Enhancing Automated Program Repair with Solution Design

Jiuang Zhao*
zja1999@buaa.edu.cn
School of Computer Science and
Engineering, Beihang University
Beijing, China

Donghao Yang*
yangdonghao@buaa.edu.cn
School of Computer Science and
Engineering, Beihang University
Beijing, China

Li Zhang
lily@buaa.edu.cn
CCSE, Beihang University
Beijing, China

Xiaoli Lian†
lianxiaoli@buaa.edu.cn
CCSE, Beihang University
Beijing, China

Zitian Yang
yangzitian@buaa.edu.cn
School of Software, Beihang
University
Beijing, China

Fang Liu
fangliu@buaa.edu.cn
CCSE, Beihang University
Beijing, China

ABSTRACT

Automatic Program Repair (APR) endeavors to autonomously rectify issues within specific projects, which generally encompasses three categories of tasks: bug resolution, new feature development, and feature enhancement. Despite extensive research proposing various methodologies, their efficacy in addressing real issues remains unsatisfactory. It's worth noting that, typically, engineers have design rationales (DR) on solution— planed solutions and a set of underlying reasons—*before* they start patching code. In open-source projects, these DRs are frequently captured in issue logs through project management tools like Jira. This raises a compelling question: *How can we leverage DR scattered across the issue logs to efficiently enhance APR?*

To investigate this premise, we introduce *DRCodePilot*, an approach designed to augment GPT-4-Turbo's APR capabilities by incorporating DR into the prompt instruction. Furthermore, given GPT-4's constraints in fully grasping the broader project context and occasional shortcomings in generating precise identifiers, we have devised a feedback-based self-reflective framework, in which we prompt GPT-4 to reconsider and refine its outputs by referencing a provided patch and suggested identifiers. We have established a benchmark comprising 938 issue-patch pairs sourced from two open-source repositories hosted on GitHub and Jira. Our experimental results are impressive: *DRCodePilot* achieves a full-match ratio that is a remarkable 4.7x higher than when GPT-4 is utilized directly. Additionally, the CodeBLEU scores also exhibit promising enhancements. Moreover, our findings reveal that the standalone application of DR can yield promising increase in the full-match ratio across CodeLlama, GPT-3.5, and GPT-4 within our benchmark

suite. We believe that our *DRCodePilot* initiative heralds a novel human-in-the-loop avenue for advancing the field of APR.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Design rationale, Issue logs, Developer discussion, Automated program repair

1 INTRODUCTION

Due to persistent tasks such as fixing bugs, introducing new features, and improving existing ones, software maintenance consistently ranks as the most time-consuming phase in the software life cycle, even accounting for up to 90% of total costs [43]. This phase is particularly challenging because it requires maintainers to possess a deep understanding of the large and complex codebase, to analyze, design, and implement modifications without introducing new issues. To expedite this process, various Automated Program Repair (APR) techniques have been proposed to lessen the burden [14, 63]. However, their effectiveness in addressing real-world problems remains suboptimal [19, 39].

Let us reflect on the human process of software maintenance. In practice, engineers typically have an idea of the solution along with its associated considerations before beginning the actual code patching process. Within the open-source community, project management tools such as Jira are commonly employed to document this patching workflow. We depict this with the case study of addressing issue FLINK-32976, as presented in Fig. 1. The process is initiated by an issue report submitted by either engineers or users. Subsequently, contributors engage in discussions about the issue, possible solutions, and their respective advantages and drawbacks. An assignee then codes the solution, informed by these discussions. The final step involves the engineers submitting the patch to GitHub via a pull request.

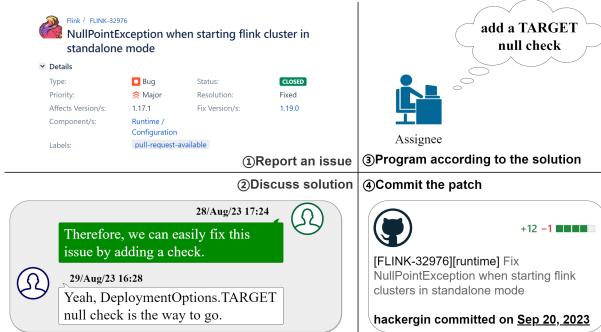*These authors contributed equally to this work.
†Corresponding author.

Figure 1: Four Stages of Issue Resolving in Open-source Community(Jira + GitHub): With One Example of FLINK-32976.

It is evident that solutions and their accompanying arguments in the discussion, collectively termed the design rationale (DR) [1, 7, 12, 29, 54, 58, 68], play an indispensable role in manual program maintenance [8]. They furnish engineers with guiding principles for developing patch, thus reducing the incidence of flawed fixes [72]. This observation prompts us to inquire: *How to effectively incorporate these design rationales to enhance the performance of current APR?*

In our examination of the current state of APR research, we observed limited utilization of design rationale from discussions. Semantic search methods primarily focus on code similarity search, leveraging existing code repositories to find fixes that previously worked in similar contexts[17, 26]. Semantic-constraint [31, 67] and pattern-based [21, 23, 25, 30] methods further optimize the search process through manually defined rules or templates. These methods are ineffective in leveraging higher-level repair knowledge from human experience [14]. Learning-based models [14, 28, 35, 36, 61] learn empirical knowledge from a large number of defect repair samples to guide the repair process. Existing work [40] has trained deep learning models to generate code patches by adhering to high-level guidelines (e.g. solution description summarized from discussions); however, the models' performance has been limited. This may be attributed to three key factors: firstly, the inherent noise and complexity present in interleaved conversational data [3], making it challenging to obtain high-quality design rationale from discussions [41]; secondly, the intricate process of collaborative problem-solving represented by design rationale [72], which demands strong reasoning abilities [10]; and lastly, the models may lack the requisite background knowledge to properly articulate the technical solutions under discussion [2, 55].

We note that work have emerged for efficiently mining design rationale from issue logs [71], providing a solid foundation for leveraging design rationale. Moreover, large language models (such as GPT-4-turbo [6]) have the potential to better bridge the gap between abstract design rationale and concrete source code patches [63] : Some work has demonstrated that LLMs can integrate different types of software artifacts (e.g., bug reports) to better repair programs [37, 70]. Therefore, we aim to explore methods for efficiently integrating design rationale into APR based on LLM and existing tools. The previously analyzed challenges in utilizing design rationale may lead to LLMs experiencing reasoning failures and code

hallucinations [55]. Fortunately, recent research has shown that LLMs can better handle complex problems through various types of feedback and self-correction mechanisms [16, 49]. In light of this, we introduce a feedback-based self-reflection framework to better empower LLM to apply design rationales.

Specifically, we introduce *DRCodePilot*, an approach that harnesses design rationale to drive automated patch generation. Initially, DRs are derived by extracting and pairing issue solutions with their corresponding arguments, leveraging the DRMiner tool[71]. Then GPT-4 pinpoints defective segments and generates patches by considering all DRs. For refining the patches, *DRCodePilot* collects feedback that encapsulates project-specific knowledge from two distinct sources:(1) reference patches generated by a fine-tuned CodeT5P [59], and (2) identifier replacement suggestions through a retrieval technique. Armed with this feedback, GPT-4 reflects on its initial answer, reasoning out final, refined patches.

To conduct an evaluation, we created a benchmark consisting of 938 issue-patch pairs by correlating Jira issues with their respective GitHub commits from two actively developed open-source projects: Flink and Solr. We selected five advanced code LLMs as baselines, including CodeLlama [45], StarCoder2 [34], CodeShell [66], GPT-3.5-Turbo, and GPT-4-Turbo (simplified as GPT-3.5 and GPT-4).

The results of our experiments demonstrate the superior performance of our *DRCodePilot*. Notably, in terms of the number of full-match patches (those identical to the gold patches), our model achieved *109 full matches out of 714 samples* in the Flink dataset and *18 out of 224* in the Solr dataset. This significantly surpasses the best-performing baseline model, GPT-4, which only managed *23 and 5 full matches* respectively, making our model 4.7 and 3.6 times more effective. Furthermore, *DRCodePilot* improved CodeBLEU scores by 5.4% and 3.9% over GPT-4. Our findings also highlight the beneficial influence of DR, alongside reference patch and identifier feedback, on the quality of generated patches. Despite the imperfections of automated DR extraction by DRMiner, the sole application of these extracted DRs has proven to obviously enhance the full-match ratio across all baseline models impressively. We also showcase that patch quality can be further elevated with improvements in DR quality.

