CHALMERS
UNIVERSITY OF TECHNOLOGY

An empirical evaluation of pre-trained large language models for repairing declarative formal specifications

Empirical Software Engineering (2025) 30:149
https://doi.org/10.1007/s10664-025-10687-1

An empirical evaluation of pre-trained large language models for repairing declarative formal specifications

Mohannad Alhanahnah[1] · **Md Rashedul Hasan**[2] · **Lisong Xu**[2] · **Hamid Bagheri**[2]

Accepted: 10 June 2025

Abstract
Automatic Program Repair (APR) has garnered significant attention as a practical research domain focused on automatically fixing bugs in programs. While existing APR techniques primarily target imperative programming languages like C and Java, there is a growing need for effective solutions applicable to declarative software specification languages. This paper systematically investigates the capacity of Large Language Models (LLMs) to repair declarative specifications in Alloy, a declarative formal language used for software specification. We designed six different repair settings, encompassing single-agent and dual-agent paradigms, utilizing various LLMs. These configurations also incorporate different levels of feedback, including an auto-prompting mechanism for generating prompts autonomously using LLMs. Our study reveals that dual-agent with auto-prompting setup outperforms the other settings, albeit with a marginal increase in the number of iterations and token usage. This dual-agent setup demonstrated superior effectiveness compared to state-of-the-art Alloy APR techniques when evaluated on a comprehensive set of benchmarks. This work is the first to empirically evaluate LLM capabilities to repair declarative specifications, while taking into account recent trending LLM concepts such as LLM-based agents, feedback, auto-prompting, and tools, thus paving the way for future agent-based techniques in software engineering.

Keywords Declarative specification · Automatic program repair · Formal methods · Alloy language · LLMs

1 Introduction

Declarative specification languages have become instrumental in addressing a multitude of software engineering challenges. Among these languages, the Alloy specification language (Jackson 2006) has emerged as a powerful tool, leveraging relational algebra and first-order logic to tackle a diverse array of tasks within the software engineering domain.

Communicated by: Hadi Hemmati.

Extended author information available on the last page of the article

Published online: 25 July 2025

Its application spans across software verification (Bagheri et al. 2020), the security analysis of cutting-edge platforms such as the IoT and Android systems (Bagheri et al. 2016, 2021; Alhanahnah et al. 2020), and the creation of test cases (Khurshid and Marinov 2004; Mirzaei et al. 2016). Integrating Alloy with the Alloy Analyzer has facilitated automated property verification, simplifying the process of checking whether specifications adhere to desired properties, and seamlessly integrating it into the Alloy environment. Despite these advancements, Alloy users, similar to developers in imperative languages, encounter challenges in debugging and correcting subtle bugs that may arise during specification writing, particularly for complex systems.

While the Alloy Analyzer aids in automatic property verification and counterexample generation, debugging and rectifying issues in Alloy specifications remain laborious and manual tasks. Unlike the rich literature and techniques available for automatic program repair (APR) in imperative languages, the landscape for APR in declarative languages is relatively sparse. Early approaches like ARepair (Wang et al. 2019) rely mainly on failing test cases to identify and rectify bugs, assuming the presence of tests for verification. However, this paradigm does not align well with Alloy's assertion-based specification approach, where users articulate expectations using assertions rather than tests. Existing APR techniques like ARepair (Wang et al. 2019) may succumb to overfitting issues when the test coverage is insufficient, compromising the correctness of generated repairs. Despite efforts to mitigate these challenges with techniques like BeAFix (Brida et al. 2021), which leverages assertions as correctness oracles, there is room for improvement in terms of efficiency and effectiveness.

The emergence of Large Language Models (LLMs) has revolutionized various domains, including natural language processing and code generation. These pre-trained models, such as GPT-3.5-Turbo and GPT-4, exhibit remarkable accuracy in predicting natural language and generating code. Inspired by these advancements and the persistent challenges in repairing formal specifications, we embark on a systematic empirical study to explore the potential of using LLMs to repair faulty alloy specifications. Our research aims to investigate key questions regarding the effectiveness, performance, adaptive prompting, failure characteristics, and repair costs associated with the use of LLM in Alloy specification repair. This investigation leverages recent advancements, including LLM-agents, tools, and feedback mechanisms.

This paper presents a systematic exploration and results of our comprehensive evaluation of the efficacy of LLMs in repairing faulty Alloy specifications, compared to state-of-the-art Alloy APR techniques. We design a repair pipeline working iteratively and integrating a dual-agent LLM framework comprising a Repair Agent and an Instructor Agent [1]. This iterative strategy centralizes around repairing bugs in defective specifications and generating specialized prompts to guide the repair process.

We conduct an extensive evaluation of LLM effectiveness in repairing defective Alloy specifications, comparing it against several state-of-the-art Alloy APR techniques, including ARepair (Wang et al. 2019), ICEBAR (Gutiérrez Brida et al. 2023), BeAFix (Brida et al. 2021), and ATR (Zheng et al. 2022), shedding light on their capabilities, benefits, and potential limitations. Our evaluation encompassed two comprehensive sets of benchmarks, comprising 1,974 defective specifications sourced from ARepair (Wang et al. 2019) and Alloy4Fun (Macedo et al. 2021), developed by external research groups. For each defective

[1] Our implementation and artifacts are available at: https://github.com/Mohannadcse/AlloySpecRepair

specification, we assess the repair pipeline across six distinct settings, conducting up to six iterations per setting.

Our findings contribute to advancing the field of automatic repair for declarative specifications and provide insights into leveraging LLMs for effective Alloy specification repair.

2 Background and Motivation

This section provides an overview of multi-agent LLMs followed by an illustrative example of a faulty Alloy specification to motivate the research and help the reader follow the discussions that ensue.

2.1 Multi-Agent Large Language Models

2.1.1 Large Language Models

Emerging pre-trained LLMs have demonstrated impressive performance on natural language tasks, such as text generation (Brown et al. 2020; Chowdhery et al. 2022; Vaswani et al. 2017) and conversations (Thoppilan et al. 2022; OpenAI 2023a). LLMs have also been proven effective in translating natural language specifications and instructions into executable code (Fan et al. 2023; Jain et al. 2022; Che 2021). These models have been trained on extensive corpora and possess the ability to execute specific tasks without the need for additional training or hard coding (Bubeck et al. 2023). They are invoked and controlled simply by providing a natural language prompt. The degree to which LLMs understand tasks depends largely on the prompting techniques used to convey user-provided instructions. These prompts are categorized into two frameworks: "zero-shot" learning and "few-shot" learning (Brown et al. 2020). Within the "zero-shot" learning framework, the prompt includes a description of the problem and instructions for its resolution, aiming that LLMs can tackle previously unseen tasks. Conversely, in the context of "few-shot" learning, LLMs are provided with examples, supplementing the guidance offered in the "zero-shot" prompt.

2.1.2 Agentic LLMs

With recent advancements in LLMs, developers are embracing the idea of creating autonomous agents that can solve various tasks and interact with environments, humans, and other agents using natural language interfaces (Zhou et al. 2023a). These agents provide several features including planning, memory, tool usage, and multi-agent communication. Therefore, LLM-based autonomous agents have gained tremendous interest in industry and academia (Wang et al. 2023). Several frameworks have been developed to support harnessing multi-agent LLM applications. AGENTS (Zhou et al. 2023b) is a unified framework and open-source library for language agents. AGENTS aims to facilitate developers to build applications with language agents. AutoGPT (AutoGPT 2024) a multi-agent framework for LLMs, is designed to support autonomous applications of LLMs. LangChain (Chase 2022) supports end-to-end language agents that can automatically solve tasks specified in natural language. It also facilitates the connection between LLM agents and external tools. Langroid (Langroid 2024) is another framework that supports the development of multi-

agent LLMs. It offers the capability to manage the history of messages, thus controlling the context window of LLMs. This is a crucial functionality for LLM apps operating iteratively (i.e., Self-Refine Madaan et al. 2023 and Reflexion Shinn et al. 2024). Moreover, Langroid enables seamless integration with a variety of LLMs.

2.1.3 Tools and Function Calling

Multi-agent frameworks and recent versions of OpenAI's LLM have introduced a feature known as function calling[2]. This feature enables users to provide function descriptions to the LLM. In turn, the LLM responds with a structured response (i.e., JSON data containing the requisite arguments for invoking any available functions). These functions serve as action executors and provide the option to supply APIs that the LLM can query to obtain essential information for responding to users. For example, when a user inquires about the current weather in a specific city and provides the LLM with a weather API call description, the LLM can augment its response with information retrieved from the API. This facilitates a seamless and efficient interaction between the user and the LLM and provides rich responses to avoid hallucinations.

2.2 Alloy - Illustrative Example

Alloy, a formal modeling language, provides a comprehensible syntax inspired by object-oriented notations and is grounded on first-order relational logic (Jackson 2006). Within an Alloy specification, three primary components shape its structure: data types, constraints expressed through formulas, and commands to initiate the analyzer.

The language uses signature (sig) definitions to define sets of elements, with fields specifying relationships between these sets. Additionally, Alloy employs fact to introduce constraints that hold in every instance of the specification. These constraints restrict the model space, ensuring its consistency. Further structuring of formulas is achieved through pred and fun, which are named parameterized Alloy expression, and assert encapsulates the properties intended for verification.

Commands such as check and run activate the automated analyzer. Check verifies assertions, while run executes predicates, aiming to identify model instances that satisfy specified conditions. Alloy's expressiveness stems from its use of relational logic, a first-order logic extended with relational operators. These include all, some, one, and lone quantifiers, along with operators like relational join and transitive closure.

To illustrate the Alloy language and provide motivation for this research, we examine the specification shown in Listing 1 from the ARepair benchmark (Wang et al. 2019). This model represents a university context with students, professors, classes, and assignments. The specification defines relationships such as teaching assistants (students assigned as assistants), instructors (professors), and the association of assignments with both classes and students.

The predicate `PolicyAllowsGrading` determines who can grade assignments, allowing only instructors or teaching assistants (TAs) of a class. However, a bug in the policy allows a student to grade their own assignment if they are TA for the same class. The fix is explicitly preventing students from grading their own assignments by adding the

[2] https://platform.openai.com/docs/guides/text-generation/function-calling

condition (at line 15). This condition ensures a person "s" who qualifies as a grader (either as a TA or an instructor) must not be among the students assigned that specific assignment.

The assertion `repair_assert_1` and predicate `repair_pred_1` at the end formalize and check the requirement that no individual is allowed to grade an assignment assigned to themselves.

