$$
\begin{aligned}
<EXPR> ::= & \ \text{RUN} \\
& \mid \text{ADD(line, new\_expr)} \\
& \mid \text{REPLACE(line, old\_expr, new\_expr)} \\
& \mid \text{DEL(line, old\_expr)} \\
& \mid <EXPR> \ \text{AND} \ <EXPR>
\end{aligned}
$$

Fig. 2 Definition of the code-editing DSL used in our experiments

this description of scientific debugging, we add the bug-specific information: concretely, the buggy function/method, the test that reveals the bug, the error message when the bug is executed, and if available a bug report. We add this information as we believe such information would be necessary, if not sufficient, for a human to debug an issue, and thus would likely also help an automated technique to predict appropriate hypotheses and ultimately succeed in debugging.

3.2 Hypothesize-Observe-Conclude

With the initial prompt, AutoSDstarts iterating over the 'hypothesize-observe-conclude' loop depicted in Fig. 1 (Ⓑ - Ⓓ). The result of each process is appended to the prompt to allow incremental hypothesis prediction; i.e. when generating the conclusion in ❸, the LLM would predict it based on the concatenation of the initial prompt, ❶, and ❷. We describe each iteration of the loop as a *step*: for example, Figure 1 ❶ - ❸ would make up one step. Unlike previous automated debugging techniques that will make statistical 'guesses' on what patches are likely through search space reduction or neural networks, by going through the hypothesize-observe-conclude loop, AutoSDcan demonstrate to developers what observations led to specifically this patch. In turn, this allows developers to scrutinize the *generation process*, which can ease developer verification efforts and lead to greater developer trust in the results.

Hypothesize. Here, we lead the language model to generate a hypothesis by appending the token `Hypothesis:` to the prompt, so that the language model generates a hypothesis about the bug. We observe that the `Prediction:` and `Experiment:` line headers are also generated in turn by the LLM, due to the detailed description of the scientific debugging process provided by the prompt. The important aspect for the next step is the `Experiment` command, where the language model either generates a debugger command that can be executed by a debugger, or a custom code modification-and-execution script so that the language model can 'test' a certain change. As the document is in Markdown format, the `Experiment` script is wrapped in backticks (`); this script is extracted from the LLM output to get concrete code execution results in the next step. Examples can be seen in Fig. 1 ❶, ❹, and ❼ - note that AutoSDalso localizes the fault as a part of the hypothesizing process, thus making fault localization explainable as well.

Observe. The generated experiment script is passed to a background process based on traditional software engineering tools that provides real execution results back to the language model, so that we can ground the generation process of AutoSDon real results, and also build credibility for developer presentation. The model can either (i) invoke a composite debugger command by setting a breakpoint and printing a value, or (ii) modify the code and run the failing test with the aforementioned DSL. When executing a debugger command, it is executed via the command-line interface of the language-appropriate debugger, and the output from the last subcommand of the composite command (assumed to be a `print` command) is returned, as in Fig. 1 ❷ and ❺. When the breakpoint is within a loop, the debugger collects values at different timesteps of execution and returns them together, e.g. 'At each loop execution, the expression was: [v1, v2, ...]', up to a maximum of 100 values. Meanwhile, upon test execution from a edit-and-execute DSL command, if an exception is raised, the exception type and message are returned as the observation; otherwise, the result '[No exception triggered]' is appended, as in Fig. 1 ❽. As described in earlier sections and in our results, this step anchors the explanations in *actual execution results*, which existing patch explanation techniques such as commit message generation (or the baseline of asking an LLM for a patch explanation) are categorically incapable of. As we demonstrate in Section 5.5, this

incorporation improved developer trust in the explanations (80% of developers responded positively that the incorporation of execution results enhanced their trust in the explanation) while preventing the LLM from *hallucinating* explanations that may mislead developers.

Conclude. Based on the observation, AutoSDinvokes the LLM to check whether the hypothesis and the observation are consistent, by having the LLM predict if the hypothesis is rejected (e.g. ❸), supported (e.g. ❻), or undecided due to an unexpected observation. We have the LLM generate the conclusion to maximize flexibility in value interpretation, as the LLM will generate complex hypotheses or predictions at times that are difficult to automatically resolve based on debugger or test execution output. As described earlier, the LLM may predict a separate `<DONE>` token at this step if it predicts the debugging process is complete; in such cases, AutoSDwould have greater confidence in its output. An example is shown in Fig. 1 ❾: on the information that the previously failing test now passes, the LLM concludes that debugging is done. If the `<DONE>` token is predicted, AutoSDproceeds to generate a fix as in Section 3.3; otherwise the loop restarts with hypothesizing based on the newly available information until a maximum iteration limit s is reached. If `<DONE>` is not predicted until then, AutoSDis failing to identify the cause of the bug, and we may be more skeptical of the generated patch.

3.3 Fix Suggestion

When AutoSDhas completed its interaction with the code, either by predicting `<DONE>` or by reaching the maximum iteration limit s, the conclusions to each of the hypotheses are assessed, and rejected hypotheses are automatically removed from the prompt prior to patch generation, as this empirically improved program repair performance in our experiments. Even if rejected hypotheses are not involved when making the fix itself, rejected hypotheses can still be presented to the developer as context for successful hypotheses. We subsequently prompt the LLM to generate a fix using the available information by appending the words "`The repaired code (full method, without comments) is:\n````". This prompt leads the LLM to generate repaired code, based on the information available from the problem description and the code interaction, as in Fig. 1 ❿. We ask the LLM to generate code without comments to ease the parsing of the generated results and to help it focus on generating the fix itself. Identically to other APR techniques, a patch is ultimately generated; what makes AutoSDunique is that it can show its *intermediate reasoning steps* (❶ - ❾) as an *explanation* that can help the developer understand where a patch comes from.

4 Evaluation Setup

Here we describe the setup for our empirical evaluation.

4.1 Research Questions

RQ1: How well does AutoSDperform repair? While the main focus of our work is to generate a reasoning chain for automated debugging results, good performance in the debugging task itself is also important (Kochhar et al. 2016; Noller et al. 2022). We thus seek to answer whether AutoSDachieves performance competitive to prior APR techniques, and when compared to prompting the same underlying LLM as AutoSDto immediately predict

a fix (this baseline is referred to as LLM- BASE in the rest of the paper). To clarify, LLM-BASE has the same initial input as AUTOSD, but predicts fixes without interacting with the code. We aim to demonstrate that the explainability of AUTOSDdoes not come with a significant performance cost, even as prior reviews on explainable AI describe a tradeoff between interpretability and performance (Arrieta et al. 2020). We evaluate AUTOSDon the Almost-Right HumanEval benchmark we construct to mitigate data leakage concerns, and the Defects4J v1.2 and 2.0 benchmarks (Just et al. 2014) consisting of real-world bugs.

RQ2: How does the debugger influence the behavior of AUTOSD? Given that hypothesis verification is a critical aspect of AUTOSD, we evaluate whether the performance of AUTOSDis better when it indicates that debugging is done via the <DONE> token, which indicates the external observations match the generated hypotheses enough for the LLM to be confident that debugging is over. If AUTOSDcan indicate when it is likely to be correct, this could help developers make decisions about how to think about the automatically generated fix, as developer inspection time spent on reviewing patches could be reduced by omitting patches that AUTOSDis not confident in. As such a property would likely aid developer adoption of AUTOSD, we evaluate to what extent <DONE> predicts better performance. Based on our these experiments, we further evaluate the performance of AUTOSDwhen debuggers are not used, and observations are 'hallucinated' by the LLM instead of obtained via actual code execution. We evaluate whether under this setting, the <DONE> token continues to be a marker of strong performance.