Our main contributions are outlined as follows:

- **Framework**: We introduce *DRCodePilot*, a novel feedback-based self-reflective framework that mimics human process in software maintenance. It begins by generating initial patches grounded in design rationale and refines them based on the feedback of reference patches and identifier recommendations. To our knowledge, *DRCodePilot* is the pioneering effort to merge design rationale with APR techniques.
- **Data**: We collect up to 938 issue-patch pairs from two open-source projects and construct a dataset. We public the dataset and source code of *DRCodePilot*[1] to facilitate the replication of our study and its application in extensive contexts.
- **Evaluation**: We conduct experiments on our dataset to show that DRCodePilot achieves substantial improvements across different projects, especially a much higher full-match ratio. Ablation experiments further confirms the significant potential of applying design rationales in LLM-based software maintenance.

---

[1]https://figshare.com/s/82ed8e86e88d3268b4c1

2 MOTIVATING EXAMPLE

In this section, we illustrate the potential impact of design rationale on program maintenance using a simple issue example, FLINK-32976[2] in Jira, as depicted in Figure 2.

Figure 2: Classification of Issue Comments and Mapping from Solution to Patch for FLINK-32976.

The discussion of FLINK-32976 centers on a null pointer exception that occurs when starting a Flink standalone cluster with Hadoop configuration. The title and description explain the issue. During the comments zone, one participant *Hackergin* provided reproduction steps and the buggy code. He identified the root cause: the absence of a TARGET parameter check when the standalone cluster is initiated, leading to the null pointer exception. He also suggested one solution: adding a null check for the TARGET parameter, which all participants agreed was the best approach.

After identifying the solution, the assignee referenced the design rationales discussed to shape the final code implementation. However, there is a notable gap between design rationale and actual code implementation, requiring developers to have an in-depth understanding of the codebase. For instance, while the design rationale indicated the need to check the TARGET parameter, in practice, developers ensured robustness by setting an empty string as the default value.

We observed that despite the issue being classified as major and the solution has already been determined, it still took the maintainer 20 days to commit the changes. Such delays can introduce several risks. Automating adherence to these design rationales and generating patches could greatly improve efficiency—a capability not currently supported by existing APR techniques.

Moreover, while the example depicted has only 5 comments, more complex scenarios exist, such as FLINK-34007[3], which involved 82 detailed comments, debated three main solutions, the involving steps and arguments. In these complex issues, even advanced LLMs suffer from input length limits by injecting the whole issue logs, and the noisy nature of logs may hinder the accurate capture of design information [32]. Furthermore, due to the lack

of project-specific background knowledge, LLMs might produce erroneous outputs, leading to inefficiencies.

In summary, design rationales are a source of valuable guidance for engineers as they resolve practical issues. However, these rationales are often abstract and can sometimes be conflicting when multiple are presented for a single issue, necessitating a deep understanding of the project's intricacies by the engineers. Additionally, design rationales offer only high-level recommendations, granting engineers the discretion to determine if and how they should be applied. In effect, the challenge lies in harnessing these rationales automatically and effectively.

This is where our research comes into play. We aim to leverage the code comprehension and reasoning capability of large language models, exploring the extent to which DR can aid in their ability to automatically repair issues.

3 APPROACH

In this section, we discuss the design of *DRCodePilot*. *DRCodePilot* is designed to work in a realistic software development lifecycle, in which users submit issue reports to a software repository for bug fixing, feature addition or improvement, and the project maintainers have discussions about the issue before craft a patch to resolve it.

The *DRCodePilot* methodology unfolds in five strategic phases: 1) *Design Rationale Acquisition* (Section 3.1), where we replicate the existing work [71] to mine solutions and corresponding reasons for specific issues from Jira logs; 2) *Defective Segment Location and Draft Patch Generation* (Section 3.2), pinpointing flawed code areas and crafting initial patches via GPT-4, leveraging the gathered design rationales; 3) *Reference Draft Generation* (Section 3.3), employing a fine-tuned *CodeT5P* model against the full project repository to generate benchmarks, addressing GPT-4's long-context limitations; 4) *Identifier Recommendation* (Section 3.4), offering alternatives for the potential unsuitable identifiers in auto-generated segments; 5) *Final Patch Generation* (Section 3.5), directing GPT-4 to improve initial drafts based on reference patches and identifier suggestions.

3.1 Design Rationale Acquisition

The goal of this phase is to distill solutions and their respective arguments regarding specific issues, i.e., design rationales. Various methods have been proposed for extracting designs or solutions from diverse documents within the open-source community, such as mining issue-solution pairs from live chats [48], identifying question-answer dynamics [11], and uncovering design-centric discussions in pull requests [57], among others. For our purposes, we adopt the *DRMiner* approach [71], which specializes in deriving design rationales directly from issue logs using advanced LLMs. In essence, it combines tailored prompt instructions with specific heuristic features to pinpoint design-related text and align solution-argument pairs. Its superior performance has been validated across 2092 sentences from 30 issues spanning three publicly available open-source systems.

3.2 Defective Segment Location and Draft Patch Generation

DRCodePilot identifies the defective segment within the buggy function and generates the initial patch using the advanced LLM, GPT-4.
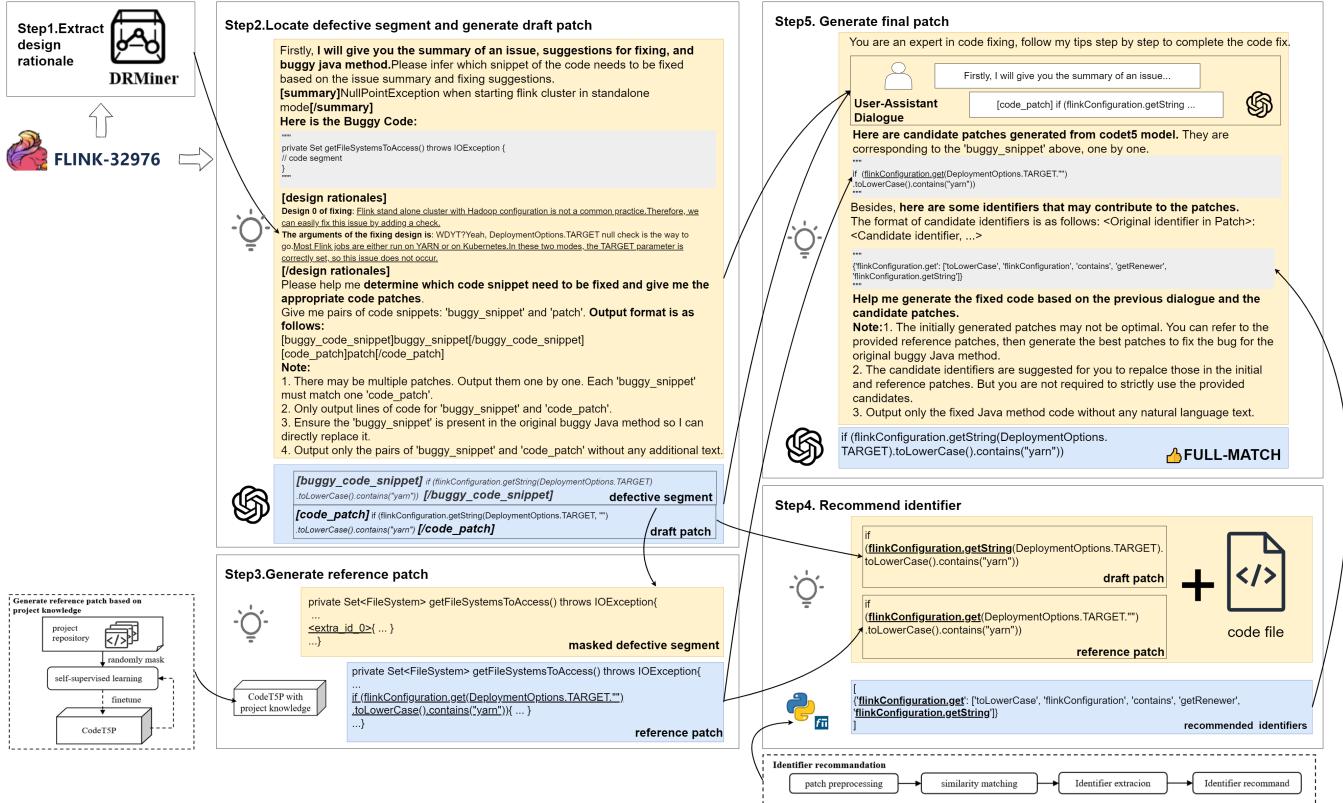
---

[2]https://issues.apache.org/jira/browse/FLINK-32976
[3]https://issues.apache.org/jira/browse/FLINK-34007

Figure 3: Overall Workflow of *DRCodePilot* on FLINK-32976.