Reasoning about and addressing this defect through direct prompting of LLMs can be challenging, as demonstrated in Hasan et al. (2023). Additionally, conventional state-of-the-art Alloy repair techniques, such as ARepair (Wang et al. 2019) and BeAFix (Brida et al. 2021), also struggled to effectively address this defect. Given recent advances in LLM agents and prompt engineering techniques, this study aims to evaluate the capabilities of LLMs in light of these developments to repair such cases and evaluate their effectiveness in contrast to state-of-the-art Alloy repair techniques. In the following sections, we explore the ability of LLMs to address such challenges and the specific conditions under which they can do so[3].

```
1  abstract sig Person {}
2  sig Student extends Person {}
3  sig Professor extends Person {}
4  sig Class {
5    assistant_for: set Student,  // as in : "is TA for"
6    instructor_of: one Professor
7  }
8  sig Assignment {
9    associated_with: one Class,
10   assigned_to: some Student
11 }
12
13 pred PolicyAllowsGrading(s: Person, a: Assignment) {
14   s in a.associated_with.assistant_for or s in a.
       associated_with.instructor_of
15   // Fix: add "s !in a.assigned_to".
16 }
17
18 assert repair_assert_1 {
19   all s : Person | all a: Assignment | PolicyAllowsGrading[s
       , a] implies not s in a.assigned_to
20 }
21
22 check repair_assert_1
23
24 pred repair_pred_1 {
25   all s : Person | all a: Assignment | PolicyAllowsGrading[s
       , a] implies not s in a.assigned_to
26 }
27
28 run repair_pred_1
```

Listing 1 A snippet of the flawed `grade` Alloy specification sourced from the ARepair benchmark[4].

[3] https://github.com/guolong-zheng/atmprep/blob/master/benchmark/arepair/grade1.als

2.3 Underconstrained and Overconstrained Specifications

2.3.1 Underconstrained Specifications

Underconstrained specifications arise when essential constraints are omitted, permitting behaviors that deviate from the intended semantics. This class of defects often results in models that allow for unintended or invalid configurations. As illustrated in Listing 2, the specification enforces that each `Tree` must have at least one `root`, but does not restrict the number of root nodes. Consequently, multiple nodes could simultaneously serve as roots for a single tree, contradicting the common assumption of a unique root in tree structures. Repairs addressing this issue typically involve replacing `some` with `one`, or introducing explicit uniqueness constraints to ensure the intended behavior.

```
sig Tree {
root: set Node
}
sig Node {
children: set Node
}
fact WellFormed {
all t: Tree | some t.root
}
```

Listing 2 Example of an underconstrained specification that permits multiple roots.

2.3.2 Overconstrained Specifications

In contrast, overconstrained specifications impose overly restrictive conditions, unintentionally excluding valid model instances. This type of defect typically results in models that are too limited, preventing the representation of legitimate scenarios. As shown in Listing 3, while the first constraint correctly enforces a unique root, the second constraint forbids any `Node` from having children–effectively eliminating all hierarchical structures. As a result, the specification precludes valid tree configurations. Repairs in such cases often involve relaxing the restrictive constraint, such as substituting `no` with `some`, to reintroduce permissible structure while preserving correctness.

```
fact WellFormed {
all t: Tree | one t.root
all n: Node | no n.children
}
```

Listing 3 Example of an overconstrained specification that disallows any node hierarchy.

3 Methodology

This section outlines the repair pipeline, detailing its architecture and workflow. The pipeline is derived from the APR phases described in Zhang et al. (2023), including localization, repair, and verification. The LLM is responsible for the localization and repair steps, while the verification step is facilitated by granting the LLM access to the Alloy analyzer. Conse-

quently, the LLM can autonomously execute all APR phases to repair the defective model. Our implementation of this pipeline utilizes the Langroid framework (Langroid 2024).

3.1 Overall Workflow

This section presents the design of the repair pipeline specifically designed for iterative self-refinement,

wherein the repair process for a defective Alloy model operates within a predetermined budget, defined by the number of iterations allocated. Previous research has indicated that this iterative prompting method produces superior results compared to traditional single-step approaches, improving task efficiency by an average of 20% (Madaan et al. 2023). Should the repair process exceed this limit without success, the pipeline halts further attempts. After each unsuccessful repair iteration, feedback is collected and used to update the prompt, refining the initial draft generated by the LLM.

The workflow of our APR pipeline, as depicted in Fig. 1, comprises two primary components: the Repair Agent and the Instructor Agent. Each agent maintains its context, tools, and prompts, but they collaborate to adaptively refine the Repair Agent prompt for enhanced effectiveness, as elaborated below. Our pipeline supports various feedback levels, employing a zero-shot prompting approach (Paul et al. 2023), where tasks are performed without explicit examples, as detailed in Table 1.

Upon receiving defective Alloy specifications and the prompt, our APR pipeline dispatches these elements to the LLM, which generates a patched version aimed at rectifying the defect. Subsequently, the Repair Agent compares the proposed model with the buggy ones to evaluate the ability of LLM to locate faults and adhere to instructions. The accuracy of the proposed model is then assessed by triggering the Alloy Analyzer tool, which runs the proposed model for validation. If the Alloy analyzer confirms the bug's resolution, the

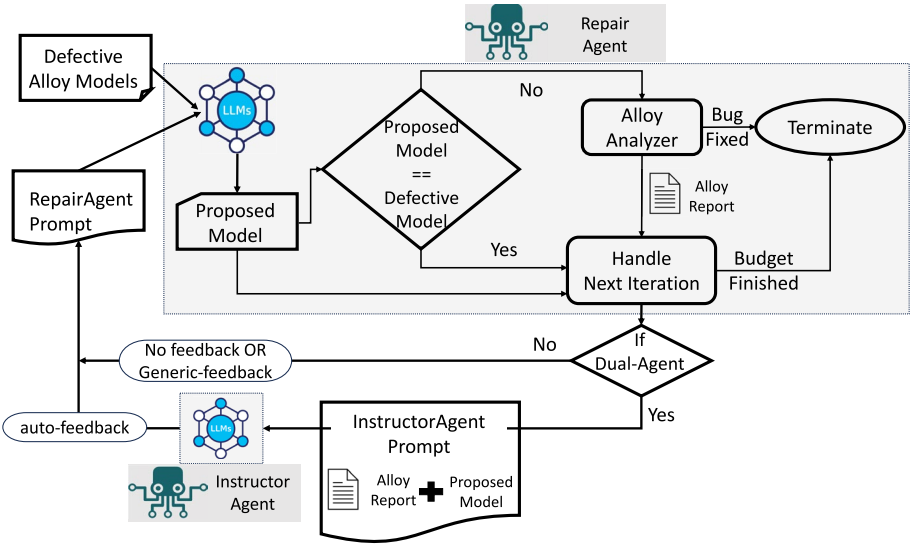


Fig. 1 APR Pipeline for Alloy specifications. It supports Single-Agent (Repair Agent Only) and Dual-Agent (Repair Agent and Instructor Agent) paradigms

Table 1 Zero-shot prompts used by the repair pipeline

Agent	Message Type	Message content
Repair Agent	Agent-Instruction	You are an expert in repairing Alloy declarative specifications. You will be presented with Alloy <Faulty_SPECIFICATIONS>. Your task is to FIX/REPAIR the <Faulty_SPECIFICATIONS>. Use the tool 'run_alloy_analyzer' to demonstrate and validate the <FIXED_SPECIFICATIONS>. Wait for my feedback, which may include error messages or Alloy solver results. You will have 5 trials to fix the <Faulty_SPECIFICATIONS>. **Adhere to the Following Rules**: - The <FIXED_SPECIFICATIONS> should be consistent (having instances) and all the assertions should be valid (no counterexample). - DO NOT REPEAT the <FIXED_SPECIFICATIONS> that I sent you. - DO NOT provide any commentary and always send me anything ONLY using the tool 'run_alloy_analyzer'. - The <FIXED_SPECIFICATIONS> MUST be different than the <Faulty_SPECIFICATIONS>.
	Tool-instruction	ALL AVAILABLE TOOLS and THEIR JSON FORMAT INSTRUCTIONS: You have access to the following TOOLS to accomplish your task: TOOL: run_alloy_analyzer, PURPOSE: To show a <FIXED_SPECIFICATIONS> to the user. Use this tool whenever you want to SHOW or VALIDATE the <FIXED_SPECIFICATIONS>. NEVER list out a <FIXED_SPECIFICATIONS> without using this tool. JSON FORMAT: "type": "object", "properties": "request": "default": "run_alloy_analyzer", "type": "string","specification": "type": "string", "required": ["specification", "request"]. When one of the above TOOLs is applicable, you must express your request as "TOOL:" followed by the request in the above JSON format.
Instructor Agent	Agent-Instruction	You are Expert in Analyzing Alloy Analyzer reports. Can you describe concisely and precisely the modifications needed to fix the error in at most 2 sentences? Based on this report from Alloy Analyzer: {Alloy_report_msg} After running this Alloy Model is: {proposed_spec}

repair process concludes. Conversely, if errors persist, the pipeline initiates another iteration, depending on the remaining repair budget. If the budget is depleted, the pipeline terminates operations. The adaptive prompt is designed to evolve and refine based on different settings (detailed in Section 3.3), ensuring continuous improvement after each unsuccessful attempt. The following sections provide a comprehensive overview of each component of the repair pipeline.

3.2 Pipeline Agents

As depicted in Fig. 1, the pipeline comprises two agents. This design facilitates evaluating the recent deployment of LLM apps as multi-agent applications.

3.2.1 Repair Agent

This agent is the core component of the APR pipeline. It has access to tools/functions (i.e., "run_alloy_analyzer"), maintains its context (i.e., history of messages) over the iterations, and decides the process for handling subsequent iterations.

The agent's prompt encompasses a set of messages combined as a system message. These messages are listed in Table 1.

- **Agent Instruction.** This message provides a general guideline to the *Repair Agent*, commencing with the agent's persona as an "expert in repairing Alloy specifications". It proceeds to notify LLM of defects within the provided Alloy model, without disclosing the defects' locations. The message finally describes the tools (i.e., "run_alloy_analyzer") to which this agent has access. In conclusion, it enumerates a set of procedural steps to be executed by the LLM.
- **Tool Instruction.** Offers instructions delineate the conditions and procedures for utilizing the tool "run_alloy_analyzer". It further specifies the data required from the LLM to ensure the proper activation of this tool. This instruction guides the LLM to transmit a JSON formatted response, encapsulating two fields: (1) `request`, denoting the tool designation intended for use by the LLM, and (2) `proposed_specification`, representing the LLM's suggested version for rectifying the identified bug.

3.2.2 Instructor Agent

This agent has its LLM settings and context and generates feedback based on the report generated by Alloy Analyzer and the proposed specification. This agent leverages the Auto-Prompt concept (Shin et al. 2020) by automatically constructing a prompt using LLMs. The produced feedback (i.e., the automatically constructed prompt) guides the *Repair Agent* on the methods for rectifying the bug. The system message that provides the instructions to this agent is described in Table 1. Unlike the system message of the *Repair Agent*, the agent's persona is described as "Expert in Analyzing Alloy Analyzer reports". This agent does not share the same context or prompts with the *Repair Agent*. But the response of the *Instructor Agent* is appended to the prompt that will be used in the next iteration by the *Repair Agent*, as illustrated in Fig. 1.

3.3 Feedback Messages

The initial prompt contains only *Agent Instruction* and *Tool Instruction* defined by the `Repair Agent`, as described in Table 1.

The pipeline refines the prompts in response to the behavior of the proposed specification. This refinement process is contingent upon repetitive buggy specifications, errors that emerge after executing these specifications through the Alloy analyzer, or LLM failure to send a response based on the required JSON format.