RQ3: How does the choice of LLM influence the performance of AUTOSD? We evaluate the performance of AUTOSDas we vary the LLM that is used. While we empirically found the best performance when using the ChatGPT model, and thus used it as the default setting throughout the rest of the paper, by varying the size of the language model and plotting the performance, we investigate automated repair performance as models improve in terms of parameter size and training sophistication.

RQ4: How do developers benefit from AUTOSDexplanations? Via our human study, we evaluate whether developers benefit materially from automatically generated explanations by AUTOSD, i.e. regardless of their opinion towards explanations. In our human study, participants are given the buggy code, a bug-revealing test, a candidate patch, and half of the time an explanation, and asked to determine whether the patch correctly addresses the issue that the test reveals. We measure the time and accuracy of developers when deciding whether a patch is correct, along with developer answers to the question 'did the explanation help you make the decision?'. We thus hope to evaluate whether developers benefit by being provided explanations.

RQ5: How do developers feel towards AUTOSDexplanations? We evaluate whether the explanations of AUTOSDare acceptable to developers by asking them six questions on whether they would want to use APR, whether they would want explanations when using APR, and whether AUTOSDand each element of its explanation were satisfactory. Unlike RQ4, which evaluates the material benefit developers derive from each explanation, this RQ focuses on developer opinion. We thus hope to measure whether developers are willing to use explanations, distinctly from whether their productivity increases from explanations. We additionally perform interviews to identify what developers liked about the explanations of AUTOSD, and what could improve.

RQ6: What do AUTOSDexplanations look like? We provide examples of liked and disliked patch attempts and their corresponding explanations in this research question as further context, along with a breakdown of common failure causes by analyzing a random sample of 25 cases in which all hypotheses generated by AUTOSDwere classified as incorrect by itself.

4.2 Environment

4.2.1 Evaluating Explainable APR Performance

To empirically evaluate AutoSD, we use four program repair benchmarks. First, the widely-used Defects4J benchmarks (Just et al. 2014) version 1.2 and 2.0, which have been used by prior work as a standard benchmark to compare APR techniques (Liu et al. 2020), are used to evaluate program repair performance, namely how many bugs are fixed by the technique. Meanwhile, we use the BugsInPy benchmark (Widyasari et al. 2020) (abbreviated to BIP in our paper) for the sake of getting real-world Python bugs to evaluate in our human study, but we do not report the program repair performance of AutoSDon BIP as many of its bugs needed additional environment setup not described in the README which makes setting it up so that tests and debuggers execute correctly is difficult, whereas execution is critical to the operation of AutoSD.

We additionally construct the Almost-Right HumanEval (ARHE) dataset based on the HumanEval Python single-function synthesis benchmark by Chen et al. (2021), and use it for both our human study and performance evaluation. We construct it in the hope that it will be free from data contamination concerns, as HumanEval was explicitly made by Chen et al. to avoid data contamination when evaluating their LLM, and was also used to evaluate the recent GPT-4 model (OpenAI 2023); constructing such a dataset is particularly important given that Lee et al. (2024) find that many bugs from Defects4J are included in the training data of the open-source LLM StarCoder. The ARHE dataset was built by mutating the human solutions in the HumanEval benchmark so that exactly one test fails, making bugs that cause the code to be 'almost' right. To do this, we first generate all possible mutants using each mutation operator, then select the mutants that result in exactly one test failure, filtering out mutants that cause more tests to fail. We do this to construct a *repair* dataset - whereas fixing a mutant that causes many tests to fail is perhaps akin to fixing the entire functionality of the code, fixing a mutant that causes one test to fail is closer to what we would consider to be repair as one must preserve the existing correct functionality while rectifying the erroneous behavior as indicated by the failing test. Through this procedure, we end up with 200 bugs to evaluate with using seven mutators, as is presented in Table 1, and compare them to mutators in PIT (Coles et al. 2016), a widely used mutation testing tool. 'Integer Literal Changer' will change literal `0` constants to `1` constants, and vice versa, which shows similar behavior to the 'Inline Constant Mutator' of PIT. 'If Remover' will remove the then-block or else-block of an if statement; if it has no remaining children, the if statement itself will be removed, similarly to 'Remove Conditionals Mutator' of PIT. 'String Literal Changer' will make a string literal

Table 1 ARHE benchmark breakdown

Mutator	Number
Integer Literal Changer ○	45
If Remover □	24
String Literal Changer Δ	63
Operator Changer ○	40
Binary Operator Remover □	24
Augmented Assignment Changer ○	3
If Negator ○	1

Reversible mutators are marked with ○, irreversible mutators are marked with □, and occasionally reversible mutators are marked with Δ

empty, lower-case, or upper-case; making the string literal an empty string is not reversible, but whether the lower-casing or upper-casing can be applied in the reverse to get the original code differs from problem to problem. The generation of empty strings is similar to the 'Empty returns Mutator' of PIT. 'Operator Changer' will change pluses to minuses, along with similar operations, similarly to the 'Math Mutator' of PIT. 'Binary Operator Remover' will remove a binary operator and only leave one of the operands, similarly to the 'Arithmetic Operator Deletion Mutator' of PIT. 'Augmented Assignment Changer' will change `+=` to `-=`, vice versa, etc., similarly to the 'Increments Mutator' of PIT. 'If negator' will add a `not` to an `if` condition, similarly to the 'Negate Conditionals Mutator' of PIT. Mutators were added iteratively until the ARHE dataset contained 200 bugs, which we deemed to be a reasonable number for APR evaluation.

When using this dataset, we additionally compare against a template-based APR baseline that has the reverse mutators of those used to construct the dataset, and randomly applies them to the buggy code. This baseline is used for this benchmark as we seek to demonstrate that it is not trivial to repair the bugs in this benchmark just because the mutators that were used to cause the bugs were simple. We run this baseline 100 times as it is stochastic. Note that 90 bugs of ARHE are created by deletion or string mutation, and consequently are not reversible by the baseline: all the remaining mutations are reversible and therefore can be fixed by our template-based baseline given sufficient time. In Table 1, the 24 bugs from If Remover and 24 bugs from Binary Operator Remover are not reversible; furthermore, we manually determine that 42 of the 63 String Literal Changer bugs are not reversible, making for a total of 90 bugs that cannot be repaired by applying the same mutation set.

Regarding specific APR parameters, for each dataset we provide AutoSDwith the buggy method and generate 10 patches, to match the settings in the large-scale empirical work by Jiang et al. (2023a), who evaluate the repair performance of multiple large language models and more traditional learning-based APR techniques. Their evaluation setup of generating 10 patches was motivated by Noller et al. (2022), who note that developers are willing to review up to ten patches, and thus provides a practical basis for comparing APR techniques acceptable to developers. We note our setting assumes less exact information and is thus more realistic: Jiang et al. evaluate with perfect statement-level FL, whereas AutoSDuses perfect method-level FL and the bug report, and thus needs to also identify which statement is faulty within the method based on available information. When evaluating the generated patches, we run the tests provided by each dataset for each bug; a fix that makes all tests pass is deemed a *plausible* patch, and plausible patches are manually inspected to see if they are semantically equivalent with the developer patch. Semantically equivalent fixes are deemed *correct*; semantic equivalence is determined by going through the generated patches and finding counterexamples where the behavior of the developer patch and generated patch would diverge. This is important to ensure that the LLM is not simply making all tests pass and thus being potentially misleading, but actually correctly fixing the bug. If at least one of the 10 generated patches are correct or plausible, the bug is deemed correctly fixed or plausibly fixed, respectively.