GPT-4 is selected for its exceptional performance across a range of natural language tasks and for its capability to generate code based on provided instructions [6].

DRs extracted from issue logs with DRMiner are integrated into our prompt structure. As demonstrated in Figure 3, our prompt is composed of five sections:

(1) *An Instruction*: This is crafted to guide GPT-4 in identifying the defective segment and generating a patch based on the issue summary and fixing suggestions. (2) *Issue Summary*: Enclosed by *[Summary]* tags, this part briefly outlines the problem. (3) *Buggy Code*: A snippet of code, typically a function related to the issue. (4) *Design Rationale*: These design rationales are drawn from the issue comments by the tool of DRMiner. (5) *Output Instruction*: This specifies how the identified defective segments and the patches should be formatted.
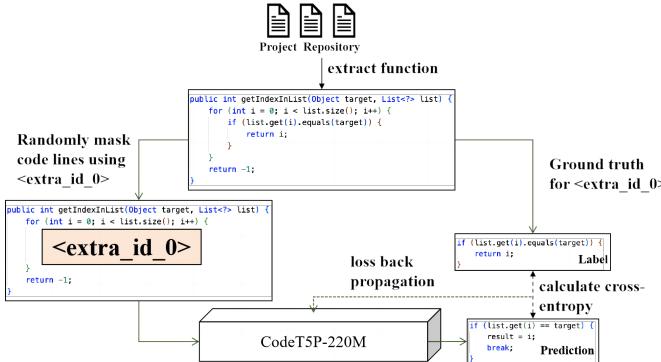
To ensure high-quality output, we impose four program-repair oriented guidelines for GPT-4 to adhere to rigorously: a) If there are multiple patches, they should be presented sequentially. b) The *buggy_snippet* and *patch* must be expressed as lines of code, without any natural language text. c) It is imperative that the *buggy_snippet* is directly excerpted from the original buggy Java method. d) Outputs should be limited strictly to pairs of *buggy_snippet* and *patch*, without any supplementary natural language explanation.

3.3 Reference Patch Generation

Advanced general-purpose LLMs are adept at generating code snippets from provided prompts, yet they may fall short in crafting precisely correct patches due to a lack of specific knowledge about the particular projects. To counter this, we utilize CodeT5P [59], a smaller model amenable to fine-tuning within a project's context, to supply reference patches as feedback for GPT-4. This approach allows GPT-4 to reassess and refine its responses when necessary.

Our methodology for generating reference patches is inspired by the principles set forth in [62]. As depicted in Step 3 of Figure 3, during the reference phase, we obscure the fault-ridden segments previously identified by GPT-4 in Step 2 using the marker <extra_id_0>. We selected CodeT5 due to its proficiency, having been pretrained on unimodal code corpora and bimodal code-text data, endowing it with considerable programming expertise [62]. To tailor it to the coding conventions of the target project, we further fine-tuned it on the project's codebase. Additionally, given the substantial resources required to train larger models, we opted for the 220M variant of CodeT5P in our study.

Fig. 4 shows the fine-tuning process. First, we build a training corpus from the project's codebase. For each Java file within this repository, we build an abstract syntax tree (AST). We then traverse through this AST to identify and extract the content of function nodes—termed "methods" in Java. To promote code conciseness, we exclude all comments and blank lines. Functions comprising less

Figure 4: Fine-tuning Process of CodeT5P-220M for Project Knowledge

.

than three lines are also excluded, ensuring that only those with sufficient complexity are considered. The bodies of these methods are masked using *<extra_id_0>*, aligning with CodeT5P's pre-training protocols.

To augment the fine-tuning corpus, we create multiple masked instances for each method so that every line within a method body gets masked at least once across different samples. Specifically, we designate the number of lines to be masked, denoted as α, which ranges from one to five. This range is informed by one observation suggesting that most defects in maintenance rarely exceed five lines [52]. Subsequently, we mask lines randomly based on the chosen scale α. In subsequent iterations, we select another quantity, β (ranging from 1-5, ensuring that $\alpha + \beta \leq n$, where n represents the number of lines in the method body), and proceed to mask different lines based on this new count. This process is repeated until all lines have been subjected to masking.

Once the training corpus is prepared, we fine-tune the model with the loss function of cross-entropy. During this phase, CodeT5P absorbs project-specific knowledge, including method names and variable identifiers, crucial for rectifying issues. This learning is vital because many bugs can be remedied using code snippets from elsewhere in the source file or elsewhere within the project [5].

3.4 Identifier Recommendation

Draft patches and reference patches may contain identifiers that are semantically similar but incorrect in the project context. As illustrated in Fig. 3, the reference patch incorrectly uses the *get()* call instead of the correct *getString()*. At this stage, we aim to offer identifier recommendations in these two kinds of patches as additional feedback to GPT-4 for further refinement.

Error-prone identifiers are detected based on a hypothesis: if an identifier appears solely in the generated patch and not elsewhere in the project code files, it is likely to be flawed. Considering the established observation that many solutions for bugs can be found within the same file [5], we source recommended identifiers just from the Java file containing the defect.

The core strategy for candidate identifier recommendation involves locating the most similar identifiers within code snippets that resemble the patch. Specifically, we divide the lengthy Java

file into snippets matching the length of the given patch. We then calculate the CodeBLEU [44] score between each snippet and the patch, utilizing CodeBLEU due to its widespread application in measuring code similarity, which accounts for both semantic and structural similarities. From the three snippets that exhibit the highest similarity, we extract identifiers and evaluate the cosine similarity between the vectors of these candidate identifiers and the error-prone identifier found in the patch. The top-3 identifiers are preserved for consideration.

Additionally, we compute the cosine similarity for identifiers outside the top-3 similar code snippets yet present in the defective Java file, selecting another set of top-3 identifiers as supplementary options. These may overlap with those chosen from the similar snippets. Consequently, for each error-prone identifier in the given patch, we provide up to six possible replacement candidates.

3.5 Final Patch Generation

In the final step, GPT-4 adjusts its initial patches by incorporating feedback from Step 3 and Step 4. Particularly, the model's responses are reformulated into a User Assistant-dialogue format, complemented with the draft and reference patches along with suggested identifiers. Using this context, GPT-4 crafts the enhanced patch.

This refinement occurs during Step 5, as illustrated in Fig. 3. Here, our prompt carefully defines GPT-4's role as a proficient code-fixing expert. By doing so, the model is directed to tap into its vast reservoir of training data, enabling the provision of sophisticated, expert-level code rectifications [22]. Furthermore, it follows a deliberate "follow my tips step by step" chain-of-thought procedure [60] (i.e., creating an initial patch, incorporating feedback, and refining the solution), which facilitates a clear understanding of the troubleshooting progression.

Within the reference patch area, we supply exemplars of efficient solutions to steer GPT-4 toward compliance with established coding norms and best practices. GPT-4 can also discover and correct fine-grained errors more effectively based on identifiers recommendations, thereby improving the usability of the final generated patch. Additionally, we outline three distinct output directives to guarantee that the model's output aligns with the project's goals, ensuring the final code is not only structured but also easy to maintain.

4 EXPERIMENTAL EVALUATION

4.1 Research Questions

To evaluate the capabilities of *DRCodePilot* in resolving real life software issues, we answer the following research questions:

RQ1 (Baseline Comparison): To what extent can *DRCodePilot* generate patches that effectively fix bugs?

RQ2 (Ablation experiment): Are the injected design rationale and feedback-driven refinement effective in *DRCodePilot*?