To this end, Table 2 presents the feedback messages that are supported by the pipeline. These feedback messages will be appended to the initial prompt. Following is a description of these:

- **Tool fallback.** This message informs LLM that the response received does not comply with the tool's JSON format, impeding the proposed specification's extraction. To address this issue, we developed a parser atop the Langroid JSON parser to retrieve Alloy specifications from the response. Nonetheless, the parser cannot handle all scenarios, resulting in the dispatch of this feedback to the LLM, soliciting adherence to the mandated JSON format.

Table 2 Feedback Messages used by the repair pipeline

Feedback Type	Generated By	Message Content
Tool-fallback	Repair Agent	You must use the CORRECT format described in the tool 'run_alloy_analyzer' to send me the fixed specifications. You either forgot to use it, or you used it with the WRONG format. Make sure all fields are filled out.
Repeated spec		The proposed <FIXED_SPECIFICATIONS> is IDENTICAL to the Alloy <Faulty_SPECIFICATIONS> that I sent you. **DO NOT** send Alloy specifications that I sent you again. ALWAYS USE the tool 'run_alloy_analyzer' to send me a new <FIXED_SPECIFICATIONS>.
No-feedback		The proposed specification DID NOT fix the bug.
Generic-feedback		Below are the results from the Alloy Analyzer. Fix all Errors and Counterexamples before sending me the next <FIXED_SPECIFICATIONS>.
Auto-feedback	Instructor Agent	The message content will be dynamically constructed by the agent based on the supplied information.

- **Repeated Spec.** This message is triggered when the proposed spec is the same as the buggy specification. In this situation, this message tells the LLM not to repeat the buggy specification.
- **Alloy Analyzer Report.** This message relays the feedback of the verification process using Alloy Analyzer. Table 2 presents three feedback levels.

In the following, we discuss the details of the various feedback levels supported by the pipeline.

4 Alloy Analyzer Feedback Levels

The verification stage involves running the proposed specifications with the Alloy Analyzer. Subsequently, feedback conveying the Alloy Analyzer's report is generated, available at different levels of detail. This process aids in assessing the effectiveness of various refinements made to the prompts.

In this study, we consider three feedback levels to refine the prompt as outlined in Table 2. These levels reflect errors, counterexamples, and instances identified after running the Alloy Analyzer in each iteration. They also mimic different repair scenarios, as described below:

1. *No-feedback*. In this setting, the Alloy Analyzer agent returns only a single response (i.e., `The proposed specification DID NOT fix the bug.`) to LLM without describing the details of the report generated by the Alloy Analyzer. This setting is

designed to demonstrate the ability of LLMs to identify and fix errors in a faulty model solely based on the content of the buggy model itself, mirroring the function of a programming language compiler.
2. *Generic-feedback*. We use a template to send the Alloy Analyzer report to LLM. This report summarizes counterexamples, instances, and errors. This scenario represents a common situation where the developer shares a summary of the issue on a question-and-answer platform (such as Stack Overflow) to get help.
3. *Auto-feedback*. When this setting is enabled, the feedback response to LLM will be generated by another LLM agent (i.e., `Instructor agent`), which represents the dual-agent repair pipeline. We forward the report generated by the Alloy Analyzer and the proposed Alloy specifications to the prompt agent, which generates a prompt to instruct the `Repair Agent`.

Noteworthy, the *No-feedback* and *Generic-feedback* mechanisms are used under the single-agent paradigm, whereas the *Auto-feedback* approach enables the dual-agent paradigm. The different reports reflect the ability of LLMs to (1) locate bugs and (2) correctly perform the repair process. For example, *Generic-feedback* and *Auto-feedback* show the ability of LLM to perform its reasoning based on a human-created prompt versus an LLM-generated prompt, which has no access to the repair context. Also, the latter reflects a real scenario wherein the user cannot repair the buggy specification, then they consult experts or other forums to get assistance.

5 Experiment Design

In this study, we address the following research questions (RQs):

- **RQ1 (Effectiveness):** How effective is the APR pipeline in repairing compared to the state-of-the-art and how various repair settings contribute to the effectiveness?
- **RQ2 (Performance):** What is the repair performance when employing pre-trained LLMs?
- **RQ3 (Failure Characteristics):** What are the characteristics of failures encountered during the repair process?
- **RQ4 (Repair Costs):** What is the cost associated with using the APR pipeline with various LLMs?

5.1 Dataset

In our evaluation, we utilize two distinct benchmark suites: ARepair (Wang et al. 2019) and Alloy4Fun (Macedo et al. 2021). These benchmark suites have undergone extensive study and developed by independent research groups, enabling a fair comparison of various techniques. The benchmark datasets consist of specifications varying from tens to hundreds of lines, featuring real bugs authored by humans. The defects within the benchmarks span a diverse range, from straightforward issues that can be addressed by modifying a single operator to complex challenges necessitating the synthesis of novel expressions and the substitution of entire predicate bodies.

The ARepair benchmark (Wang et al. 2019) encompasses 38 faulty specifications extracted from a total of 12 Alloy problems. Within this benchmark, both single and multi-line errors are present, distributed across 28 faulty models with single-line errors and 10 faulty models with multi-line errors. The Alloy4Fun benchmark (Macedo et al. 2021) comprises a collection of 1,938 handwritten defective models sourced from student submissions across six different Alloy problems. All bugs within this benchmark are single-line bugs. Both benchmarks are accompanied by correct versions of the specifications, serving as ground truths for verifying the accuracy of the obtained results.

5.2 Pre-processing Benchmark Dataset

As mentioned earlier, the benchmark datasets include comments indicating the locations of the bugs and their corresponding fixes. To ensure a fair evaluation, we perform two steps: (1) determining the uniqueness of Alloy specifications and (2) removing fix comments from the defective models.

Uniqueness of Alloy Specifications To ensure that all specifications in the ARepair and Alloy4Fun benchmarks are unique, we applied a systematic normalization and hashing process. Each specification was processed in its entirety, preserving the original content. We removed block comments, and collapsed all whitespace (spaces, tabs, line breaks) to a single space. This normalization step eliminates superficial differences such as formatting and comments, allowing us to focus on the core logic of each Alloy specification. After normalization, we computed an MD5 hash for each specification, treating the hash as a unique fingerprint. Specifications with identical hashes were considered duplicates. We grouped specifications by hash and maintained counts for each benchmark and for the combined dataset. Our analysis found that all 38 ARepair specifications and all 1936 Alloy4Fun files were unique–no duplicates were detected in the combined set of 1974 specifications. This confirms the diversity and uniqueness of the Alloy specifications in both datasets.

Removal of Fix comments We have removed these comments from the defective models. Specifically, we have eliminated all lines starting with the phrase "*Fix:*" in the defective models. This line typically outlines the necessary changes required to resolve the bug. Models with multiple "*Fix:*" comments are categorized as having multi-line bugs, whereas those with only one "*Fix:*" comment are considered to have single-line bugs. Removing comments is essential to prevent LLMs from receiving explicit clues about bug locations and fixes. This ensures an accurate assessment of their repair capabilities and effectiveness in localizing bugs within defective models. Importantly, this reflects a realistic scenario in which the user employs LLMs without prior knowledge of the bug location and the fix.

5.3 Implementation

Our implementation of the APR pipeline was realized using Python 3.11, with the assistance of Langroid to facilitate the integration of multi-agent LLMs. This implementation encompasses several crucial functionalities. Firstly, it offers support for various LLMs, including local LLMs, ensuring flexibility in model selection. Secondly, it incorporates a message history control mechanism, essential for preventing context-length limitations, particularly

during iterative repair processes. Lastly, our implementation facilitates the creation and customization of tools, allowing for the definition of response formats and fields. Additionally, we developed a specialized parser to address issues caused by special situations that could impede the extraction of proposed specifications.

The experiments were carried out on a system equipped with a 2.3 GHz Quad-Core Intel Core i7 processor, 32 GB of RAM, and running macOS Sonoma. Furthermore, the system was configured with Oracle Java SE Development Kit 8u202 (64-bit version), ensuring compatibility and optimal performance throughout the execution of the APR pipeline.

5.4 Subject LLMs

To assess the performance of existing pre-trained LLMs, we have carefully chosen three representative models. We explored local LLMs such as Llama-2 and Mistral, but they lacked support for tools and JSON responses, making it challenging to process their responses, which often contained incomplete Alloy specifications. Following is a description of the selected models, which are summarized in Table 3:

- **GPT-3.5-Turbo.** This standard LLM, provided by OpenAI and utilized in ChatGPT, boasts a sophisticated architecture with 175 billion parameters, endowing it with extensive capabilities. Engineered to tackle a wide array of natural language processing (NLP) tasks, including text generation and completion, GPT-3.5-Turbo epitomizes advanced computational linguistics.
- **GPT-4-32k.** This model contains over 1.76 trillion parameters, demonstrating significantly enhanced capabilities compared to GPT-3.5. For our study, we leverage the GPT-4-32k-0613 model version.
- **GPT-4-Turbo.** Another iteration of the GPT-4 model, GPT-4-Turbo features an updated knowledge cutoff as of April 2023 and introduces a 128k context window. Moreover, it offers cost-effectiveness compared to GPT-4, alongside notable enhancements such as improved instruction following, JSON mode, and reproducible outputs (OpenAI 2023b).
- **GPT-4o.** This model shares features with **GPT-4-Turbo** but is more cost-effective and has an updated knowledge base as of October 2023.

5.5 Subject SOTA Systems

We conducted a comparative analysis of repair performance against several state-of-the-art Alloy repair tools: ARepair (Wang et al. 2019), ICEBAR (Gutiérrez Brida et al. 2023), BeA-

Table 3 Characteristics of Pre-trained LLMs

Model	Version	Cut-off	Context Window (Tokens)	Input Cost per 1M tokens	Output Cost per 1M tokens
GPT-3.5 Turbo	1106	Sep 2021	16,385	$1	$2
GPT-4-32k	0613	Sep 2021	32,768	$60	$120
GPT-4 Turbo	1106-preview	Apr 2023	128k	$10	$30
GPT-4o	2024-05-01-preview	Oct 2023	128k	$5	$15

Fix (Brida et al. 2021), ATR (Zheng et al. 2022), and Hasan et al. (2023). Each tool employs a distinct development approach and tackles specification attributes differently.

ARepair (Wang et al. 2019) generates fixes for Alloy specifications that violate test cases. ICEBAR (Gutiérrez Brida et al. 2023) utilizes ARepair as a backend tool to repair faulty Alloy specifications based on a predefined set of Alloy tests. ATR (Zheng et al. 2022) adopts a template-based methodology to enhance the repair process, leveraging fault localization strategies to identify potentially erroneous Alloy expressions that lead to assertion failures. BeAFix (Brida et al. 2021) relies on user input to identify faulty statements. It exhaustively explores all possible repair candidates up to a certain threshold through mutation and employs Alloy counterexamples to evaluate the feasibility of generalization. Finally, Hasan et al. (2023) employed ChatGPT (GPT-3.5-Turbo) for repairing faulty specifications across five scenarios. This approach employs a purely LLM-based method, wherein a single repair iteration is conducted without the implementation of agent setup or feedback mechanisms. For our comparison, we exclude scenarios where ChatGPT is provided with "*Fix:*" comments, as this represents an unrealistic setting and does not align with our "Benchmark Preprocessing" step (cf. Section 5.2).