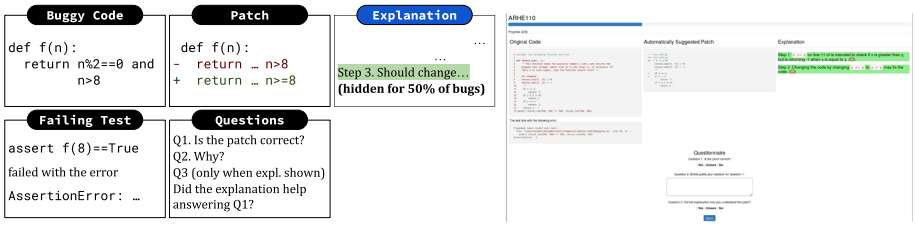
AutoSDrequires the use of an LLM and a debugger. For the LLMs, we experiment with the CodeGen (Nijkamp et al. 2022), Codex (Chen et al. 2021) (code-davinci-002), and ChatGPT (a sibling model to InstructGPT (Ouyang et al. 2022)) LLMs, with the ChatGPT LLM being the default model. LLM output is sampled using a temperature of 0.7. Different debuggers are used depending on the target language; we use the `jdb` tool for the Java benchmarks (Defects4J v1.2 and v2.0) and the `pdb` tool for the Python benchmarks (ARHE and BugsInPy). The maximum iteration limit, s, is set to 3.

4.2.2 Human Study Parameters

To approximate the real-world impact of AutoSD, we perform a human study by asking participants to review patches, based on the real-world applications of APR (Marginean et al. 2019; Kirbas et al. 2021). We specifically sampled 12 bugs where AutoSDmade a patch that caused the initially failing test to pass: a random sample of six such bugs from the ARHE dataset (which had complete documentation), and six real-world bugs from the BugsInPy Python dataset (Widyasari et al. 2020). In our preliminary studies, we found that reviewing 12 patches could take a long time, so we divided the 12 bugs into two groups of six (each containing three ARHE and three BugsInPy bugs) and randomly assigned participants to solve code review problems from one of the groups, so that each participant would see six bugs. A scheme of the code review screen that was presented to participants is shown in Fig. 3 (a); a screenshot of the the survey website which corresponds to our schematic is shown in Fig. 3 (b). Our human study received IRB review exemption (IRB-23-054); our study was conducted on the basis of Ko et al. (2015), who recommend randomization and institutional oversight on study design.

For each code review problem, participants are provided with the buggy code, the bug-revealing (failing) test, along with the patch; they are provided with the explanation in a randomly selected three of the six cases. Each step of the explanation has a header, which is a summary of the hypothesis explaining the bug; the header is color-coded based on the predicted conclusion, with supported/rejected/undecided hypotheses being green/red/yellow, respectively, as in Fig. 3. Each header can be clicked to reveal the full reasoning process of AutoSDas depicted in Fig. 1. Participants are asked three questions for each patch: (Q1) whether the patch is a correct patch, where they may answer yes, no, or unsure (as a proxy for checking correctness during the code review process (Sadowski et al. 2018)); (Q2) a short justification of their decision in Q1, to filter potential bad-faith answers; and (Q3) when an explanation is available, whether the explanation was helpful in making their decision, to measure the differing impact of explanations for different patches. Based on the developer patch for each bug, the authors determined which patches were accurate; developer assessment accuracy was calculated based on whether developers under a certain condition (i.e., with or without explanations) had the same answer to Q1 when compared to the developer-patch determined patch correctness.

To recruit participants, we advertised the task to both undergraduate and graduate students with at least 1 year of Python experience, as well as professional developers at a company that specializes in software testing techniques. Overall, we recruit 20 participants: eight undergraduate and six graduate students, as well as six professional developers whose career span from 3 to 10 years. As a result of dividing the participants into two groups, each debugging problem was inspected by 10 people. Participants start with a briefing of what



(a) Schematic (b) Screenshot

Fig. 3 Human Study Screen Scheme and Screenshot. For a larger version of the screenshot, see our Appendix

Table 2 Repair results on the ARHE benchmark

Result	Template-based	LLM- BASE	AUTOSD
Plausible	85.77 ± 4.20	179	189
Correct	-	177	187

The template-based performance is based on 100 reruns, and shows the mean and standard deviation repair performance

they should do in the study, solve an example code review problem as practice, and then solve six code review problems in 30-40 minutes in a randomized order. The six code review tasks contain 2 correct and 1 incorrect patches for ARHE and BugsInPy benchmarks, respectively. After conducting a post-questionnaire about their demographics and overall satisfaction with explanations, we perform an interview that lasted about 5 minutes on their impression of the tool for qualitative analysis.

5 Experimental Results

We present the results of empirical evaluation below.

5.1 RQ1: How Well Does AUTOSDPerform Repair?

In Table 2, we present the APR performance of AUTOSDon the ARHE benchmark when compared with LLM- BASE and the template-based baseline. Note that the template-based baseline shows significantly weaker repair performance than both LLM- BASE and AUTOSDwhen evaluated under the same conditions; as a result, we did not assess correctness for the thousands of patches generated, as the upper bound of correctness is the plausible patch count. Additionally, the performance of LLM- BASE and AUTOSDare similar, demonstrating AUTOSDretains the repair performance of the LLM while simultaneously being capable of generating explanations.

In Table 3, we present the APR performance of AUTOSDon the Defects4J benchmarks when compared against LLM- BASE and the best-performing techniques from the empirical study by Jiang et al. (2023a): Recoder, a DL-based APR technique (Zhu et al. 2021) which a custom neural architecture for repair that utilizes ASTs, and finetuned InCoder (Fried et al. 2022), a language model from Facebook, which was finetuned to predict the fixed line given the exact buggy line, i.e. perfect statement-level FL results, and thus uses more exact information than AUTOSD. We find that AUTOSDagain shows competitive performance when compared to other baselines, even those that have more specific information provided. As an additional reference point, when compared against the repair results of Codex on Defects4J presented by Xia et al. (2022) and ChatRepair (Xia and Zhang 2023), which generate 200 patch candidates (unlike our 10) on both benchmarks under the 'patch function' setting, we find that AUTOSDoutperforms or matches their performance with substantially

Table 3 Correct repair results on the Defects4J benchmarks

Benchmark	Recoder	InCoder	Codex*	ChatRepair*	LLM- BASE	AUTOSD
D4J v1.2	24	41	63	76	87	76
D4J v2.0	11	28	45	48	110	113

Results for Recoder and InCoder are from Jiang et al. (2023a). , while results from Codex and ChatRepair are from Xia et al. (2022) and Xia and Zhang (2023), and use 200 patch generations instead of 10

less patch generation, while assuming the same FL conditions as our setup. We additionally analyzed why AutoSDunderperformed LLM- Base on the Defects4J v1.2 dataset. We find that AutoSDshowed roughly the same performance as the baseline for every project in the benchmark except Closure, which is a JavaScript compiler and hence deals with complex objects with nested reference structures. Consequently, AutoSDseemed to have difficulty making proper judgments on whether the hypothesis was actually met on the basis of these complex values, and these incorrect decisions ultimately led to inaccurate patch generation. It is noteworthy that such complex values would also require significant efforts for the developer to understand as well. Thus, we argue that there is a need to research how to present values for both LLM and developer consumption.