RQ3 (DR with other LLMs): What is the effect of integrating the extracted DRs into other LLMs, and does the quality of DRs matter?

4.2 Data Preparation

Our experimental benchmark is composed of two critical elements: real-world issues enriched with pertinent design rationale discussions, and the corresponding human-crafted patches. While Defects4J [20] holds the title as the most prevalent benchmark in

the APR domain, it lacks incorporation of developers' discussions. SWE-bench [19] has emerged as a benchmark that is rapidly gaining attention in the field. While it includes valuable developers' discussions, it unfortunately lacks critical meta-data labels such as the creator's name, comment participants, and comment indexing. This omission presents challenges for effectively extracting design rationales.

To address these deficiencies, we have developed our benchmark by selecting issues from the open-source systems Flink and Solr on Jira, ensuring comprehensive meta-data collection from issue logs, and correlating these with relevant GitHub pull requests (PRs) by tracing back through the provided issue links. Our focus on Flink and Solr stems from their status as emblematic large-scale software systems grappling with a wide array of problems.

We refine our dataset to only include instances where issues are closed and PRs are merged, which affords us the use of these human-written patches as verified ground truth. Harmonizing with Defects4J's approach, our filtration process scrutinizes patch files, guided by the parameters delineated in Table 1. Our priority lies with Java source file fixes that pivot on code logic—excluding unit tests, configuration files, and the like—and we narrow our scope to PRs altering solely one source file, with a maximum threshold set at 22 lines of change. This cut-off aligns with findings from Defects4J where 95% of patches adjust under 22 lines [52].

The data presented in Table 2 showcases the distribution of 'gold' patches based on the number of modified code lines within our benchmark. Notably, the majority of issues are resolved by altering fewer than 10 lines of code. This benchmark, inspired by Defects4J, includes issues that are, on average, less complex than those found in SWE-bench—judging by the scale of changes in lines and files. The rationale for this approach is to primarily investigate the extent to which design rationales embedded in developers' discussions can aid large language models in issue resolution. To this end, we have chosen to match the complexity level found in traditional Defect4J benchmarks. Looking ahead, we intend to tackle more challenging issues by incorporating design rationales from comments.

Furthermore, there is a noticeable disparity in data volume between Solr (1.1k stars) and Flink (23.3k stars), which likely stems from the different levels of popularity the two projects enjoy on GitHub; lower popularity often results in fewer commits, issue reports, and subsequent fixes.

It is also important to highlight that we did not build an execution-based evaluation pipeline for these patches. Although both projects feature comprehensive unit test suites, the immense evolution of these projects over time—Flink over 10 years and 20 years for Solr—has led to significant changes in both Maven project structures and their library dependencies. This evolution posed challenges in compiling test suites for an execution-based evaluation pipeline, as outdated library versions caused unresolved dependencies. Consequently, even among expert-written and merged code patches, less than 1% are amenable to executable evaluations.

Additionally, to avoid unfair comparisons caused by patches appearing directly in comments, all code snippets in comments were replaced with [code] tags.

Table 1: Key Indicators of Our Benchmark.

Indicator	Condition
Included projects	Solr and Flink
Data collection source	Closed issues and merged PRs
Issue metadata	Yes
File types for fixes	Java source files
Number of files modified per PR	One source file
Number of code lines changed per PR	Maximum of 22 lines
Length of issue comments	Unlimited

4.3 Experiment Setting

Evaluation Metrics. We employ two metrics to assess the effectiveness of the various APR approaches involved. First, we tally the number of *full-match* patches—those that are identical to the provided gold patches. We did not utilize the widely applied pass@k criterion based on test cases for two primary reasons. Firstly, as outlined in Section 4.2, constructing test cases for legacy Java projects presents significant challenges. More importantly, this choice is predicated on the reality that in professional settings, patches must undergo a rigorous manual review by multiple engineers before they can be deemed suitable for integration into the project—even if they succeed in passing predefined tests. Hence, we employ the 'Full-Match' criterion to select patches that are identical to the manual patches and ready for use as solutions.

Second, we apply *CodeBLEU* [44], a widely recognized metric for evaluating code generation quality that compares the semantic and syntactic precision of the generated code against gold code. It should be noted that CodeBLEU is better suited for assessing the quality of code at the level of functions or code snippets, due to its comprehensive analysis of components like syntactic accuracy, data flow, and logical structure, which typically span multiple lines of code. As per Table 2, it is observed that over half of the patches involve fewer than five lines of change. Therefore, we calculate CodeBLEU scores for the entire repaired function, meaning the original function wherein the flawed segment has been modified with the suggested patch.

Baselines. The baseline methods selected in this experiment include four state-of-the-art large code models:

- *StarCoder2* [34]: Through a new training technique called Code-BERTa, StarCoder2 improves the model's generalization ability and understanding of programming languages without increasing the number of parameters. The model version used in this experiment is StarCoder2-15B.
- *CodeLlama* [45]: Based on pre-training from LLaMA2 [56], CodeLlama enhances coding capabilities, better follows human instructions, and understands zero-shot tasks. The model version taken for this experiment is CodeLlama-7B.
- *CodeShell* [66]: Using high-quality pre-training data, CodeShell fuses the core features of StarCoder [27] and LLaMA2, supports code-specific generation methods, and has a high-performance and easily extensible context window architecture. CodeShell has a model parameter count of 7B. It outperforms models with the same number of parameters on the HumanEval test.
- *GPT-3.5 (gpt-3.5-turbo-1106)*: We choose GPT-3.5 because it is free to use and is widely used as an effective aid to coding, and

Table 2: Distribution of the Patches over the Changed Code of Lines in Our Benchmark.

	$1 \leq n < 5$	$5 \leq n < 10$	$10 \leq n < 15$	$15 \leq n < 20$	$20 \leq n < 22$	Total
Flink	514 (71.99%)	139 (19.47%)	40 (5.60%)	12 (1.60%)	9 (1.26%)	714
Solr	125 (55.80%)	75 (33.48%)	12 (5.36%)	4 (1.79%)	8 (3.57%)	224

researchers can easily access and utilize it for experimental comparisons.

- *GPT-4 (gpt-4-1106-preview)* : Successor to GPT-3.5, but with major enhancements in model size, training data, and capabilities, reportedly up to 1 trillion parameters[4]. It is often considered to represent the highest level of the current large language models.

4.4 Baseline Comparison (RQ1)

Table 3 showcases the comparative evaluation of our design rationale-driven approach, *DRCodePilot*, against baselines that generate patches directly from buggy functions. We detail the number of full matches and their proportion relative to all samples. Our method outperforms the baselines across both systems in the benchmark, as indicated by the number of full-match instances and CodeBLEU scores. Among the baselines, GPT-4 leads, followed by GPT-3.5, CodeLlama, StarCoder2, and CodeShell.

DRCodePilot significantly surpasses the best-performing baseline, GPT-4, in generating full matches. For instance, it achieves 109 full matches for Flink, which is about 4.7 times more effective than GPT-4. In Solr, the number of full matches by *DRCodePilot* is up to 3.6 times that achieved by GPT-4. The CodeBLEU scores also position *DRCodePilot* at the forefront within both systems' benchmarks. The increase in CodeBLEU, while not as pronounced as the full-match improvement, can be attributed to the need for calculating CodeBLEU over complete functions [44], where the altered code may only constitute a minor segment of the whole function. This implies that generated patches have a constrained influence on CodeBLEU scores. Nonetheless, this metric still reflects the similarity between the patches and the 'gold' standard answers.

Upon cross-examining the two benchmarks, we noted that the full-match ratio of *DRCodePilot* in Flink (15.27%) exceeds that in Solr (8.03%), a trend consistent among the other baselines. Excluding *DRCodePilot* and GPT-4, all other baselines register better CodeBLEU scores in Flink compared to Solr. A contributing factor is the greater complexity of issues in Solr, as indicated by a higher proportion of patches with changed lines in the ranges of [5,10],[15,20], [20,22], and much less proportion in the range of [1,5], compared to those in Flink (refer to Table 2).

4.5 Ablation Study (RQ2)

Three types of knowledge play a crucial role in our *DRCodePilot*: design rationale for initial patch generation, reference patches and identifier recommendations for patch optimization. We aim to assess the contribution of each knowledge type to the quality of the final patch produced.