Since the state-of-the-art tools–ARepair, ICEBAR, BeAFix, and ATR–were evaluated on static Alloy models prior to the release of Alloy 6, our experimental analysis focused exclusively on static Alloy models to ensure a fair comparison. These tools do not support mutable specifications or recently introduced features such as linear temporal logic (LTL), making them incompatible with models that leverage Alloy 6's new capabilities. Additionally, due to structural dependencies and the absence of updated implementations, these tools could not be reliably executed on datasets incorporating Alloy 6-specific features. Nevertheless, our repair technique is not inherently limited to static specifications and, in principle, can be extended to support the dynamic capabilities introduced in Alloy 6.

5.6 Experimental Configuration and Settings

LLM Temperature To balance between deterministic progression and iterative refinement, we set the temperature parameter of the Large Language Models (LLMs) to 0.2, allowing for a moderate level of randomness while maintaining some level of determinism throughout the repair process (Kong et al. 2024; Zhang et al. 2024).

Number of Iterations Initially, we conducted a preliminary experiment using the ARepair benchmark framework with GPT-4-32k, allocating a budget of ten iterations. However, we observed diminishing returns after six iterations in most cases. Therefore, we opted for a six-iteration budget in the APR pipeline.

Metric for Comparing Repair Performance of LLMs We employ the *Correct@k* metric (Liu et al. 2024) to evaluate the success rate of the techniques in repairing defective Alloy specifications. This metric quantifies the number of defects successfully repaired within a maximum of k iterations, where k is set to 6 in our study (denoted as *Correct@6*). The formula for this metric is described in (1).

$$\text{Correct@}k = \left(\frac{\#\text{ of bugs successfully repaired within } k \text{ iterations}}{\text{Total number of bugs evaluated}} \right) \times 100 \qquad (1)$$

Experiment Settings and Baseline Our evaluation encompasses a total of 1,974 defective models. We assess three distinct LLMs, as outlined in Table 3. The APR pipeline supports three levels of report granularity and permits multiple iterations, resulting in a total of 106,596 repair attempts (calculated as the product of 1,974 benchmarks, three LLMs, three feedback levels, and up to six iterations). The Single-agent paradigm utilizes the No-feedback and Generic- feedback mechanisms, whereas the dual-agent paradigm employs the Auto-feedback approach within the repair pipeline.

The repair experiment on the Alloy4Fun benchmark leverages GPT-4o–the most cost-effective option among GPT-4-32k and GPT-4-Turbo–for a randomly sampled subset of 357 models, maximizing both efficiency and performance. For comprehensive coverage, GPT-3.5-Turbo is applied to the entire Alloy4Fun dataset, taking advantage of its substantially lower cost compared to the GPT-4 family. This experimental design enables a broad and cost-efficient evaluation across all models, while also allowing for targeted, high-performance analysis using GPT-4o on a representative subset. Table 4 presents the settings corresponding to different combinations of LLMs and feedback levels.

6 Experimental Results

This section summarizes the data we collected, its interpretation, and our results.

6.1 Results for RQ1: Effectiveness

This research question investigates the effectiveness of LLMs in repairing Alloy specifications under various settings within the repair pipeline. To address this, we (1) compare the repair pipeline with state-of-the-art Alloy repair tools and (2) assess the impact of different feedback levels, LLMs, and agent paradigms on repair performance, and study the influence of LLMs on the repaired models.

Comparing with SoTA Tables 5 and 6 present a comprehensive comparison of our approach against state-of-the-art Alloy repair techniques, including ARepair (Wang et al. 2019), ICEBAR (Gutiérrez Brida et al. 2023), BeAFix (Brida et al. 2021), ATR (Zheng et al. 2022),

Table 4 Settings corresponding to various combinations of LLMs and feedback levels

Setting No.	# Agents	LLM	Feedback Level
Setting-1	Single	GPT-4-32k	No-Feedback
Setting-2	Single	GPT-4-32k	Generic-Feedback
Setting-3	Dual	GPT-4-32k	Auto-Feedback
Setting-4	Single	GPT-4-Turbo	No-Feedback
Setting-5	Single	GPT-4-Turbo	Generic-Feedback
Setting-6	Dual	GPT-4-Turbo	Auto-Feedback
Setting-7	Single	GPT-4o	No-Feedback
Setting-8	Single	GPT-4o	Generic-Feedback
Setting-9	Dual	GPT-4o	Auto-Feedback
Setting-10	Single	GPT-3.5-Turbo	No-Feedback
Setting-11	Single	GPT-3.5-Turbo	Generic-Feedback
Setting-12	Dual	GPT-3.5-Turbo	Auto-Feedback

Table 5 Comparison of state-of-the-art Alloy repair techniques–ARepair (Wang et al. 2019), ICEBAR (Gutiérrez Brida et al. 2023), BeAFix (Brida et al. 2021), ATR (Zheng et al. 2022), and Hasan et al. (2023)–on the ARepair benchmark, using the settings detailed in Table 4

Model	Total	Defects	ARepair	ICEBAR	BeAFix	ATR	Hasan et al.	#repairs Under Setting											
	#specs	Count	#repairs	#repairs	#repairs	#repairs	#repairs	GPT-4-32k			GPT-4-Turbo			GPT-4o			GPT-3.5-Turbo		
								1	2	3	4	5	6	7	8	9	10	11	12
addr	1	1	1	1	1	1	0	0	0	1	0	1	1	0	0	0	0	0	0
arr	2	2	2	2	1	0	0	0	0	0	1	0	1	1	2	1	0	1	1
balancedBST	3	8	1	2	1	1	0	1	1	2	1	1	1	1	1	1	0	0	1
bempl	1	1	0	1	0	1	1	0	0	0	1	1	1	0	0	0	0	0	0
cd	2	3	0	2	2	2	0	1	2	2	2	1	2	2	2	2	1	0	0
ctree	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0
dll	4	8	0	3	3	2	0	4	4	4	2	2	4	4	4	4	1	3	3
farmer	1	2	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
fsm	2	2	2	2	1	2	0	1	1	2	0	0	1	1	1	2	0	0	0
grade	1	1	0	1	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0
other	1	1	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	0	1
student	19	34	2	7	13	10	2	6	6	9	8	11	13	12	14	14	0	1	11
Summary	**38**	**64**	**9**	**21**	**24**	**22**	**4**	**15**	**16**	**22**	**17**	**19**	**28**	**24**	**27**	**28**	**4**	**6**	**18**
%repair			**23.7**	**55.3**	**63.2**	**57.8**	**10.5**	**39.5**	**42.1**	**57.9**	**44.7**	**50**	**73.4**	**63.2**	**71.1**	**73.4**	**10.5**	**15.8**	**47.4**

and Hasan et al. (2023), on the ARepair and Alloy4Fun benchmarks, respectively. Each table reports the number of defects addressed and repairs achieved by each tool, as well as by various LLM settings described in Table 4. For the ARepair benchmark (Table 5), results are shown across all settings, enabling a detailed evaluation of repair effectiveness. For the Alloy4Fun benchmark (Table 6), GPT-4o is applied to a representative subset of models to balance performance and computational cost, while GPT-3.5-Turbo is evaluated on the full dataset.

The "Defects Count" column reports the total number of bugs identified within each model category, providing insight into the distribution of defects across specifications. When the value in "Defects Count" is equal to "Total #specs," it indicates that each specification contains exactly one defect, suggesting the presence of only single-line bugs. In contrast, if "Defects Count" exceeds "Total #specs," this reveals that some specifications contain multiple defects, indicative of multi-line bugs. For example, in Table 5, the `balancedBST` model exhibits 8 defects across 3 specifications, clearly demonstrating the occurrence of multi-line bugs.

The subsequent columns present the number of correctly repaired specifications for each state-of-the-art Alloy repair tool, with the remaining columns reporting the repair outcomes under the various settings described in Table 4.

The results highlight the superior performance of the GPT-4 family–particularly GPT-4o and GPT-4-Turbo–over both GPT-4-32k and traditional state-of-the-art Alloy repair tools. In contrast, GPT-3.5-Turbo (Settings 10–12) demonstrates significantly lower repair effectiveness than traditional tools, which aligns with previous findings by Hasan et al. (2023).

Furthermore, across all evaluated LLMs and benchmarks, the Auto-feedback configuration (Settings 3, 6, 9, and 12 in Tables 6 and 5) consistently delivers the strongest repair performance. Notably, the Generic-feedback configuration (Setting 8) achieves results comparable to Setting 9 on the Alloy4Fun sampled benchmark with GPT-4o, a phenomenon attributed to the inherent stochasticity of LLMs. To further investigate, we re-executed repair experiments on five models that were initially only repaired under Generic-feedback; two of these were successfully repaired upon rerunning with Auto-feedback. This outcome underscores the effectiveness of LLM-driven prompt construction over generic, human-crafted prompts and supports the adoption of multi-agent LLM configurations for automated specification repair.

The repair results for the ARepair benchmark, as presented in Table 5, show that Settings 6 and 9 deliver substantially superior performance compared to all SoTA tools, including traditional program repair approaches. Notably, the APR pipeline is able to repair specifications that remain unresolved by most, or even all, existing SoTA techniques. For instance, the `grade` specification (see Listing 1), which could not be repaired by the majority of SoTA tools, was successfully addressed under both Settings 6 and 9. Similarly, the `farmer` specification, which contains a multi-line bug and was not repaired by any SoTA tool, was successfully fixed by all APR configurations. The `ctree` specification, which was addressed by only a single SoTA tool, was also repaired by all APR configurations leveraging GPT-4 models.

These results demonstrate that models from the GPT-4 family are particularly effective at handling complex specifications, especially those involving multi-line bugs. In particular, the findings underscore the promising potential of agentic LLMs enhanced with feedback

Table 6 Comparison of state-of-the-art Alloy repair techniques–ARepair (Wang et al. 2019), ICEBAR (Gutiérrez Brida et al. 2023), BeAFix (Brida et al. 2021), ATR (Zheng et al. 2022), and Hasan et al. (2023)–on the Alloy4Fun benchmark, using the settings described in Table 4

Model	Total	Defects	ARepair	ICEBAR	BeAFix	ATR	Hasan et al.	#repairs Under Setting					
	#specs	Count	#specs	#repairs	#repairs	#repairs	#repairs	GPT-4o			GPT-3.5-Turbo		
								7	8	9	10	11	12
classroom	999(60)	999(60)	88(3)	424(18)	387(16)	688(29)	88(4)	(0)	(0)	(6)	0	1	21
cv	138(60)	138(60)	2(0)	86(26)	82(20)	38(0)	4(3)	(56)	(58)	(59)	38	32	70
graphs	283(60)	283(60)	19(8)	237(39)	232(38)	260(44)	20(39)	(22)	(40)	(22)	23	51	35
lts	249(60)	249(60)	1(0)	73(20)	41(9)	70(15)	21(32)	(38)	(48)	(46)	68	115	106
production	61(60)	61(60)	27(23)	36(33)	56(53)	43(46)	12(12)	(59)	(60)	(58)	25	32	34
trash	206(57)	206(57)	48(9)	195(46)	183(49)	187(34)	2(2)	(1)	(0)	(10)	0	0	12
Summary	**1936(357)**	**1936(357)**	**185(43)**	**1051(182)**	**981(185)**	**1286(168)**	**147(92)**	**(176)**	**(206)**	**(201)**	**154**	**231**	**278**
%repair			**9.5(12)**	**54.2(50.9)**	**50.6(51.8)**	**66.4(47)**	**7.6(25.7)**	**(49.2)**	**(55.7)**	**(56.3)**	**7.6**	**11.9**	**14.3**

Bracketed numbers indicate the models selected for the GPT-4o repair experiment to manage computational costs

mechanisms. As shown in Table 7, which provides a comparative analysis of repair performance across all evaluated LLMs using the *Correct@6* metric, the dual-agent paradigm (Auto-Feedback) consistently outperforms the single-agent approach across all models. Within this paradigm, GPT-4o achieved the highest repair performance.