Answer to RQ1: AutoSDis capable of operating at a competitive level of program repair performance when compared to a diverse set of baselines on three repair benchmarks.

5.2 RQ2: How Does the Debugger Influence the Behavior of AutoSD?

This RQ first investigates whether the confidence in a result indicated by the prediction of the <DONE> token actually correlates with better performance. The results are presented in Fig. 4. For Defects4J, as it was infeasible to manually label all 1045 plausible patches generated for the dataset, we sampled 100 patches with and without <DONE> to get results. As the figure shows, for both the ARHE and Defects4J datasets, AutoSDshows a higher precision when the <DONE> token is generated as part of a conclusion, indicating that AutoSDcan indeed signal when it is likely to generate a plausible or correct patch. Furthermore, for bugs where a plausible patch was generated and the <DONE> token was predicted, 89% were correctly fixed, while for bugs with plausible patches but without <DONE> predictions 82% were correctly fixed. These results indicate that AutoSDcan indicate when its output is likely to help developers based on its interaction, and thus aid developer decision-making and potentially reduce developer inspection cost when processing automated debugging results. It is noteworthy that this is a natural property of the patch generation process of AutoSDitself, and did not require a separate patch correctness detector (Xiong et al. 2018) being added specifically for this purpose.

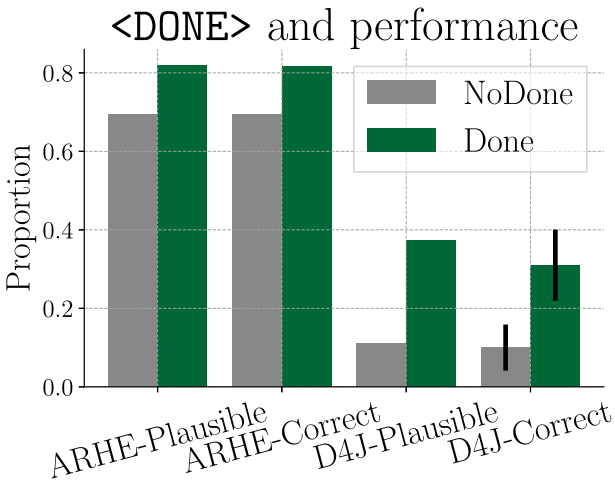


Fig. 4 <DONE> & perf

We also investigate the performance when the debugger/code execution results are also predicted by the LLM, instead of being obtained via concrete execution, for the ARHE dataset; would the <DONE> token still predict good performance? In this 'debugger hallucination' scenario, <DONE>-predicted solutions were actually 11%p *less* likely to be plausible; this is in contrast to using actual code execution results, where <DONE>-predicted solutions are 12.4%p more likely to be plausible. Furthermore, individual runs became much less likely to be plausible: while 73% of the individual AUTOSDruns would yield a plausible patch, only 63% would when the debugger was ablated. Thus, incorporating code execution contributes to the reliability of AUTOSD; we later demonstrate in RQ5 that developers found real code execution results useful as well.

Answer to RQ2: AUTOSDcan indicate when its answers are more likely to be correct with the <DONE> token, which we also use to verify the utility of debugger use.

5.3 RQ3: How Does the Choice of LLM Influence the Performance of AUTOSD?

In Fig. 5, we depict the performance of AUTOSDas different underlying LLMs are used, with the x axis showing different LLMs roughly sorted in terms of number of parameters and the technical advancement of training, and the y axis showing the performance of AUTOSDwhen using the LLM on the ARHE benchmark. The performance of AUTOSDis depicted along with the performance of simply querying the LLM to fix the bug. As shown, the performance of AUTOSDrapidly improves and ultimately becomes comparable to the performance of LLM-BASE, suggesting that AUTOSDshows better performance when using stronger language models; for smaller models such as CodeGen-6B, repair itself fails in a zero-shot setting, as in our experiments it would simply return the original buggy code. (We confirm that the model implementation works by also evaluating in a few-shot setting for CodeGen-6B; it could fix 44 bugs in that case.) Indeed, it appears that CodeGen-6B is incapable of running AUTOSD, as it failed to match the provided format for Scientific Debugging in 68% of all cases; meanwhile, that was only the case in 0.7% of runs when using ChatGPT. Thus, we may speculate that as language models improve, the performance of AUTOSDwill also become stronger.

Answer to RQ3: Under our experimental setup, as the underlying language model improves, the performance of AUTOSDalso increases.

5.4 RQ4: How do Developers Benefit from AUTOSDExplanations?

In this section, we evaluate whether developers benefit from explanations in a way that is unlikely to be swayed by a participant's opinion about explanations. The results of measuring the code review time, accuracy, and whether the explanation was rated as helpful in making the decision are presented in Fig. 6.

First, looking at the amount of time that it took to solve the code review problems, we find that the time it took to solve a problem was generally similar between the case where there was no explanation and when there was an explanation. There is no case where the difference is statistically significant, despite the explanations of AUTOSDproviding more information than the case without explanations, and thus potentially requiring more processing time from developers.

Regarding the accuracy with and without explanations, participants were more accurate when solving the same problems with explanations than without explanations in seven cases,

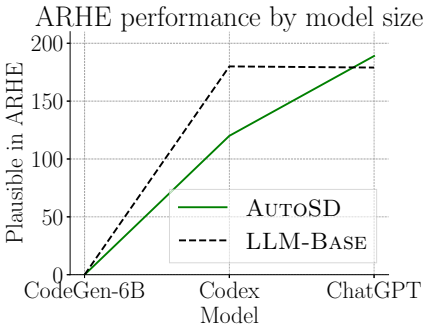


Fig. 5 Model Size

with five of them being concentrated in the real-world BugsInPy benchmark. Using the Mann-Whitney U Test as suggested by Arcuri and Briand (Arcuri and Briand 2011), we found explanations to have a statistically significant effect on developer accuracy assessment ($p < 0.05$). These results demonstrate that AutoSDcould have a positive impact on real-world developer productivity when using APR, as the judgment quality improved when evaluating real-world bugs while requiring roughly the same amount of developer time. Meanwhile, there are two cases where the use of explanations lead to a drop in accuracy: ARHE105 and BIP003. For BIP003, we found that the respondents became more cautious after looking at the explanation, and answered that they needed more information to judge it. Meanwhile, for ARHE105 the participants who answered incorrectly accepted the reasoning of AutoSDwithout significant scrutiny. While this was a somewhat rare incidence that happened in one of the 12 randomly sampled problems, it highlights the need of further research to identify potentially misleading reasoning. Additionally, developer accuracy improved with explanations on the two incorrect patches from BIP (BIP002 and BIP004) meaning developers are not blindly accepting patches with explanations.

On whether the participants found the explanations helpful in their decision-making, in eight of the twelve questions developers noted that the explanations were actually helpful when coming to their conclusion, underscoring the psychological benefit that providing explanations for patches holds.

Answer to RQ4: When exposed to explanations generated by AutoSD, human participants could process patches in roughly the same time, while achieving a higher accuracy in five of the six of the real-world bugs. They also rate the explanations as helpful in two-thirds of all bugs.