To accomplish this, we crafted three variants of *DRCodePilot* for our study: 1) *DRCodePilot-DR*, which omits design rationale from the prompt during the initial patch creation phase (removing

Step 1 and changing Step 2 in Fig.3). 2) *DRCodePilot-PF*, which does not incorporate the generation of reference patches (removing Step 3 from DRCodePilot shown in Fig. 3). And 3) *DRCodePilot-ID*, which removes identifier recommendations from the process (excluding Step 4 from *DRCodePilot* illustrated in Fig. 3). All other experimental variables and procedures are kept consistent across the three variants.

These three variants were assessed using our benchmark, with the final results presented in Table 4. To better illustrate the comparative performance, we calculate the performance drop ratio for each variant relative to the original *DRCodePilot* across the two metrics. Specifically, if the metric value for *DRCodePilot* is denoted as α, and the corresponding value for one variant is β, then the performance drop ratio is computed as $\frac{\alpha - \beta}{\alpha}$. A positive ratio, indicated by a downward arrow and red font, signifies that excluding a certain type of knowledge has weakened the performance of that variant.

Generally, the absence of any knowledge type tends to diminish the model's performance, affecting both the count of full-match patches and CodeBLEU scores. Notably, the most pronounced performance decrease is observed with *DRCodePilot-DR* in terms of both the number of full matches and CodeBLEU metrics. The count of full-match patches plummeted by 81.65% and 94.44%, while the CodeBLEU scores fell by 9.17% and 4.87% across the two systems, respectively. This suggests that incorporating design rationales into the initial patch generation prompts can significantly improve patch quality, as they offer pertinent discussions that catalyze GPT-4's issue resolution capabilities. It's unsurprising that the decline in CodeBLEU is more modest than in full-match counts due to the relatively small proportion of changed lines within the entire defective functions.

Additionally, the greater reduction in full-match patches observed in Solr compared to Flink may suggest that design rationales assume a more crucial role in resolving complex issues, as evidenced by the larger amount of code line changes in Solr relative to Flink. We aim to further investigate this hypothesis in future work.

The second most noticeable performance decline is seen with *DRCodePilot-ID*, that is, upon removing identifier recommendations (Step 4), with drops of 3.67% and 5.55% in full-match counts, and 0.51% and 0.87% in CodeBLEU, respectively, for the two systems. These figures reveal that even an advanced code-focused LLM like GPT-4 can err in generating precise correct identifiers during issue repairs. However, our feedback-based, self-reflective prompt guides it towards re-evaluation and refinement of unsuitable identifiers.

Upon reviewing the outcomes of *DRCodePilot-PF*, we observe that the count of exact-match patches remains constant in Flink and experiences a marginal increase in Solr. Concurrently, all CodeBLEU scores within these two systems exhibit a downturn. This indicates that GPT-4 does not merely replicate the reference patch but rather assimilates pertinent information from such feedback to refine its

---

[4]https://the-decoder.com/gpt-4-has-a-trillion-parameters/

Table 3: Comparing the Performance of Different Methods on FLink and Solr

Approach	Flink (Total: 714 samples)		Solr (Total: 224 samples)	
	Full-Match	CodeBLEU	Full-Match	CodeBLEU
CodeShell	6(0.84%)	0.71	0	0.61
CodeLlama	8(1.12%)	0.64	2(0.89%)	0.62
StarCoder2	9(1.26%)	0.74	0	0.64
GPT-3.5	15(2.10%)	0.72	1(0.45%)	0.69
GPT-4	23(3.22%)	0.74	5(2.23%)	0.77
DRCodePilot	**109**(15.27%)	**0.78**	**18**(8.03%)	**0.80**

Table 4: Ablation Results on Our Benchmark.

Approach	Flink		Solr	
	Full-Match	CodeBLEU	Full-Match	CodeBLEU
DRCodePilot-DR	20 (↓ 81.65%)	0.713 (↓ 9.17%)	1 (↓ 94.44%)	0.762 (↓ 4.87%)
DRCodePilot-PF	109 (-)	0.783 (↓ 0.26%)	19 (↑ 5.55%)	0.800 (↓ 0.12%)
DRCodePilot-ID	105 (↓ 3.67%)	0.781 (↓ 0.51%)	17 (↓ 5.55%)	0.794 (↓ 0.87%)
DRCodePilot	**109**	**0.785**	**18**	**0.801**

responses. Furthermore, we aim to investigate scenarios where project-specific reference answers contribute to the creation of superior quality patches. A thorough examination of the exact-match patches produced by both *DRCodePilot* and *DRCodePilot-PF* suggests that when design rationales behind issue resolutions are discernible, the sophisticated GPT-4 model demonstrates proficiency in generating high-caliber patches. In contrast, reference patches derived from simpler models can erode its confidence, leading to the displacement of exact matches. On the flip side, in cases where commentary on issues is scant, reference patches seem advantageous. To elucidate this phenomenon, we provide a representative example in Figure 5.

Figure 5: One Example Indicating the Usefulness of Project Knowledge-based Reference Patch

.

One example illustrating the utility of project knowledge-based reference patches: Take the issue FLINK-23579[5] as an instance. The associated issue log includes a mere pair of comments, with the first pinpointing the root cause yet neither suggesting a fix. Consequently, we cannot incorporate any design rationale into the initial

---

[5]https://issues.apache.org/jira/browse/FLINK-23579

patch generation prompt. GPT-4 formulates the initial patch solely based on internal knowledge, contrasting with the reference patch crafted using project-specific insights. The reference patch highlights the overloaded *hash()* function, which aligns with the actual fix implemented. Evidently, the reference patch closely matches the engineer's approach. This exemplifies how project knowledge can effectively supplement design rationales, especially when explicit solutions are absent, and the model must discern the most project-appropriate solution among several possibilities.

4.6 General Impact of Design Rationale (RQ3)

In this section, we aim to address three sub-questions. Firstly, our framework has thus far only utilized design rationales in conjunction with GPT-4. We are intrigued by the broader applicability of this approach in APR tasks. Specifically, we seek to understand *the extent to which these design rationales may bolster the effectiveness of different models engaged in program repair (RQ3.1).*

Secondly, the design rationales we have employed were autonomously derived using the DRMiner methodology [71]. However, the evaluation outcomes presented in the originating paper indicate that the extraction process is not infallible, evidenced by inaccuracies and omissions. In light of this, our experiment sets out to ascertain *the degree of improvement in program repair performance achievable through the integration of meticulously hand-annotated design rationales (RQ3.2).*

Moreover, given that the design rationales are gleaned from developers' comments, a natural question arises: *can advanced general-purpose LLMs independently comprehend these comments and distill pertinent insights to facilitate issue resolution (RQ3.3)?*

To explore the above sub-questions, we selected three baseline models capable of interactive prompting: CodeLlama, GPT-3.5, and GPT-4.

Since expert annotations are involving, we randomly selected 61 issues that featured extensive comments from our benchmark. We focus on the quantity of comments because design rationales are often given in the comments, and we aim to assess the impact of design rationale quality. In other words, the presence of design rationales is a necessary condition for our issue selection.

We enlisted 16 participants with varied backgrounds, including two professors with over 12 years of academic experience in computer science; seven Ph.D. students and six postgraduate students engaged in both academic and industrial software research; along with one software engineer boasting more than six years of development experience at internet companies.

The annotation process involved independent analyses and joint discussions among three participants. Specifically, each issue was assigned to two of them, who would annotate the design rationales independently. Once the independent annotations were completed, a discussion involving the two initial annotators and a third participant took place. In cases of disagreement, a majority vote decided the final labeling outcome.

The evaluation results are summarized in Table 5, from which we can draw the following insights:

- *Generality (RQ3.1)*: Our approach and all three baseline models demonstrate promising results when utilizing design rationales, even when these are imperfectly extracted by a tool (+DR). Without design rationale support, the three baseline models could only generate one or two exact match patch corresponding to the reference solution. However, when provided with design rationales—even noisy ones—the figures rise to 11 and, impressively, 19 for the more advanced GPT-4 (11x-19x).