To gain deeper insights into the complementarity of these methods, the Venn diagrams in Fig. 2 illustrate cases fixed exclusively by one tool but not by others, as well as the overlap between techniques (i.e., cases correctly repaired by multiple tools). In particular, the repair pipeline successfully addresses faulty models that other SoTA repair tools cannot resolve.

Influence of Defective Model Complexity/Size on Repair Performance To investigate the relationship between model complexity and repair performance, we assessed complexity based on the number of variables clauses in the propositional formulas generated by the Alloy Analyzer from LLM-produced repairs (see Tables 8 and 9) and conducted a detailed analysis thereof.

GPT-4o consistently outperforms GPT-3.5-Turbo in repairing faulty specifications, despite the latter generating more complex formulas with significantly more clauses. The clause-to-variable ratio further highlights this difference (2.87 for GPT-3.5-Turbo vs. 1.36 for GPT-4o), indicating that GPT-4o constructs more concise and efficient logical expressions. This suggests that GPT-4o achieves more efficient and accurate repairs by leveraging simpler logical structures while maintaining correctness.

The tendency of GPT-3.5-Turbo to generate more verbose and constraint-heavy formulas, even with comparable variable counts, underscores the limitations imposed by its smaller context window and less sophisticated reasoning capabilities. The consistently higher clause-to-variable ratios across benchmarks for GPT-3.5-Turbo further support this conclusion. These findings confirm that model capability–especially in context comprehension and logical reasoning–is a critical factor for successful Alloy repair. GPT-4o's superior performance, as highlighted in the %repair row of Table 6, demonstrates how advances in LLM architecture directly enhance formal reasoning and model repair effectiveness.

Discussing LLMs repair performance The evaluation reveals a clear performance gap between GPT-3.5-Turbo and the GPT-4 family across all experimental configurations. GPT-3.5-Turbo consistently exhibits the lowest repair rates, largely due to its architectural limitations such as a smaller context window and less advanced reasoning capabilities. These constraints hinder its ability to effectively incorporate iterative feedback and maintain coherence over successive repair attempts, often causing context overflows and semantic inconsistencies. Additionally, GPT-3.5-Turbo frequently generates Alloy-specific errors, including incorrect operator usage, type mismatches, and hallucinated constructs, that are less common in specialized repair tools designed with native Alloy semantics. These shortcomings persist even in simpler Alloy4Fun models, limiting its practical utility in automated specification repair. In contrast, GPT-4 models, including GPT-4-32k, GPT-4-Turbo, and GPT-4o, demonstrate significantly improved repair performance across all settings and benchmarks. Benefiting from larger context windows and enhanced reasoning abilities, these models handle complex specifications and multi-line bugs more effectively. Among them, GPT-4o and GPT-4-Turbo achieve comparable results, with GPT-4o often leading in agentic (Auto-Feedback) configurations that leverage multi-agent prompt construction

Table 7 Comparison of repair performance for different LLMs across various feedback levels–No-Feedback (NF), Generic-Feedback (GF), and Auto-Feedback (AF)–as measured by the Correct@6 metric

GPT-3.5-Turbo			GPT-4-32k			GPT-4-Turbo			GPT-4o		
Single-agent		Dual-agent	Single-agent		Dual-agent	Single-agent		Dual-agent	Single-agent		Dual-agent
NF	GF	AF	NF	GF	AF	NF	GF	AF	NF	GF	AF
10.5	15.8	47.4	39.5	42.1	57.9	44.7	50.0	73.4	63.2	71.1	73.4

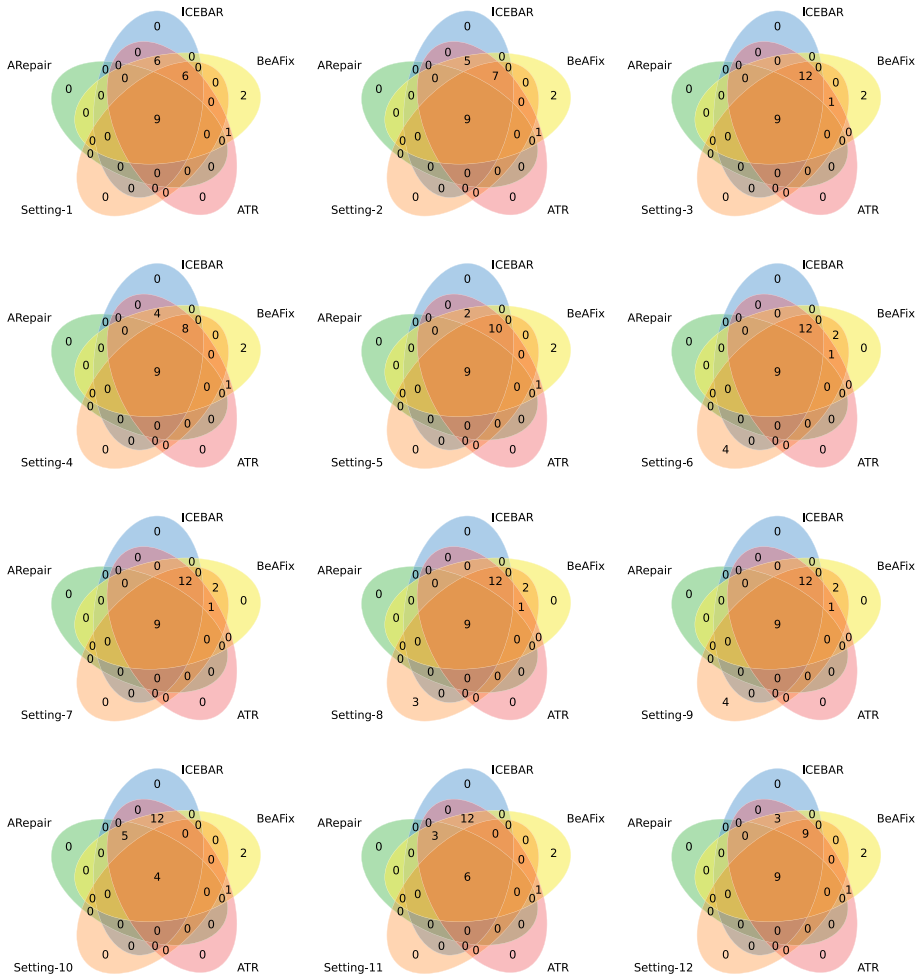


Fig. 2 Venn diagrams showing the exclusive and overlapping successful repairs for the ARepair benchmark achieved by different repair methods. These illustrate the complementary and unique capabilities of the APR pipeline compared to state-of-the-art (SoTA) repair tools

Table 8 Average number of variables and clauses in propositional formulas generated by the Alloy Analyzer from LLM-produced repairs on the ARepair benchmark

Model	Total # spec	LLM Models			
		GPT-4-32k	GPT-4-Turbo	GPT-4o	GPT-3.5-Turbo
addr	1	variables: 333	variables: 567	variables: 66	variables: 187
		clauses: 368	clauses: 627	clauses: 74	clauses: 215
arr	2	variable: 1,973	variable: 7,578	variables: 6,364	variables: 1,482
		clauses: 4,969	clauses: 16,284	clauses: 6,096	clauses: 3,314
balancedBSt	3	variables: 2,001	variables: 15,312	variables: 4,155	variables: 1,797
		clauses: 3,720	clauses: 52,602	clauses: 8,616	clauses: 3,306
bempl	1	variables: 2,249	variables: 700	variables: 744	variables: 694
		clauses: 3,489	clauses: 1,012	clauses: 1,071	clauses: 1,003
cd	2	variables: 294	variables: 280	variables: 288	variables: 576
		clauses: 196	clauses: 164	clauses: 326	clauses: 1,188
ctree	1	variables: 2,976	variables: 1,130	variables: 1,721	variables: 1,679
		clauses: 2,857	clauses: 1,035	clauses: 1,591	clauses: 1,524
dll	4	variables: 21,572	variables: 20,012	variables: 21,524	variables: 6,324
		clauses: 66,088	clauses: 65,248	clauses: 50,128	clauses: 4,108
farmer	1	variables: 1,468	variables: 1,213	variables: 2,252	variables: 1,199
		clauses: 2,808	clauses: 2,177	clauses: 4,096	clauses: 2,429
fsm	2	variables: 894	variables: 2,224	variables: 540	variables: 1,080
		clauses: 1,816	clauses: 4,556	clauses: 1,108	clauses: 2,278
grade	1	variables: 4,763	variables: 2,830	variables: 1,024	variables: 491
		clauses: 7,930	clauses: 4,578	clauses: 1,585	clauses: 604
other	1	variables: 477	variables: 264	variables: 113	variables: 268
		clauses: 78	clauses: 352	clauses: 128	clauses: 358
Student	19	variables: 100,548	variables: 63,992	variables: 31,236	variables: 63,878
		clauses: 321,727	clauses: 208,696	clauses: 20,520	clauses: 208,354
Total	**38**	**variables: 139,548**	**variables: 116,102**	**variables: 70,027**	**variables: 79,655**
		clauses: 416,046	**clauses: 357,331**	**clauses: 95,339**	**clauses: 228,681**

Results are reported as overall averages across all evaluated settings for each LLM model

and iterative feedback. This dual-agent paradigm consistently boosts repair success, highlighting the advantages of advanced LLM architectures combined with adaptive prompting strategies.

> The results for RQ1 indicate that an agentic LLM incorporating an iterative feedback mechanism achieves improved repair performance compared to both traditional state-of-the-art Alloy repair tools and non-agentic LLM configurations. Notably, GPT-4o consistently outperformed other LLMs, yielding the most effective repair outcomes across both agentic and non-agentic settings.