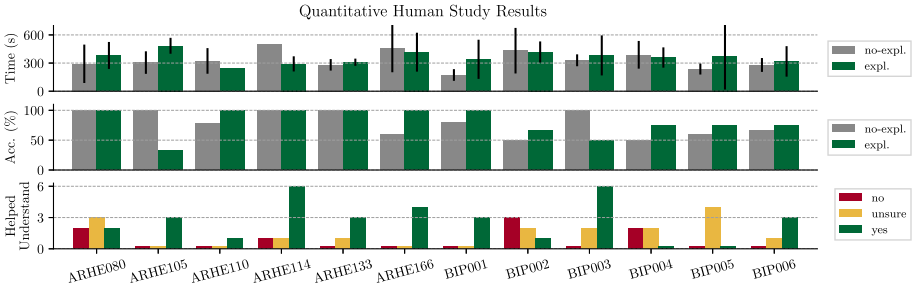


Fig. 6 Developer performance on code review tasks with and without explanations from AutoSD, and explanation ratings

5.5 RQ5: How do Developers Feel Towards AutoSDExplanations?

The results of our post-questionnaire are presented in Fig. 7. To our surprise, there was a discrepancy in satisfaction of AutoSDbetween students and professional developers: while more than half of the students were satisfied with AutoSD, only one of the six developers were satisfied. We use these differing results as an opportunity to discuss the strengths and potential improvements of AutoSD-generated explanations.

What did students find appealing about the explanations of AutoSD? Ten out of the 14 student participants noted that they 'missed' the explanations when they were not available. When asked why they wanted to see the explanations in these cases, and how they used explanations when they were available, students described a wide range of thought processes that were aided by the existence of explanations. One common pattern was to think through the patch by oneself, then comparing one's internal thoughts to the provided explanation; one participant referred to the explanation as useful because it could function as a 'rubber duck'.[2] Another common usage of explanations was to look at the explanation to discern where to focus effort on, and thus guide the direction of judgment. Other students would use the explanation to gain a better understanding of what the code was intended to do. We thus argue that a strength of AutoSD-generated explanations is that **they can accommodate a diverse set of thought processes**, potentially aiding a wide range of developers.

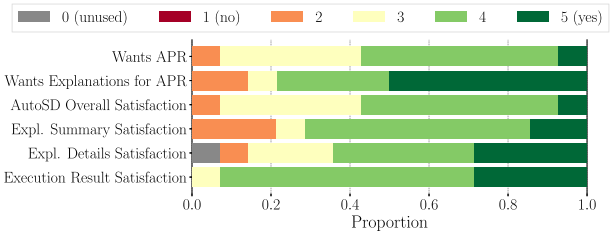
Meanwhile, another usage pattern was to look at the experiments and observations within the explanations to get a concrete idea of what the values are at certain points, and use those values to build a mental model of how the bug happened. This points to another strength of AutoSD, which is that it **incorporates actual values in its explanations**: in Fig. 7 (a), we note that more than 90% of students thought that the addition of execution results improved their trust in the explanations. Critically, these results justify AutoSD's architectural choice of interacting with code: while the interaction may cause program repair to take a greater amount of time, it provides reliable elements in the explanation that improve developer trust of the explanations.

On the other hand, professional developers showed a more mixed attitude towards the explanations of AutoSD. It is noteworthy that developers are not opposed to explanations themselves: half agreed or strongly agreed that explanations would be important when using an APR tool (Fig. 7 (b)), highlighting the importance of the problem. When asked why they found the explanations of AutoSDleft more to be desired, one suggestion was that the current explanations would be more useful if they were connected with "business logic" or specifications, a suggestion echoed by one of the student participants as well. The professional developers argued that without such connections, the explanations needed to be verified rigorously and even after that were of limited value. Thus one potential direction of improvement would be to **integrate explanations with existing development artifacts like specification documents**.

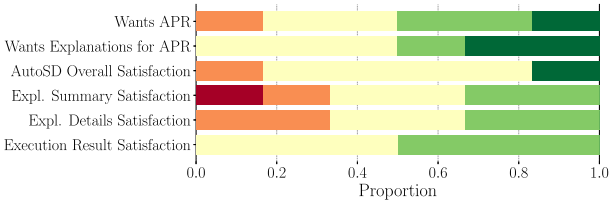
Another common suggestion was to improve the interface of the tool: developers noted that they might use the tool if it was attached to an IDE, and that the explanations were too wordy. This feedback suggests that to improve developer satisfaction, we may consider **integrating explanations to platforms that developers frequent** (as also suggested by Kochhar et al. (2016)), and further study the specifics of explanations that developers find satisfactory.

Looking at the overall statistics, we find that 70% of participants agreed that explanations were an important factor when using program repair, and 55% found the scientific debug-

[2] See https://en.wikipedia.org/wiki/Rubber_duck_debugging.



(a) Satisfaction among students.



(b) Satisfaction among professional developers.

Fig. 7 Human study post-questionnaire results by group

ging details (Expl. Details Satisfaction of Fig. 7) satisfactory, showing that a majority of participants agreed with the overall motivation and formulation of AutoSD.

Answer to RQ5: While the explanations of AutoSDare capable of accommodating diverse thought processes and improving developer trust by using concrete execution results, they could be further improved by enhancing the interface and by linking to specifications.

5.6 RQ6: What do AutoSDExplanations Look Like?

What do the explanations generated by AutoSDlook like? In addition to the example embedded in Fig. 1, we provide two reasoning traces generated by AutoSDthat were liked (BIP006 - 75% liked) and disliked (BIP002 - 16% liked) in the human study from the real-world BugsInPy problems. On the left of Fig. 8, we show a liked explanation, along with a condensed failing test and the generated fix. Looking at the patch, the developer will see that a `.lower()` call was added; without an explanation, this fix can appear spurious. In contrast, by providing a rationale on why AutoSDfocused on this area, participants could swiftly identify whether this fix was related to the test. For example, Student-6 said "I first looked at the explanation, which helped me identify which part of the code to look at". The subsequent experiment confirms that an uppercase `'Chunked'` header was within the program state, which is the source of the bug. These execution results helped participants understand the bugs, e.g. Student-11 who noted that "expression values were useful in making decisions". Overall, this patch was correct, and the explanation aided developer comprehension and built trust. While we provide a simple example from the human study, we also note that AutoSDworks on more complex bugs as demonstrated in Section 5.1, and provide additional examples in the Appendix.