- *Better Design Rationale, Better Patch (RQ3.2)*: Across the board, more full-match patches are produced by all baselines and our *DRCodePilot* with the deliberately annotated design rationales (+DDR). Nevertheless, it's noteworthy that the CodeBLEU score for GPT-3.5's output experienced a decrease. A probable explanation is that *DRMiner* favors recall over precision [71], indicating that the extraction process includes some sentences unrelated to design rationale. However, GPT-3.5 can find useful hints from these noisy information to enhance program repair.

  This reveals the disparity between automated tools for design rationale extraction and expert annotations. Future research is expected to narrow this gap and enhance the efficiency of utilizing design rationales. In conclusion, the results confirm the influence of design rationales on repair results, with more precise design rationales leading to better code repair outcomes.

- *Enhanced Performance with Developer Comments Over No Comments, Yet Surpassed by Design Rationale Integration (RQ3.3)*: The inclusion of original developer comments (+AC) distinctly enhances program repair performance for all models compared to those that do not incorporate discussions. An evident increase in the number of full-match patches is observed across all four models, supporting the findings of Panthaplackel et al. [40] which emphasize the constructive role of developer conversations in bug-fixing activities.

  Furthermore, the degree of this enhancement varies among models of different complexities. CodeLlama and GPT-3.5 demonstrate around an elevenfold increase in full-match patches, while GPT-4 shows a twelfthfold improvement. Our *DRCodePilot* especially stands out, achieving twenty-two times more full-match patches, showcasing the considerable advantage of directly leveraging developer comments. The rationale is straightforward: advanced models such as GPT-4 are capable of assimilating and deciphering valuable solution-oriented design information from developer comments.

However, upon examining the data for +AC, +DR, or +DDR, it becomes clear that models utilizing design rationales consistently outperform those relying solely on original comments, in terms of both full-match patches and CodeBLEU scores. This suggests that despite the presence of design rationales within the comments, general models struggle to pinpoint these key insights amidst the intertwined discussions, and this challenge persists even for sophisticated models like GPT-4. Therefore, there is a need to employ an additional tool to initially extract the solution-oriented design information before incorporating it into prompts to guide GPT-4 towards generating superior patches.

Additionally, it is evident that our *DRCodePilot* retains its frontrunner status than the baselines, even with the baseline models showing enhanced performances through the incorporation of design rationales. Nevertheless, the performance gap between GPT-4 and our *DRCodePilot* is relatively small compared to other baselines. As previously mentioned, the 61 issue logs contain abundant comments, which suggests the presence of extensive discussion and high-quality design rationale. In such scenarios, GPT-4 is capable of generating high-quality patches. However, we incorporate feedback mechanisms and self-reflective prompts, recognizing that it's not always possible to guarantee the existence of high-quality comments for all issues, particularly new and unresolved ones.

Table 5: The Effect of Design Rationale Quality on Varied Models' Performance (with 61 cases). "+DR" denotes the inclusion of automatically mined design rationales, "+DDR" signifies the addition of deliberately manual-annotated design rationales, and "+AC" denotes the incorporation of all developer comments in each issue log.

Approach	Full-Match	CodeBLEU
CodeLlama	1	0.68
CodeLlama+DR	11	0.75
CodeLlama+DDR	14	0.83
CodeLlama+AC	11	0.80
GPT-3.5	1	0.75
GPT-3.5+DR	11	0.71
GPT-3.5+DDR	14	0.66
GPT-3.5+AC	11	0.70
GPT-4	1	0.75
GPT-4+DR	19	0.85
GPT-4+DDR	21	0.84
GPT-4+AC	19	0.87
DRCodePilot w/o DR	1	0.76
DRCodePilot (+DR)	26	0.88
DRCodePilot+DDR	**26**	**0.88**
DRCodePilot+AC	22	0.86

5 RELATED WORKS

5.1 Design Rationale Extraction

The design rationale encompasses the alternative design options considered and decided upon throughout the software life cycle [12, 33, 38], as well as the justifications for accepting or rejecting certain alternatives. It is widely applied to assist human engineers in development and design activities [4, 9, 15, 54, 58, 69]. Effectively documenting design rationale is critical to maintaining the long-term health and vitality of a project, allowing software developers to understand past decisions and continue to implement these core design ideas in future updates and maintenance.

In recent years, automatically mining design knowledge from various sources (such as emails, meeting notes, or open-source communities) has attracted widespread research interest [3, 24, 41, 46–48, 53, 57], as a substantial number of decisions actually occur in informal discussions. For instance, Panthaplackel et al. [41] generated descriptions of solutions by synthesizing relevant content. However, human evaluators scored the informativeness of the descriptions (using a scale of 1-5) at an average of 3.3, suggesting that the descriptions include useful information but does not capture the solution well.

A major challenge of this task is that these discussion texts are interleaved, noisy, and the expressions of design rationale are diverse and abstract [3, 10], which makes previous methods ineffective in acquiring design rationales from issue logs. The recently emerged DRMiner [71], leveraging customized features and large language models, achieves promising performance on the design rationale extraction. It effectively identifies solutions and their associated arguments from issue logs on Jira, laying a solid foundation for leveraging design rationale in software activity.

5.2 Automatic Program Repair

Automated program repair (APR) technologies have exerted significant influence across various domains such as software engineering, system security, and artificial intelligence [14]. APR can broadly be classified into four categories: (1) *Semantic search-based methods* [17, 26] analyze the structure and contextual information of code to find and recommend repair solutions. (2) *Semantic constraints-based methods* guide the repair process by developing a set of constraint specifications, transforming program repair problems into constraint solving problems [31, 67]. (3) *Pattern-based techniques* reduce search space through template matching, which can be manually extracted or automatically mined to guide specific types of defect repairs [21, 23, 25, 30]. (4) *Learning-based APR* learn the experiential knowledge of program repair from a large number of repair samples, having the flexibility and scalability to follow complex instructions and handle types of defects [28, 35, 36, 41, 61, 62, 64]. Panthaplackel et al. obtain useful information from discussions (e.g. solution description) to enhance learning-based APR [40]. They fine-tuned a sequence-to-sequence model to generate the fixed code given varying input context representations.

Recent work has explored applying code large language models for APR [27, 34, 45, 59, 66]. A prevalent paradigm involves formulating APR as a code generation task, and leveraging prompt engineering techniques to steer the model towards generating more effective patches[42, 63, 65]. One challenge of leveraging LLM is to

overcome code hallucination , where LLM sometimes generate code that appears plausible but fails to meet the expected requirements or executes incorrectly [55]. Some studies suggest fine-tuning the LLM for program repair to enhance its effectiveness [13, 18, 51], but the cost is high. Other methods include providing feedback to the model based on external tools and guiding to self-correct [16, 49], or providing useful information to the model such as bug reports [23, 37], retrieved API context [70], etc.

In this paper, we extract comprehensive design rationales (e.g. alternative solutions and arguments) , to guide LLM in generating patches using zero-shot inference. To better generate feasible patches, we further devise a feedback mechanism by addressing potential technical information gaps in design rationales.

6 DISCUSSION

6.1 Failure Case Analysis

To better understand the conditions that influence DRCodePilot's performance, we analyzed 72 patches: 36 full-match patches (18 randomly selected from Flink and all 18 full-match patches from Solr) and 36 lower-quality patches (randomly selected in the same way, all with a CodeBLEU score below 0.5). Through comparisons, we find two scenarios that our *DRCodePilot* may not perform as expected.

- *Quality and Richness of the Extracted DR largely impact the performance of DRCodePilot*: The efficacy of *DRCodePilot* is intrinsically linked to the quality and comprehensiveness of the extracted DRs. Challenges arise when DR is limited or absent—such as when developers' discussions go unrecorded (e.g., FLINK-28513)—resulting in subpar DRs that hinder DRCodePilot's performance. For simple issues, *DRCodePilot* can generate fine patches. However, it stuggles with complex issues.

- *Insufficient Contextual Knowledge in DRs*: Our approach hinged exclusively on the "one-stop" DR, which is directly gleaned from developer discussions. We refrained from incorporating supplementary context to fill gaps essential for a comprehensive understanding. The absence of such extended contextual knowledge can render the extracted DRs obscure, hampering DRCodePilot's capacity to infer solutions accurately, and consequently resulting in less-than-optimal results. We recognize three particular scenarios illustrative of this limitation:

  – **Code References in DR Solutions:** If a DR refers to code specifics but omits the actual code snippet, like in SOLR-15896 where it instructs to 'Just copy the *toString* method and change the broken parts,' without including the *toString* content, *DRCodePilot* finds it challenging to generate patches aligned with manually crafted ones.