Table 9 Average number of variables and clauses in propositional formulas generated by the Alloy Analyzer from LLM-produced repairs on the A4F benchmark

		GPT-4o	GPT-3.5-Turbo	
Model	# specs used	Variables	# specs used	Variables
classroom	60	variables: 35,760 clauses: 47,340	60	variables: 3,580 clauses: 4,691
cv	60	variables: 105,660 clauses: 8,640	60	variables: 31,320 clauses: 34,620
graphs	60	variables: 17,040 clauses: 37,200	60	variables: 7,902 clauses: 6,180
lts	60	variables: 26,100 clauses: 26,820	60	variables: 88,560 clauses: 199,260
production	60	variables: 16,560 clauses: 19,560	60	variables: 61,380 clauses: 124,560
trash	57	variables: 7,068 clauses: 8,892	57	variables: 23,598 clauses: 43,605
Total	**357**	**variables: 208,188 clauses: 148,452**	**357**	**variables: 216,340 clauses: 412,916**

Results are reported as overall averages across all evaluated settings for each LLM model. For a fair comparison, results are based on the same set of 357 specifications for both GPT-3.5-Turbo and GPT-4o

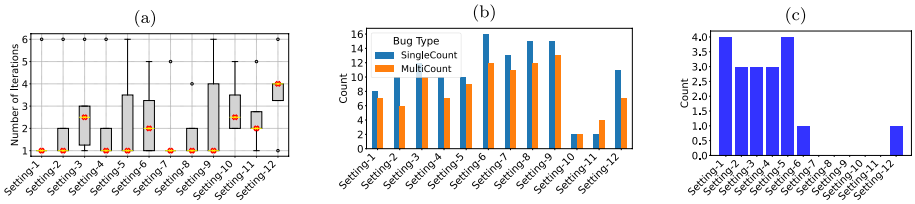


Fig. 3 (a) Iteration count distribution for repairing specifications across various settings; (b) Repair distribution categorized by bug type: single-line or multi-line (higher values preferred); (c) The incident count for initial repair attempts mirroring the buggy specification (lower values preferred)

6.2 Results for RQ2: Performance

This research question delves into the repair performance of various pre-trained LLMs concerning factors such as repair iteration budget, bug type, localization capabilities, and adherence to instructions.

This analysis leverages a comprehensive benchmark that includes all evaluated LLMs and a diverse range of defect types, encompassing both single-line and multi-line bugs. As shown in Table 5, each defective model was tested across up to 12 distinct settings and up to 6 repair iterations, resulting in a total of 2,736 repair trials–effectively corresponding to the repair of 2,736 defective models. This extensive experimental setup provides a robust and representative evaluation of the repair capabilities of the LLMs across a wide variety of scenarios.

Repair Iteration Budget The box-whisker plot in Fig. 3(a) illustrates the iterations needed by different settings to rectify faulty specifications. Across all settings, the median number of iterations predominantly centers around 1.0, except for Settings 3, 6, and 10-12. Although

Settings 3 and 6 exhibit superior repair capabilities, they may require a marginally higher number of iterations to achieve optimal results. However, median values for Settings 1, 2, 4, and 5 remain at 1.0, suggesting that at least half of the issues are resolved within the first iteration. In contrast, the median values for Settings 7–9 are consistently 1.0, indicating that such settings require the fewest repair iterations, independent of the employed feedback mechanism.

Repairing Based on Bug Type Figure 3(b) illustrates the repair effectiveness of various configurations in the repair pipeline with respect to bug types–specifically, single-line and multi-line defects. The dual-agent configurations in Setting-6 (auto-feedback, GPT-4-Turbo) and Setting-9 (auto-feedback, GPT-4o) demonstrated the highest overall repair rates, successfully fixing 16 single-line and 12 multi-line bugs, and 15 single-line and 13 multi-line bugs, respectively. In comparison, the best-performing configuration for GPT-4-32k (Setting-3, auto-feedback) showed relatively stronger performance on single-line bugs, repairing 12 Alloy models, versus 10 for multi-line bugs.

Following Instructions Adherence to instructions is crucial, in tasks such as automated repair, to ensure repair quality and control costs. Figure 3(c) reveals that both GPT-4 models sometimes repeat buggy models in their initial repair iterations, despite system instructions to avoid this. GPT-4o generally shows better compliance, under all settings, with zero number of repeated buggy specifications. Interestingly, GPT-3.5-Turbo exhibited similar behavior to GPT-4o.

> The evaluated LLMs exhibited varying behaviors across key factors, including instruction adherence, the number of repair iterations, and effectiveness in handling both single-line and multi-line bugs. Among all models and across all settings, GPT-4o consistently outperformed its counterparts across all measured dimensions. Notably, the dual-agent configuration generally resulted in an increased number of iterations for all models–except GPT-4o.

6.3 Results for RQ3: Characteristics of Failed Repairs

This research question explores the underlying causes of unsuccessful repair iterations. Figure 4 illustrates the distribution of failed iterations in different settings and provides a comprehensive breakdown of the reasons for the failure. Predominantly, failures arise when the proposed repairs fail to fulfill the assertions, resulting in the generation of counterexamples. Additionally, syntax errors in proposed repairs impede the compilation process by Alloy Analyzer. Settings 1-3, employing GPT-4-32k, manifest instances of failures attributed to incorrect message formats, a phenomenon absent in Settings 4-6 utilizing GPT-4-Turbo. Such disparities suggest discrepancies in the compliance of LLM responses with the JSON format mandated by the alloy_analyzer_tool. Categories such as “Repetition” and “No instances” exhibit lower prevalence. “Repetition” denotes instances where proposed repairs same as the supplied defective models, while “No instances” signify scenarios

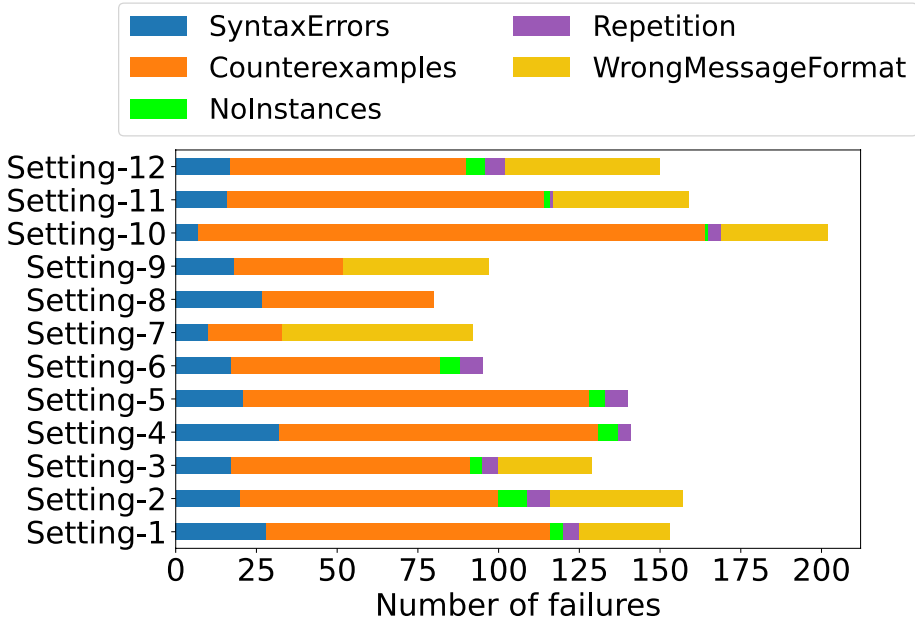


Fig. 4 Error types observed in failed repair iterations across all settings

where proposed models fail to generate instances, indicative of an incapacity to meet model constraints.

Figure 5 tracks the results across all iterations in Setting-6, revealing the evolving nature of failure categories during the iterative prompting process. Although instances of "Counterexample" failures are effectively addressed, challenges persist to rectify issues classified under "Repetition".

We investigated the repair behavior of the ten defective models that remained unresolved after six iterations. Among these models, five were categorized as "Counterexample," four exhibited syntax errors, and one was identified as a "No instance". Notably, the statuses of these ten specifications have remained largely unchanged since iterations 3 and 4. Particularly, the defective model `arr1` consistently exhibited the same status from the first iteration onwards. This persistent behavior corroborates our preliminary analysis regarding the number of iterations, wherein no further progress was observed beyond the sixth iteration.

Constraint-Related Issues Our analysis of LLM-generated results reveals an important nuance in constraint-related issues: underconstrained and overconstrained specifications often coexist with various error manifestations. These fundamental constraint problems can simultaneously present as syntax errors, type errors, absence of valid instances, and counter examples. Rather than being distinct categories, constraint issues and error types frequently intersect. Addressing these interrelated challenges requires sophisticated specification engineering techniques that can balance constraint expressiveness with feasibility, ultimately leading to more reliable and robust LLM-based specification repairs.

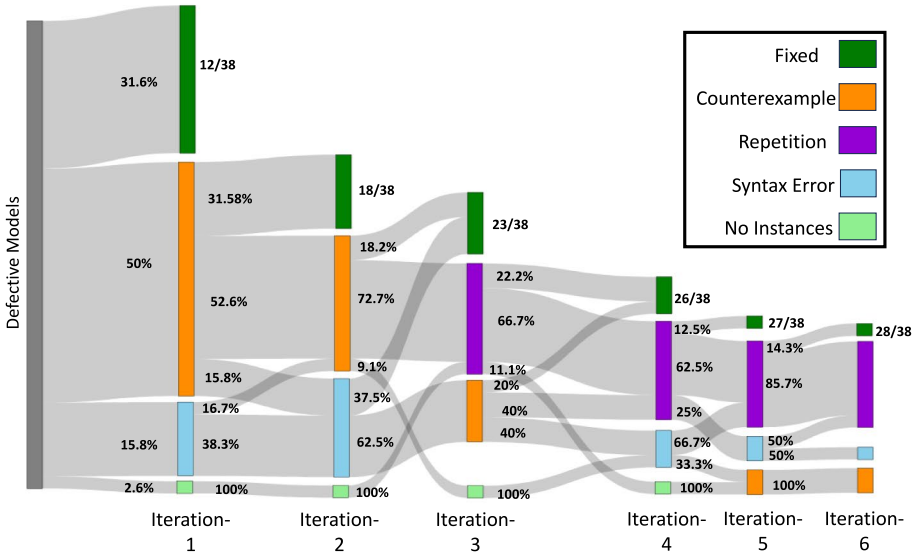


Fig. 5 Repair progress in dual-agent Setting-6 (GPT-4-Turbo and Auto-Feedback) across iterations. Percentage numbers show transition volumes between statuses in subsequent iterations. Cumulative counts of fixed models follow each "Fixed" block

Our analysis proceeds in two principal steps to identify specification constraint defects across LLM-generated models. First, we detect *overconstrained* specifications by inspecting counterexamples and the presence of instances. If a counterexample is present and no instance is found–or if only a counterexample is detected, indicating overly restrictive constraints–we classify the model as *overconstrained*. This suggests that the model's constraints are too strict to allow any valid instance. Second, for specifications not flagged as *overconstrained*, we identify *underconstrained* issues by comparing each model against our ground-truth benchmarks. If there is a discrepancy in the constraints–such as a missing constraint or an implementation that fails to reflect the intended semantics–we label the model as *underconstrained*. This two-stage procedure ensures that each model is categorized into exactly one defect class or deemed correct if neither condition applies.

Table 10 summarizes the distribution of overconstrained and underconstrained specifications among all unfixed issues across various LLM configurations on the ARepair and Alloy4Fun benchmarks, respectively. Notably, the results reveal that the constraint tendencies of each LLM can vary substantially depending on the benchmark context. On ARepair, GPT-4-32k and GPT-4-Turbo most frequently produce overconstrained specifications, with over 56% of unfixed issues falling into this category across most settings. GPT-4o shows mixed behavior, with some settings demonstrating a greater tendency toward underconstraining, particularly settings 7 and 9 (57.1% and 50.0% respectively).