Attempts at hypothesizing can fail as well. The right side of Fig. 8 depicts an case where AutoSDfails to validate any hypotheses. While AutoSDinitially generates a hypothesis about appending in the wrong order, the line that is suggested in the experiment is actually

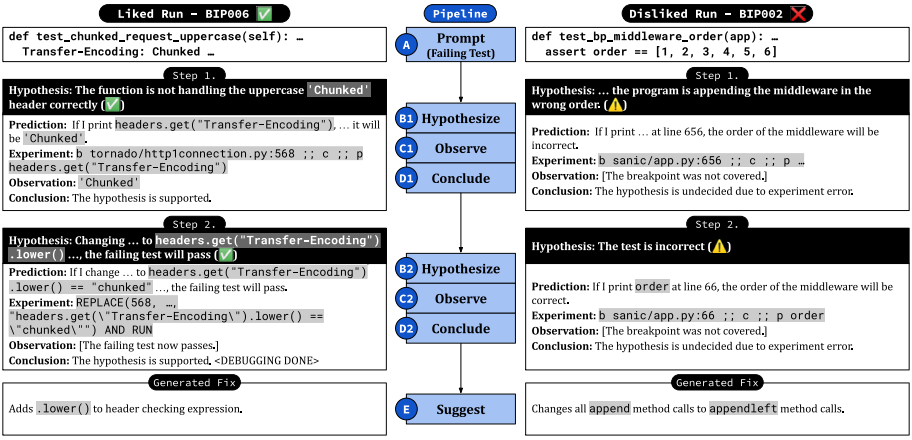


Fig. 8 Example successful and unsuccessful repairs and explanations of AutoSDfrom the human study

not covered; as a result, the debugger provides feedback that the breakpoint was not covered. This is one of the most common failure causes - our analysis on 25 cases where all hypotheses were rejected revealed that in 13 of the 25 cases, breakpoints suggested by AutoSDwere never hit, and consequently AutoSDcould not get results for generated experiments. In BIP002's case, instead of looking for new breakpoints that could be covered by the test, the LLM starts suggesting that the test is wrong. Ultimately, while a fix is generated, the explanation has little connection to the patch, and as a result the human study participants rated the explanation as unhelpful; the patch itself is plausible but incorrect as well. Nonetheless, the example also illustrates how bad explanations can still lead to better decision-making: developers may see that the foundations of the patch are weak, and be (rightly) more suspicious about the patch. In this context, it is noteworthy that developers who saw the explanation of BIP002 more accurately assessed it (Fig. 6). Other failure modes include generating an invalid experiment expression (2/25) or adding multiple print commands in the experiment script when the infrastructure of AutoSDonly allows one print command, causing inaccurate hypothesis rejection (2/25).

We additionally present examples from the more complex bugs of the Defects4J dataset in Fig. 9. In the left case, AutoSDhypothesizes that the bug is happening when the current token is `END_OBJECT`, and generates an experiment to confirm that this is the case. As this is actually the case, it proceeds to search for what behavior would lead to the failing test to pass in Attempt 2. Combining these two steps together, it generates a patch identical (in this method) to the developer patch, and that makes all tests in the test suite pass. Meanwhile, on the right, another example of failing to identify the right breakpoint is provided. In this case, the same hypothesis and experiments are parroted, leading to no improvement.

Answer to RQ6: AutoSDcan generate helpful explanations on its patches, but the reasoning process may fail as well. A common failure cause is an inability to identify the right breakpoints.

6 Discussion

This section provides threats and limitations of our work.

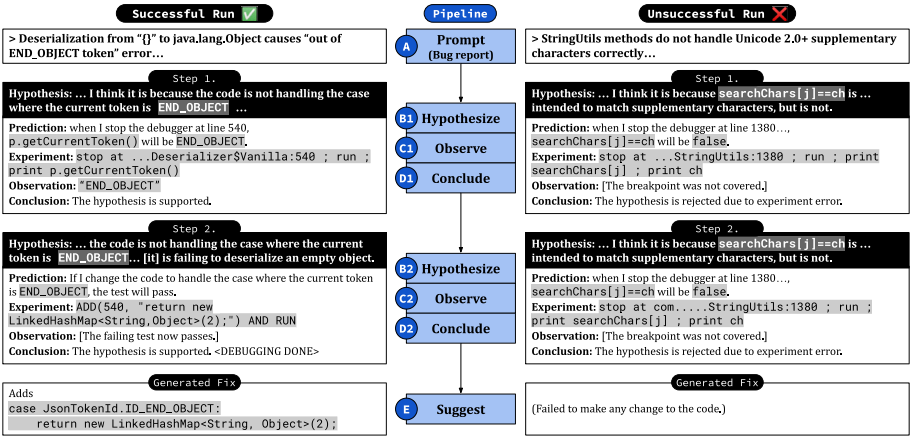


Fig. 9 Example AUTOSDruns from Defects4J bugs

6.1 Threats to Validity

Internal Validity concerns whether the analysis supports claims about cause and effect. Potential threats include incorrect implementations, inaccurate patch correctness assessment, and the risk of biased responses in our human study. To mitigate the impact of the first two concerns, we plan to make our implementation and repair results publicly available for scrutiny. For our human study, in addition to gathering developer sentiment about the generated explanations (which included occasional negative feedback), we also find that participant accuracy improved in five of the six BugsInPy problems, which is a result difficult to be due to bias. In addition to these issues, we note that AUTOSDcan at times generate misleading explanations; however, the explanations are still grounded in actual execution results, and thus more reliable than techniques that do not use such results at all, such as commit message generation techniques or LLM-generated explanations without such results.

External Validity concerns whether the results presented in this paper may generalize to other results. A particular concern when using large language models is that their training data may include segments of the evaluation data. To mitigate this issue, we newly constructed the ARHE dataset for repair and evaluated AUTOSDon that benchmark. Furthermore, our explanations were likely never within the training data, as developers usually describe code with less of a structure than Scientific Debugging prescribes, even if they think along the lines of it.

6.2 Limitations

AUTOSDhas a number of limitations that we would like to highlight. First, to enable multi-step interaction with code, both the language model and debugger must be invoked multiple times, which increases the repair time of the technique; in our experiments, AUTOSDtook on average 4.66 times longer to generate a patch when compared to LLM- BASE. Specifically, to generate ten patches for a single bug, AUTOSDwould on average take 9 minutes and 22 seconds, while LLM- BASE would take 1 minutes and 59 seconds. Nonetheless, given the significant developer demand for explanations of automatically generated patches as shown in Fig. 7, we believe that the additional cost needed to build explanations for patches is justified. We note that this execution time is primarily due to the interaction with external

tools, and is not not significantly impacted by the length of the prompts itself; when removing the bug report, the execution time was practically the same, while the performance dropped by 39%, indicating that providing additional information can help without significant execution cost. Second, as a step towards explainable automatic debugging, we evaluated in the setting where method-level FL was done, and AutoSDwould then perform statement-level FL in an explainable manner. Our main focus in this paper was to establish that AutoSDcan generate explanations that aid developers in practice; we hope to work on explainable method-level FL in future work. On a related note, our technique can only handle single-method bugs as of now; incorporating a wider range of information to handle more complex bugs is also an interesting research direction. In this work, we experimented on benchmarks for Python and Java debugging problems. Further experimentation is required to discern how well AutoSDwould perform with other languages and benchmarks. Nonetheless, because the prompt of AutoSDis presented in a zero-shot manner, one would only need to change the debugger examples when the language changes. Indeed, aside for the Java and Python-specific debugger command examples, our starting prompt did not change when using ARHE and Defects4J. Finally, the generated explanation may occasionally lend credibility to incorrect patches; by allowing our technique to indicate its confidence in its output and demonstrating that confidence is correlated with correctness, we take the first steps to address this issue. Furthermore, our explanation includes concrete code execution results, aiding developer decision-making (Fig. 7).