  – **Involvement of External Links:** Instances where DRs point to external resources or related issues can deprive *DRCodePilot* of critical context. An example is FLINK-28488, where a DR references FLINK-27487 for key details. However, our methodology is not designed to extract these particulars, thereby omitting the integration of vital insights.

  – **Unclear Code Intent:** At times, developer discussions may highlight errors without specifying the correct behavior. This impedes DRCodePilot's patch quality, as in FLINK-15386, where a DR identifies a logical mistake but fails to convey the desired

logic, leaving the model at odds with deriving an accurate fix despite recognizing the error.

6.2 Threats to Validity

The principal threat to the *construct validity* in our study is associated with the choice of benchmark. Publicly available benchmarks, such as Defects4J [20] and SWE-bench [19], were not employed due to their lack of developer discussions or the meta-data labels of discussions —data critical for extracting design rationales. Accordingly, we constructed a new benchmark by aligning issues from Jira with their corresponding patches in GitHub. We believe that our custom benchmark will unlock novel possibilities for advancing design-guided program repair research.

We limited our evaluation to design rationales extracted solely from Jira, a single issue tracking system. Other platforms such as GitHub issues also host design discussions, which could affect the *external validity* of our findings. Given the analogous metadata structures between Jira and GitHub issues, we anticipate that our approach would be compatible. Nonetheless, further assessment across different platforms is warranted to validate this assumption.

Manual annotation often poses a threat to *internal validity* due to potential inconsistencies or biases. To mitigate such risks, we employed standard procedures involving independent annotations followed by collaborative discussions for consensus building, ultimately deciding outcomes through majority vote.

Another potential threat to *internal validity* stems from the content within developer comments. It is unsurprising that comments, or the design rationales derived from them, play a significant role if they contain explicit target code snippets. In practice, developer discussions are typically high-level and advisory in nature. Nonetheless, to minimize the impact of such rare occurrences, we ensure all code within the comments is obscured using a [code] token.

6.3 Limitations

- *Absence of Repair-Specific Baselines*: The baselines utilized in this study are general-purpose code LLMs. Specific bug-fixing models such as RepairLLaMA [50] were not selected because they do not fit our experimental context. Most specialized models require additional inputs, like the defective locations in the buggy function [50] or a buggy template [30]. In our study, such supplementary information is not available.
- *Evaluation Limited to Simpler Issues*: Echoing the benchmark of Defects4J [20], where 95% of patches involve fewer than 22 lines of changed code, our dataset selection was similarly scoped to primarily assess the impact of design rationales on APR with relatively simpler issues. Nonetheless, considering the strategic essence of design rationales, they hold the potential to assist in resolving more intricate and demanding problems.
- *Lack of Separate Impact Analysis for Solution and Argument in APR*: While this study demonstrates the beneficial effect of design rationale on patch quality, it does not dissect and scrutinize the distinct impacts of solutions and arguments. Such an analysis is vital because more methods can identify or derive solutions from online developer discussions [41, 48]. Should the arguments prove to be less critical for patch generation, we could employ

more techniques focused on generating solutions that could more effectively steer the patch generation process.

- *Absence of Execution-Based Evaluation Pipeline*: While we assessed patch quality using static metrics such as CodeBLEU and full-match, these methods have notable limitations. For instance, CodeBLEU evaluates the entire function rather than focusing on the modified segments, which can result in inflated scores. Unchanged portions of the function contribute positively to the score, even though they are irrelevant to the actual repair. This skews the evaluation by overestimating the patch quality. In contrast, an execution-based approach would offer a more accurate measure of whether the patch genuinely resolves the issue by directly testing the code. Without this, our current evaluation risks missing important nuances in functional correctness and real-world applicability. Therefore, future work would benefit from integrating an execution-based evaluation pipeline to provide a more reliable and comprehensive validation of the patches.

7 CONCLUSION AND FUTURE WORK

Emulating the typical development process where solutions and their respective arguments are deliberated *prior* to crafting patches, we introduce *DRCodePilot*, a design rationale-driven program repair approach leveraging the capabilities of the advanced GPT-4. Besides, we devised a feedback-based self-reflective framework to prompt GPT-4 to revisit and refine its proposed fixes according to the given reference patches and suggestions for identifier replacement. For assessment, we curated a benchmark comprising 938 matched issue-patch pairs. Our experiments reveal that *DRCodePilot* achieves up to 4.7x and 3.6x more full-match patches than the leading GPT-4 in our benchmark. Furthermore, we illustrate the constructive influence of design rationales, reference patches, and identifier recommendations. Moving forward, we plan to improve upon extracting design rationales and to incorporate retrieval-augmented generation techniques to tackle increasingly complex issues.

ACKNOWLEDGMENTS

REFERENCES

[1] Rana Alkadhi, Teodora Laţa, Emitza Guzman, and Bernd Bruegge. 2017. Rationale in Development Chat Messages: An Exploratory Study. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, Piscataway, NJ, USA, 436–446. https://doi.org/10.1109/MSR.2017.43

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 281–293. https://doi.org/10.1145/2635868.2635883

[3] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 454–464. https://doi.org/10.1109/ICSE.2019.00058

[4] Muhammad Ali Babar and Ian Gorton. 2007. A Tool for Managing Software Architecture Knowledge. In *Proceedings of the Second Workshop on SHAring and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI '07)*. IEEE Computer Society, Washington, DC, USA, 11–. https://doi.org/10.1109/SHARK-ADI.2007.1

[5] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (16–22). Hong Kong, China, 306–317.

[6] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. arXiv:2303.12712 [cs.CL]

[7] Janet E Burge and David C Brown. 2008. Software engineering using rationale. *Journal of Systems and Software* 81, 3 (2008), 395–413.

[8] J. E. Burge, D. C. Brown, and D. C. BROWN. 2003. Rationale Support for Maintenance of Large Scale Systems. In *Workshop on Evolution of Large-Scale Industrial Software Applications (ELISA), ICSM '03, Amsterdam, NL, 2003.*

[9] Rafael Capilla, Francisco Nava, Sandra Pérez, and Juan C. Dueñas. 2006. A Web-based Tool for Managing Architectural Design Decisions. *SIGSOFT Softw. Eng. Notes* 31, 5, Article 4 (Sept. 2006). https://doi.org/10.1145/1163514.1178644

[10] Senthil K. Chandrasegaran, Karthik Ramani, Ram D. Sriram, Imré Horváth, Alain Bernard, Ramy F. Harik, and Wei Gao. 2013. The evolution, challenges, and future of knowledge representation in product design systems. *Computer-Aided Design* 45, 2 (2013), 204–228. https://doi.org/10.1016/j.cad.2012.08.006 Solid and Physical Modeling 2012.

[11] Preetha Chatterjee, Kostadin Damevski, and Lori Pollock. 2021. Automatic Extraction of Opinion-Based Q amp;A from Online Developer Chats. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1260–1272. https://doi.org/10.1109/ICSE43902.2021.00115

[12] Thomas R. Gruber and Daniel M. Russell. 1991. Design Knowledge and Design Rationale: A Framework for Representation, Capture, and Use. https://api.semanticscholar.org/CorpusID:18436716

[13] Soneya Binta Hossain, Nan Jiang, Qiang Zhou, Xiaopeng Li, Wen-Hao Chiang, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. A Deep Dive into Large Language Models for Automated Bug Localization and Repair. *arXiv preprint arXiv:2404.11595* (2024).

[14] Kai Huang, Zhengzi Xu, Su Yang, Hongyu Sun, Xuejun Li, Zheng Yan, and Yuqing Zhang. 2023. A Survey on Automated Program Repair Techniques. arXiv:2303.18184 [cs.SE]

[15] Anton Jansen, Jan Bosch, and Paris Avgeriou. 2008. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software* 81, 4 (2008), 536 – 557. https://doi.org/10.1016/j.jss.2007.08.025 Selected papers from the 10th Conference on Software Maintenance and Reengineering (CSMR 2006).