In contrast, on the Alloy4Fun benchmark, GPT-4o consistently produces overconstrained specifications (60-62% of unfixed issues), while GPT-3.5-Turbo maintains a similar ratio of overconstrained issues (approximately 60%). This reversal in behavior between benchmarks is particularly notable, suggesting that its constraint tendencies are heavily influenced by the specific characteristics and complexities of the benchmark rather than being an intrinsic property of the model. These findings underscore a consistent trade-off between

Table 10 Frequency of *overconstrained* and *underconstrained* specifications among all unfixed issues across all LLM settings on ARepair and Alloy4Fun benchmarks

ARepair Benchmark												
Part A: Over-Constrained Issues												
Model	GPT-4-32k			GPT-4-Turbo			GPT-4o			GPT-3.5-Turbo		
	1	2	3	4	5	6	7	8	9	10	11	12
addr	1	1	0	1	0	0	1	1	1	1	1	1
arr	1	1	1	0	1	0	0	0	0	1	0	0
balancedBST	1	1	0	1	1	1	1	1	1	2	2	1
bempl	0	0	0	0	0	0	0	1	1	0	0	0
cd	0	0	0	0	1	0	0	0	0	1	1	1
ctree	0	0	0	0	0	0	0	0	0	1	1	1
dll	0	0	0	1	1	0	0	0	0	2	0	0
farmer	0	0	0	0	0	0	0	0	0	0	0	0
fsm	0	0	0	1	1	0	0	0	0	1	1	1
grade	1	1	1	1	1	0	1	1	0	1	1	1
other	1	1	1	1	1	0	0	0	0	0	1	0
student	8	8	6	7	5	3	3	2	2	12	11	5
Summary	**13**	**13**	**9**	**13**	**12**	**4**	**6**	**6**	**5**	**22**	**19**	**11**
%ratio	**56.5**	**59.1**	**56.3**	**61.9**	**63.2**	**40.0**	**42.9**	**54.5**	**50.0**	**64.7**	**59.4**	**55.0**
Part B: Under-Constrained Issues												
Model	**GPT-4-32k**			**GPT-4-Turbo**			**GPT-4o**			**GPT-3.5-Turbo**		
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
addr	0	0	0	0	0	0	0	0	0	0	0	0
arr	1	1	1	0	1	0	0	0	0	1	0	0
balancedBST	1	1	0	1	1	1	1	1	1	1	1	1
bempl	1	1	1	0	0	0	1	0	0	1	1	1
cd	1	0	0	0	0	0	0	0	0	0	1	1
ctree	0	0	0	0	0	0	0	0	0	0	0	0
dll	0	0	0	1	1	0	0	0	0	1	1	1
farmer	0	0	0	0	0	0	0	0	0	0	0	0

Table 10 (continued)

ARepair Benchmark												
Part A: Over-Constrained Issues												
Model	GPT-4-32k			GPT-4-Turbo			GPT-4o			GPT-3.5-Turbo		
	1	2	3	4	5	6	7	8	9	10	11	12
fsm	1	1	0	1	1	1	1	1	0	1	1	1
grade	0	0	0	0	0	0	0	0	0	0	0	0
other	0	0	0	0	0	0	0	0	0	0	0	0
student	5	5	4	4	3	3	4	3	3	7	7	3
Summary	**10**	**9**	**6**	**8**	**7**	**6**	**8**	**5**	**5**	**12**	**13**	**9**
%ratio	**43.5**	**40.9**	**37.5**	**38.1**	**36.8**	**60.0**	**57.1**	**45.5**	**50.0**	**35.3**	**40.6**	**45.0**

Alloy4Fun Benchmark						
Part A: Over-Constrained Issues						
Model	**#issues Under Setting**					
	GPT-4o			**GPT-3.5-Turbo**		
	7	**8**	**9**	**10**	**11**	**12**
classroom	(38)	(39)	(35)	612	610	593
cv	(3)	(1)	(0)	60	65	41
graphs	(23)	(13)	(23)	150	135	145
lts	(13)	(8)	(9)	110	82	88
production	(1)	(0)	(1)	22	18	16
trash	(32)	(33)	(27)	120	120	114
Summary	**(110)**	**(94)**	**(95)**	**1074**	**1030**	**997**
%ratio	**(60.8)**	**(62.3)**	**(61.0)**	**60.3**	**60.4**	**60.1**
Part B: Under-Constrained Issues						
Model	**#issues Under Setting**					
	GPT-4o			**GPT-3.5-Turbo**		
	7	**8**	**9**	**10**	**11**	**12**
classroom	(22)	(21)	(19)	387	388	385
cv	(1)	(1)	(1)	40	41	27

Table 10 (continued)

ARepair Benchmark																		
Part A: Over-Constrained Issues																		
Model	GPT-4-32k			GPT-4-Turbo			GPT-4o			GPT-3.5-Turbo								
	1	2	3	4	5	6	7	8	9	10	11	12						
graphs	(15)	(7)	(15)	110	97	103												
lts	(9)	(4)	(5)	71	52	55												
production	(0)	(0)	(1)	14	11	11												
trash	(24)	(24)	(20)	86	86	80												
Summary	**(71)**	**(57)**	**(61)**	**708**	**675**	**661**												
%ratio	**(39.2)**	**(37.7)**	**(39.0)**	**39.7**	**39.6**	**39.9**												
Total	**ARepair**												**Alloy4Fun**					
Issues	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**7**	**8**	**9**	**10**	**11**	**12**
Over-constrained	13	13	9	13	12	4	6	6	5	22	19	11	(110)	(94)	(95)	1074	1030	997
Under-constrained	10	9	6	8	7	6	8	5	5	12	13	9	(71)	(57)	(61)	708	675	661

generating overly strict and overly permissive constraints, with distinct patterns emerging for each model and benchmark. Our systematic categorization method robustly captures these nuanced behaviors, providing insight into both model capabilities and the influence of benchmark design on LLM-driven specification repair.

> Analysis of failure characteristics during the repair process highlights common issues such as proposed repairs failing assertions and syntax errors, with observed transitions between failure categories over iterative prompting. Therefore, increasing the number of iterations does not necessarily result in a successful repair of the model.

6.4 Results for RQ4: Repair Costs

We consider two primary cost factors to assess the financial implications of utilizing the APR pipeline with different LLMs: (i) time taken for bug resolution and (ii) monetary expenses associated with token consumption, based on OpenAI's pricing as of March 2024.

Time Analysis We measure the time taken for bug resolution using the results presented in Table 5. Figure 6 illustrates the runtime analysis, contrasting fixed and unfixed scenarios. Across different settings, the median duration required for bug repair varies from 37.3 seconds in Setting-1 to 120.3 seconds in Setting-3. Notably, the maximum time spent on repairs reached 493.28 seconds in Setting-5. Conversely, the median duration associated with unfixed bugs amounted to 273.8 seconds and 392.2 seconds in Settings-1 and 4, respectively. Interestingly, all settings supporting the *dual-agent* setup (i.e., 3, 6, 9, and 12) consistently exhibit increased running time compared to the *single-agent* configuration, across all LLMs and under both fixed and unfixed bug scenarios.

Monetary Costs Figure 7 presents the monetary costs incurred under each setting for both fixed and unfixed models. Settings utilizing GPT-3.5-Turbo show the lowest costs for both fixed and unfixed models, attributed to the cheaper cost per token for GPT-3.5-Turbo compared to GPT-4 family models (see Table 3). Additionally, settings employing the *dual-agent* (3, 6, 9, and 12) exhibit marginally higher costs than their counterparts utilizing the same LLM under the *single-agent* setup.

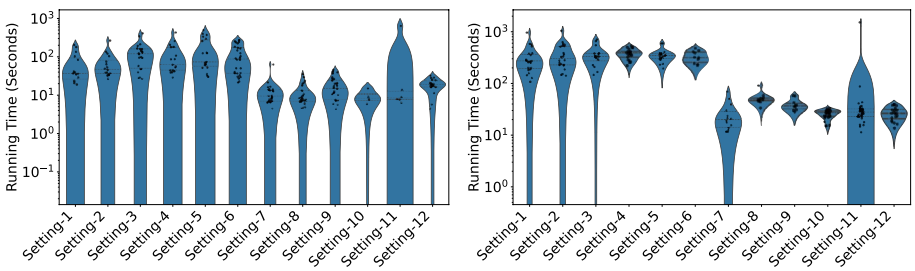


Fig. 6 Distribution of running time across all settings for fixed bugs (left) and unfixed bugs (right)

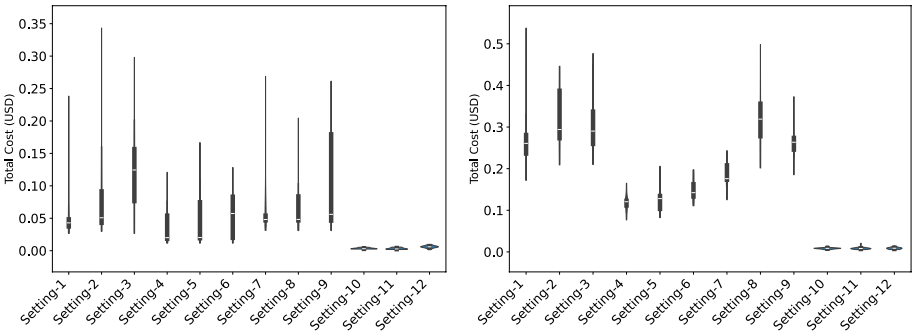


Fig. 7 Monetary distribution across all settings for fixed bugs (left) and unfixed bugs (right)

The analysis of repair costs in RQ4 reveals notable variations in both execution time and monetary expenditure across different configurations. Among the models evaluated, GPT-3.5-Turbo consistently incurs the lowest cost for bug resolution. Within the GPT-4 family, GPT-4o demonstrates superior efficiency, achieving the lowest repair cost and runtime. Additionally, settings employing the *dual-agent* paradigm tend to exhibit slightly higher expenses compared to their single-agent counterparts.

7 Threats to Validity

Internal Validity Variations in prompt design and format may introduce biases in the repair process, affecting the outcomes independently of the LLM's capabilities. To mitigate this threat, we carefully standardized the prompt design across all experiments, ensuring consistency in the information provided to the LLMs. Additionally, we conducted analyses to assess the impact of prompt variations on repair performance, enabling us to isolate the effects of LLM capabilities from potential biases introduced by prompt design.

External Validity The generalizability of our findings may be limited by the specific characteristics of the benchmarks used. To address this, we utilized diverse benchmark suites and evaluated the performance of the repair pipeline across various scenarios. Furthermore, we documented the experimental setup comprehensively to facilitate replication and external validation of our results. Finally, some repair instances in our benchmark have been fixed in prior work and may have been included in the training data of the LLMs. This raises the concern that the models might be memorizing existing repairs rather than generating novel solutions, which could limit their applicability to Alloy models they have not been trained on. As a result, this poses a threat to the generalizability of our findings, as the LLMs may perform well on known faults but struggle with previously unseen ones. Future work could further investigate this aspect, following the approach of Salerno et al. (2025), and introduce

a new benchmark dataset–analogous to the ConDefects dataset (Wu et al. 2024)–designed specifically to mitigate the issue of data leakage.

Construct Validity Biases in the measurement of repair performance metrics could distort our assessment of LLM capabilities. To mitigate this, we employed standardized metrics and conducted sensitivity analyses to validate the robustness of our findings. Additionally, we ensured transparency and reproducibility in our methodology to enhance the validity of our measurements.