6.3 Future Work

The limitations presented in our work also provide a few directions that AutoSDcould improve to further its cause of helping developers in automatically debugging issues. As described in RQ5 (Section 5.5), professional developers pointed out problems with the current explanation format, such as that it was too long or lacked connection with specification artifacts. These results (i) provide valuable feedback on what types of explanations are appealing to developers, and (ii) suggest that AutoSDcould be improved by incorporating text summarization techniques (Allahyari et al. 2017) (while taking care to preserve the actual value aspects, which improved developer trust), and by improving the user interface via links to concrete artifacts. Thanks to the incorporation of chain-of-thought (Wei et al. 2022) prompting in APR as well, AutoSDcan also benefit from the large volume of recent work that aims to improve the performance of CoT prompting. Of particular interest is using techniques such as reflection (Shinn et al. 2023) or tree-of-thought (Yao et al. 2023), which could help improve the reasoning process of AutoSDand reduce failures such as those presented in RQ6 (Section 5.6) by allowing the LLM to 'correct' its unhelpful reasoning process. Finally, while in this work 10 patches were generated per bug, which is likely the limit of what developers are willing to manually inspect (Noller et al. 2022), as a future direction one could improve the performance of AutoSDby sampling more patches and ranking them using techniques such as self-consistency (Wang et al. 2023).

6.4 Implications

Our work has a few implications for future work. One is that explainable software engineering tasks, once considered difficult due to the significant difficulty of dealing with natural language, has been made significantly easier as LLMs are fluent in natural language. In our work, we also present criteria for explanations, namely that explanations should provide

both new information and a new perspective, which was highly rated by developers as well; these guidelines should help researchers develop novel explainable automated debugging techniques going forward. Another potential implication of this work is that we have demonstrated LLMs can follow development processes that were designed for humans in mind. This indicates the potential for research on how LLMs could fit naturally into human workflow, which was highlighted as important by prior work (Winter et al. 2022).

7 Conclusion

In this paper, we summarize the importance of explanations for automated debugging results as revealed by prior studies, and the lack of automated techniques capable of providing adequate explanations for humans. We argue this is due to a lack of automated debugging techniques that deduce in a human way, and bridge this gap between automatic and manual debugging practices by using LLMs to emulate the Scientific Debugging process. We demonstrate that AutoSDis capable of achieving competitive repair performance when compared to other repair baselines, while having favorable properties for practical use such as an indication of confidence in the output. The repair performance of AutoSDalso improves as language models become more capable, suggesting the performance and availability of explanations may improve as language models get better. Finally, our human study reveals that the automatically generated explanations could improve developer assessment of patches, with a majority of students also expressing that they 'missed' the explanations when they were not available. The interviews we performed show that the explanations AutoSDgenerates could aid a wide range of developer thought patterns, and that they could be improved via tighter integration into the development process, such as making connections to written specification. Overall, we believe that the rapid improvement in language model capabilities can be harnessed to significantly ease developer use of automated techniques, and we hope to develop more human-friendly automated debugging techniques as future work.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10664-024-10594-x.

Acknowledgements Sungmin Kang and Shin Yoo have been supported by National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (RS-2023-00208998), as well as the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2022-00155958, RS-2021-II211001, and RS-2022-II220995).

Funding Open Access funding enabled and organized by KAIST.

Data Availability Our code and experimental results can be found at https://doi.org/10.6084/m9.figshare.27861528.v3.

Declarations

Conflicts of Interest The authors declare that Shin Yoo is a member of the EMSE Editorial board. All co-authors have seen and agree with the contents of the manuscript and there is no financial interest to report.

References

Alaboudi A, LaToza TD (2020) Using hypotheses as a debugging aid. In: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), IEEE, pp 1–9

Allahyari M, Pouriyeh S, Assefi M, Safaei S, Trippe ED, Gutierrez JB, Kochut K (2017) Text summarization techniques: A brief survey. International Journal of Advanced Computer Science and Applications 8(10). https://doi.org/10.14569/IJACSA.2017.081052

Arcuri A, Briand L (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '11, pp 1–10. https://doi.org/10.1145/1985793.1985795

Arrieta AB, Díaz-Rodríguez N, Del Ser J, Bennetot A, Tabik S, Barbado A, García S, Gil-López S, Molina D, Benjamins R et al (2020) Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. Information Fusion 58:82–115

Böhme M, Soremekun EO, Chattopadhyay S, Ugherughe EJ, Zeller A (2017) How developers debug software - the dbgbench dataset. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp 244–246, https://doi.org/10.1109/ICSE-C.2017.94

Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A et al (2020) Language models are few-shot learners. Adv Neural Inf Process Syst 33:1877–1901

Chen M, Tworek J, Jun H, Yuan Q, Pinto HPdO, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G et al (2021) Evaluating large language models trained on code. arXiv:2107.03374

Coles H, Laurent T, Henard C, Papadakis M, Ventresque A (2016) Pit: A practical mutation testing tool for java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2016, pp 449–452

Dam HK, Tran T, Ghose A (2018) Explainable software analytics. In: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, Association for Computing Machinery, New York, NY, USA, ICSE-NIER '18, pp 53–56

Fried D, Aghajanyan A, Lin J, Wang S, Wallace E, Shi F, Zhong R, Yih Wt, Zettlemoyer L, Lewis M (2022) Incoder: A generative model for code infilling and synthesis. arXiv preprint arXiv:2204.05999

Gao L, Madaan A, Zhou S, Alon U, Liu P, Yang Y, Callan J, Neubig G (2022) Pal: Program-aided language models. arXiv:2211.10435

Gazzola L, Micucci D, Mariani L (2019) Automatic software repair: A survey. IEEE Trans Software Eng 45(1):34–67. https://doi.org/10.1109/TSE.2017.2755013

Goues CL, Pradel M, Roychoudhury A (2019) Automated program repair. Commun ACM 62(12):56–65

Gould JD (1975) Some psychological evidence on how people debug computer programs. Int J Man Mach Stud 7(2):151–182. https://doi.org/10.1016/S0020-7373(75)80005-8, URL https://www.sciencedirect.com/science/article/pii/S0020737375800058

Haas R, Elsner D, Juergens E, Pretschner A, Apel S (2021) How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2021, pp 1281–1291

Jiang J, Xiong Y, Zhang H, Gao Q, Chen X (2018) Shaping program repair space with existing patches and similar code. Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis

Jiang N, Liu K, Lutellier T, Tan L (2023a) Impact of code language models on automated program repair. 2302.05020

Jiang N, Lutellier T, Lou Y, Tan L, Goldwasser D, Zhang X (2023b) Knod: Domain knowledge distilled tree decoder for automated program repair. 2302.01857

Jiang S, Armaly A, McMillan C (2017) Automatically generating commit messages from diffs using neural machine translation. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 135–146, https://doi.org/10.1109/ASE.2017.8115626

Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '02, pp 467–477

Just R, Jalali D, Ernst MD (2014) Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2014, pp 437–440.https://doi.org/10.1145/2610384.2628055

Kirbas S, Windels E, McBello O, Kells K, Pagano M, Szalanski R, Nowack V, Winter ER, Counsell S, Bowes D, Hall T, Haraldsson S, Woodward J (2021) On the introduction of automatic program repair in bloomberg. IEEE Softw 38(4):43–51. https://doi.org/10.1109/MS.2021.3071086

Ko AJ, LaToza TD, Burnett MM (2015) A practical guide to controlled experiments of software engineering tools with human participants. Empirical Softw Engg 20(1):110–141. https://doi.org/10.1007/s10664-013-9279-3

Kochhar PS, Xia X, Lo D, Li S (2016) Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2016, pp 165–176, https://doi.org/10.1145/2931037.2931051

Layman L, Diep M, Nagappan M, Singer J, Deline R, Venolia G (2013) Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, pp 383–392, https://doi.org/10.1109/ESEM.2013.43