[16] Ziwei Ji, Tiezheng Yu, Yan Xu, Nayeon Lee, Etsuko Ishii, and Pascale Fung. 2023. Towards Mitigating Hallucination in Large Language Models via Self-Reflection. *arXiv preprint arXiv:2310.06271* (2023).

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. https://doi.org/10.1145/3213846.3213871

[18] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.

[19] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL]

[20] René Just, Darioush Jalali, and Michael D. Ernst. 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* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[21] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (ICSE)*. IEEE, 802–811.

[22] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, Xin Zhou, Enzhi Wang, and Xiaohang Dong. 2024. Better Zero-Shot Reasoning with Role-Play Prompting. arXiv:2308.07702 [cs.CL]

[23] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug report driven program repair. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 314–325.

[24] Rrezarta Krasniqi. 2021. Recommending Bug-fixing Comments from Issue Tracking Discussions in Support of Bug Repair. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 812–823. https://doi.org/10.1109/COMPSAC51774.2021.00114

[25] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[26] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[27] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023.

[28] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 602–614.

[29] Yan Liang, Ying Liu, Chun Kit Kwong, and Wing Bun Lee. 2012. Learning the "Whys": Discovering Design Rationale Using Text Mining - An Algorithm Perspective. *Comput. Aided Des.* 44, 10 (Oct. 2012), 916–930. https://doi.org/10.1016/j.cad.2011.08.002

[30] K. Liu, A. Koyuncu, and D. et al. Kim. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42.

[31] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 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*. 615–627.

[32] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. arXiv:2307.03172 [cs.CL]

[33] Claudia López, Víctor Codocedo, Hernán Astudillo, and Luiz Marcio Cysneiros. 2012. Bridging the Gap Between Software Architecture Rationale Formalisms and Actual Architecture Documents: An Ontology-driven Approach. *Sci. Comput. Program.* 77, 1 (Jan. 2012), 66–80. https://doi.org/10.1016/j.scico.2010.06.009

[34] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).

[35] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*. 101–114.

[36] Gastón Márquez, Mónica M Villegas, and Hernán Astudillo. 2018. An empirical study of scalability frameworks in open source microservices-based systems. In *2018 37th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, 1–8.

[37] Manish Motwani and Yuriy Brun. 2023. Better automatic program repair by using bug reports and tests together. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1225–1237.

[38] G.C. Murphy, D. Notkin, and K.J. Sullivan. 2001. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering* 27, 4 (2001), 364–380. https://doi.org/10.1109/32.917525

[39] Kunihiro Noda, Yusuke Nemoto, Keisuke Hotta, Hideo Tanida, and Shinji Kikuchi. 2020. Experience Report: How Effective is Automated Program Repair for Industrial Software?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–616. https://doi.org/10.1109/SANER48275.2020.9054829

[40] Sheena Panthaplackel, Milos Gligoric, Junyi Jessy Li, and Raymond J. Mooney. 2022. Using Developer Discussions to Guide Fixing Bugs in Software. In *Findings of the Association for Computational Linguistics: EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7-11, 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, 2292–2301. https://doi.org/10.18653/V1/2022.FINDINGS-EMNLP.169

[41] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Ray Mooney. 2022. Learning to Describe Solutions for Bug Reports Based on Developer Discussions. In *Findings of the Association for Computational Linguistics: ACL 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 2935–2952. https://doi.org/10.18653/v1/2022.findings-acl.231

[42] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.

[43] Ammar Rashid, William YC Wang, and Dan Dorner. 2009. Gauging the differences between expectation and systems support: the managerial approach of adaptive and perfective software maintenance. In *2009 Fourth International Conference on Cooperation and Promotion of Information Resources in Science and Technology*. IEEE, 45–50.

[44] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]

[45] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[46] Pankajeshwara Nand Sharma, Bastin Tony Roy Savarimuthu, and Nigel Stanger. 2021. Extracting Rationale for Open Source Software Development Decisions - A Study of Python Email Archives. *CoRR* abs/2102.05232 (2021). arXiv:2102.05232 https://arxiv.org/abs/2102.05232

[47] Lin Shi, Xiao Chen, Ye Yang, Hanzhi Jiang, Ziyou Jiang, Nan Niu, and Qing Wang. 2021. A First Look at Developers' Live Chat on Gitter. *CoRR* abs/2107.05823

Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

(2021). arXiv:2107.05823 https://arxiv.org/abs/2107.05823

[48] L. Shi, Z. Jiang, Y. Yang, X. Chen, Y. Zhang, F. Mu, H. Jiang, and Q. Wang. 2021. ISPY: Automatic Issue-Solution Pair Extraction from Community Live Chats. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 142–154. https://doi.org/10.1109/ASE51524.2021.9678894

[49] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs.AI]

[50] André Silva, Sen Fang, and Martin Monperrus. [n. d.]. *RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair.* Technical Report 2312.15698. arXiv. http://arxiv.org/abs/2312.15698

[51] André Silva, Sen Fang, and Martin Monperrus. 2023. RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair. *arXiv preprint arXiv:2312.15698* (2023).

[52] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Los Alamitos, CA, USA, 130–140. https://doi.org/10.1109/SANER.2018.8330203

[53] Pranjal Srivastava, Pranav Bhatnagar, and Anurag Goel. 2022. Argument mining using bert and self-attention based embeddings. In *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*. IEEE, 1536–1540.

[54] Antony Tang, Yan Jin, and Jun Han. 2007. A Rationale-based Architecture Model for Design Traceability and Reasoning. *J. Syst. Softw.* 80, 6 (June 2007), 918–934. https://doi.org/10.1016/j.jss.2006.08.040

[55] Yuchen Tian, Weixiang Yan, Qian Yang, Qian Chen, Wen Wang, Ziyang Luo, and Lei Ma. 2024. CodeHalu: Code Hallucinations in LLMs Driven by Execution-based Verification. *arXiv preprint arXiv:2405.00253* (2024).

[56] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

[57] Giovanni Viviani, Michalis Famelis, Xin Xia, Calahan Janik-Jones, and Gail C. Murphy. 2021. Locating Latent Design Information in Developer Discussions: A Study on Pull Requests. *IEEE Transactions on Software Engineering* 47, 7 (2021), 1402–1413. https://doi.org/10.1109/TSE.2019.2924006

[58] Wei Wang and Janet E. Burge. 2010. Using Rationale to Support Pattern-based Architectural Design. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge* (Cape Town, South Africa) *(SHARK '10)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/1833335.1833336

[59] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[60] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL]

[61] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 479–490.

[62] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. The Plastic Surgery Hypothesis in the Era of Large Language Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 522–534.

[63] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[64] C. S. Xia and L. Zhang. 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*. 959–971.

[65] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *arXiv preprint arXiv:2304.00385* (2023).

[66] Rui Xie, Zhengran Zeng, Zhuohao Yu, Chang Gao, Shikun Zhang, and Wei Ye. 2024. CodeShell Technical Report. *arXiv preprint arXiv:2403.15747* (2024).

[67] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[68] Chun Kit Kwong Ying Liu, Yan Liang and Wing Bun Lee. 2010. A New Design Rationale Representation Model for Rationale Mining. *Journal of Computing and Information Science in Engineering* 10 (2010), 031009–1–0.1009–10. Issue 3.

[69] Yingzhong Zhang, Xiaofang Luo, Jian Li, and Jennifer J. Buis. 2013. A semantic representation model for design rationale of products. *Advanced Engineering Informatics* 27, 1 (2013), 13–26. https://doi.org/10.1016/j.aei.2012.10.005 Modeling, Extraction, and Transformation of Semantics in Computer Aided Engineering Systems.

[70] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE]

[71] Jiuang Zhao, Zitian Yang, Li Zhang, Xiaoli Lian, and Donghao Yang. 2024. A Novel Approach for Automated Design Information Mining from Issue Logs. arXiv:2405.19623 [cs.SE]

[72] Roshanak Zilouchian Moghaddam, Brian Bailey, and Wai-Tat Fu. 2012. Consensus building in open source user interface design discussions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Austin, Texas, USA) *(CHI '12)*. Association for Computing Machinery, New York, NY, USA, 1491–1500. https://doi.org/10.1145/2207676.2208611