Lee J, Kang S, Yoon J, Yoo S (2024) The github recent bugs dataset for evaluating llm-based debugging applications. In: 2024 IEEE Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, Los Alamitos, CA, USA, pp 442–44https://doi.org/10.1109/ICST60714.2024.00049, URL https://doi.ieeecomputersociety.org/10.1109/ICST60714.2024.00049

Li X, Li W, Zhang Y, Zhang L (2019) Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2019, pp 169–180

Lim BY, Dey AK, Avrahami D (2009) Why and why not explanations improve the intelligibility of context-aware intelligent systems. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Association for Computing Machinery, New York, NY, USA, CHI '09, pp 2119–2128

Liu K, Wang S, Koyuncu A, Kim K, Bissyandé TF, Kim D, Wu P, Klein J, Mao X, Traon YL (2020) On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '20, pp 615–627, https://doi.org/10.1145/3377811.3380338

Marginean A, Bader J, Chandra S, Harman M, Jia Y, Mao K, Mols A, Scott A (2019) Sapfix: Automated end-to-end repair at scale. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp 269–278, https://doi.org/10.1109/ICSE-SEIP.2019.00039

Mariani L, Pastore F, Pezze M (2011) Dynamic analysis for diagnosing integration faults. IEEE Trans Software Eng 37(4):486–508. https://doi.org/10.1109/TSE.2010.93

Martinez M, Monperrus M (2019) Astor: Exploring the design space of generate-and-validate program repair beyond genprog. J Syst Softw 151:65–80

Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp 691–701,https://doi.org/10.1145/2884781.2884807

Monperrus M (2019) Explainable software bot contributions: Case study of automated bug fixes. In: 2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE), IEEE Computer Society, Los Alamitos, CA, USA, pp 12–1https://doi.org/10.1109/BotSE.2019.00010, URL https://doi.ieeecomputersociety.org/10.1109/BotSE.2019.00010

Monperrus M (2020) The Living Review on Automated Program Repair, URL https://hal.archives-ouvertes.fr/hal-01956501, working paper or preprint

Moon S, Kim Y, Kim M, Yoo S (2014) Ask the mutants: Mutating faulty programs for fault localization. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, pp 153–162,https://doi.org/10.1109/ICST.2014.28

Nijkamp E, Pang B, Hayashi H, Tu L, Wang H, Zhou Y, Savarese S, Xiong C (2022) Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint

Noller Y, Shariffdeen R, Gao X, Roychoudhury A (2022) Trust enhancement issues in program repair. In: Proceedings of the 44th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '22, pp 2228–2240,https://doi.org/10.1145/3510003.3510040

OpenAI (2023) Gpt-4 technical report. 2303.08774

Ouyang L, Wu J, Jiang X, Almeida D, Wainwright CL, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, et al. (2022) Training language models to follow instructions with human feedback. arXiv preprint arXiv:2203.02155

Rigby PC, Bird C (2013) Convergent contemporary software peer review practices. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2013, pp 202–212, https://doi.org/10.1145/2491411.2491444

Sadowski C, Söderberg E, Church L, Sipko M, Bacchelli A (2018) Modern code review: A case study at google. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, Association for Computing Machinery, New York, NY, USA, ICSE-SEIP '18, pp 181–190, https://doi.org/10.1145/3183519.3183525

Shinn N, Cassano F, Labash B, Gopinath A, Narasimhan K, Yao S (2023) Reflexion: Language agents with verbal reinforcement learning. 2303.11366

Siegmund B, Perscheid M, Taeumel M, Hirschfeld R (2014) Studying the advancement in debugging practice of professional software developers. In: 2014 IEEE International Symposium on Software Reliability Engineering Workshops, pp 269–274, https://doi.org/10.1109/ISSREW.2014.36

Sun C, Khoo SC (2013) Mining succinct predicated bug signatures. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2013, pp 576–586

Wang X, Wei J, Schuurmans D, Le QV, Chi EH, Narang S, Chowdhery A, Zhou D (2023) Self-consistency improves chain of thought reasoning in language models. In: The Eleventh International Conference on Learning Representations, URL https://openreview.net/forum?id=1PL1NIMMrw

Wei J, Wang X, Schuurmans D, Bosma M, hsin Chi EH, Le Q, Zhou D (2022) Chain of thought prompting elicits reasoning in large language models. ArXiv abs/2201.11903

Widyasari R, Sim SQ, Lok C, Qi H, Phan J, Tay Q, Tan C, Wee F, Tan JE, Yieh Y, Goh B, Thung F, Kang HJ, Hoang T, Lo D, Ouh EL (2020) Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2020, pp 1556–1560

Winter ER, Nowack V, Bowes D, Counsell S, Hall T, Haraldsson S, Woodward J, Kirbas S, Windels E, McBello O, Atakishiyev A, Kells K, Pagano M (2022) Towards developer-centered automatic program repair: Findings from bloomberg. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, pp 1578–1588, https://doi.org/10.1145/3540250.3558953

Xia CS, Zhang L (2022) Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, pp 959–971

Xia CS, Zhang L (2023) Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. arXiv preprint arXiv:2304.00385

Xia CS, Wei Y, Zhang L (2022) Practical program repair in the era of large pre-trained language models. arXiv preprint arXiv:2210.14179

Xiong Y, Wang J, Yan R, Zhang J, Han S, Huang G, Zhang L (2017) Precise condition synthesis for program repair. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp 416–426

Xiong Y, Liu X, Zeng M, Zhang L, Huang G (2018) Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '18, pp 789–799, https://doi.org/10.1145/3180155.3180182

Yao S, Zhao J, Yu D, Du N, Shafran I, Narasimhan K, Cao Y (2022) React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629

Yao S, Yu D, Zhao J, Shafran I, Griffiths TL, Cao Y, Narasimhan K (2023) Tree of thoughts: Deliberate problem solving with large language models. 2305.10601

Zeller A (2009) Why programs fail: a guide to systematic debugging. Elsevier

Zhu Q, Sun Z, Ya Xiao, Zhang W, Yuan K, Xiong Y, Zhang L (2021) A Syntax-Guided Edit Decoder for Neural Program Repair. Association for Computing Machinery, New York, NY, USA, pp 341–353

Sungmin Kang received the Ph.D. degree under the supervision of Prof. Shin Yoo. He is a postdoctoral researcher with Korea Advanced Institute of Science and Technology. His research interests include application of large language models to software engineering problems, as well as a theoretic analyses of probabilistic techniques to software engineering problems.

Bei Chen is a senior researcher at Microsoft. She received her Ph.D. degree from the Department of Computer Science and Technology at Tsinghua University, Beijing, China, in 2017. She is mainly working on pre-trained language models and multimodal understanding, and their applications in code intelligence. She has published above 40 papers in top conferences, including ICLR, NeurIPS, ACL, EMNLP, KDD, AAAI, IJCAI, etc.

Shin Yoo received the Ph.D. degree under the supervision of Prof. Mark Harman from King's College London, in 2009. He is a Tenured Associate Professor with Korea Advanced Institute of Science and Technology, and worked as a Lecturer of software engineering with the Centre for Research on Evolution, Search, and Testing, University College London from 2012 to 2015 before joining KAIST. His research interests include search based software engineering, software testing for AI systems, and the use of AI/ML for software testing as well as automated debugging.

Jian-Guang LOU is now a senior principal research manager at Microsoft. His main research interests include data mining and AI for software engineering, performance analysis and diagnosis of online services, chatbot and agent based on large language models. Many of his research results have been applied and deployed to the large-scale online services in Microsoft.