8 Related Work

Several recent studies have explored the integration of Large Language Models (LLMs) into software engineering tasks, particularly in the realm of program repair.

AlphaRepair (Xia and Zhang 2022) leverages LLMs for APR in a zero-shot setting, but it requires removing the buggy line and replacing it with masked tokens. It then queries the LLM to fill-in the masked tokens with the correct tokens to generate patches. Xia and Zhang (2023a) improve APR performance by incorporating test feedback into prompts, while Kang et al. (2023) enabled LLMs to utilize a debugger for information gathering and patch generation. Additionally, Fuzz4All (Xia et al. 2024) leverages LLMs as an input generation and mutation engine, employing an auto-prompting phase to generate concise input prompts. RepairAgent (Bouzenia et al. 2024) employs LLMs, agents, and dynamic prompts for APR, albeit in the context of repairing Java applications. TestPilot (Schäfer et al. 2024) uses LLMs to generate unit test cases for JavaScript, employing a few-shot learning approach to refine prompts with failed tests and error messages.

In contrast to these efforts, our study focuses on using LLMs for automated specification repair, a less explored area in software engineering. We adopt an approach similar to Fuzz4All's auto-prompting phase, specifically by incorporating LLM to construct the prompt (Shin et al. 2020). Moreover, while existing work predominantly uses LLMs for program repair in imperative languages like Java and JavaScript, our study extends the application of LLMs to the domain of Alloy specifications, addressing a gap in the literature regarding specification repair techniques for declarative languages. Consequently, a direct quantitative comparison is largely infeasible, with the exception of the tool proposed by Hasan et al. (2023), which utilizes LLMs to repair Alloy specifications. Nevertheless, Table 11 presents a comparative analysis of state-of-the-art LLM-based APR tools. This comparison highlights that our APR pipeline incorporates recent advances and techniques in LLMs that are also employed by contemporary repair tools.

Table 11 Comparing the feature that we considered in the APR pipeline with contemporary state-of-the-art LLM-APR tools, the column labeled "Autoprompt" denotes that the LLM generates the prompt

APR Tools	Agentic	Tools	Feedback	Autoprompt	Zero-shot	Spec Repair	# LLMs
AlphaRepair (Xia and Zhang 2022)	×	×	×	×	✓	×	1
ChatRepair (Xia and Zhang 2023b)	×	✓	✓	×	✓	×	1
conversationalAPR (Xia and Zhang 2023a)	×	✓	✓	×	✓	×	10
Xia et al. (2023)	×	×	×	×	×	×	9
Hasan et al. (2023)	×	✓	×	×	✓	✓	1
RepairAgent (Bouzenia et al. 2024)	✓	✓	✓	×	✓	×	1
Ours	✓	✓	✓	✓	✓	✓	4

9 Conclusion

In this study, we explore the potential of pre-trained LLMs to facilitate the repair of Alloy specifications, taking into account recent advancements in LLMs, including the use of agents, feedback mechanisms, zero-shot learning capabilities, and auto-prompting techniques. The investigation reveals that employing a dual-agent repair pipeline enhances the repair process, albeit with a marginal increase in token consumption. The comparative evaluation highlights the superior performance of the GPT-4 model family over GPT-3.5-Turbo, underscoring their promising applicability. Overall, the findings of this research indicate a positive outlook for applying LLMs in automated program repair for declarative specifications, particularly through repair pipelines that integrate contemporary innovations in the field of LLMs.

Appendix Additional Results

This section presents the results for the Alloy4Fun benchmark based on settings 7-12. Figs. 8, 9, 10, 11, 12 and 13.

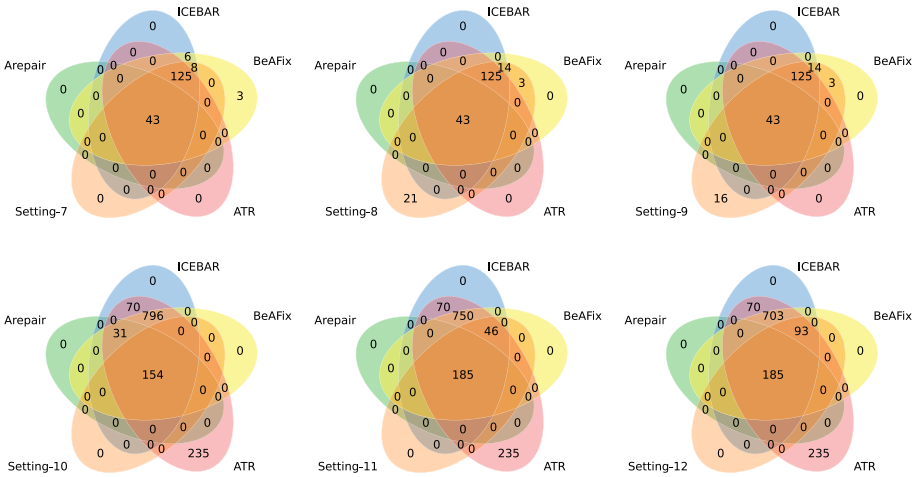


Fig. 8 Venn diagrams showing the exclusive and overlapping successful repairs for the Alloy4Fun benchmark achieved by different repair methods. These illustrate the complementary and unique capabilities of the APR pipeline compared to state-of-the-art (SoTA) repair tools. Settings 7-9 are based on the sampled 357 models, while settings 10-12 are based on the entire set of models in the benchmark. Accordingly, the results are reported for the SoTA tools

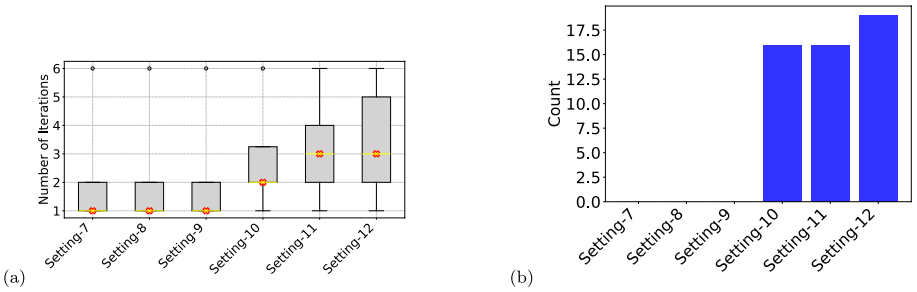


Fig. 9 Alloy4Fun benchmark results: (a) Iteration count distribution for repairing specifications across various settings. (b) Incident count for initial repair attempts mirroring the buggy specification (lower values preferred)

Fig. 10 Error types observed in failed repair iterations for the Alloy4Fun benchmark across settings 7-9

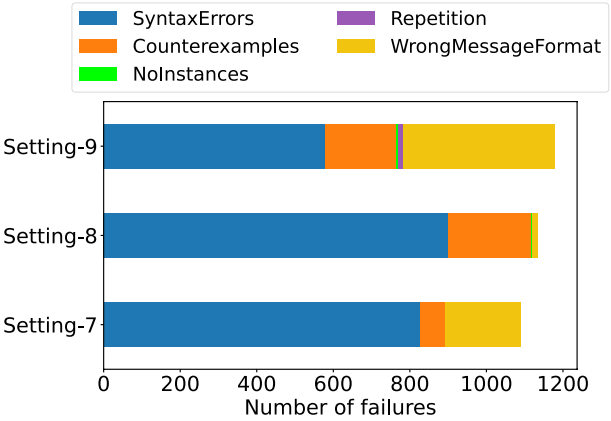
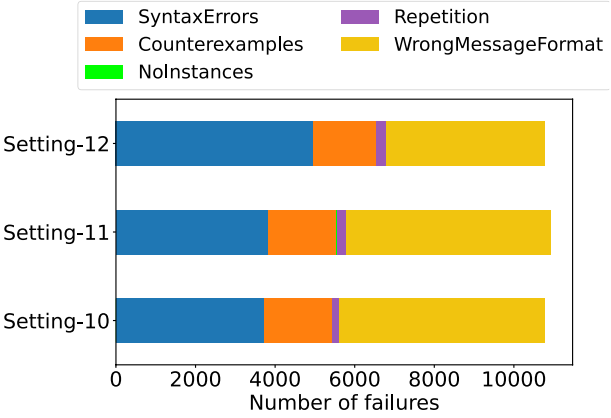


Fig. 11 Error types observed in failed repair iterations for the Alloy4Fun benchmark across settings 10-12



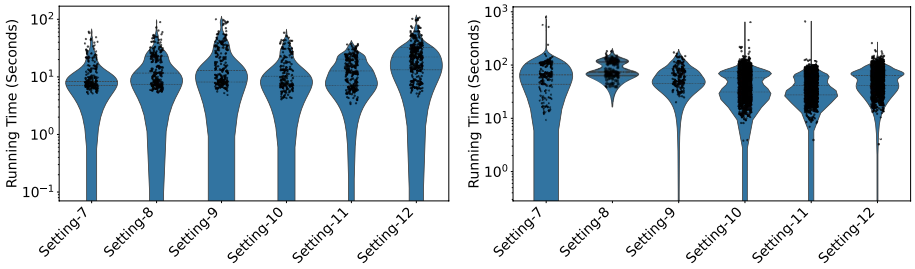


Fig. 12 Distribution of running time across settings 7-12 for the Alloy4Fun benchmark for fixed bugs (left) and unfixed bugs (right)

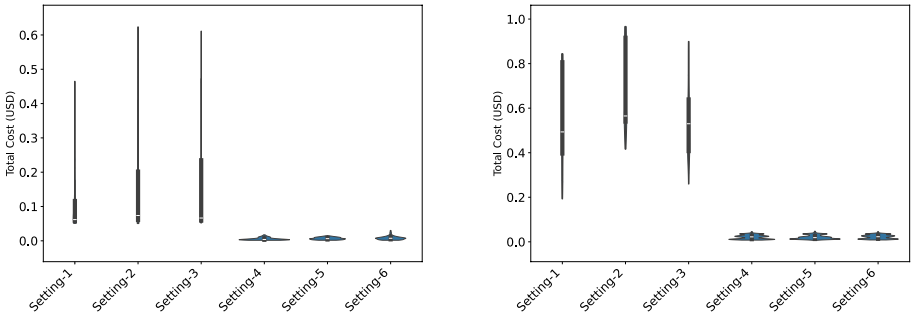


Fig. 13 Monetary distribution across settings 7-12 for the Alloy4Fun benchmark for fixed bugs (left) and unfixed bugs (right)

Author Contributions Mohannad Alhanahnah: Design, implementation, and evaluation of the APR pipeline. He also drafted and prepared the manuscript.
Md Rashedul Hasan: Experimental Data Collection, Results Analysis, Evaluation of the APR pipeline. He also drafted sections of the Experimental Results.
Lisong Xu: Design and analysis of experiments.
Hamid Bagheri: Conceptualization of the research idea, team formation and supervision, expert guidance on Alloy and project direction, and management of team collaboration throughout the study.

Funding Open access funding provided by University of Gothenburg. This work was supported in part by awards CCF-1755890, CCF-1618132, CCF-2139845, and CCF-2124116 from the National Science Foundation.

Data Availability All data used to write this paper is open-source and publicly available at https://github.com/Mohannadcse/AlloySpecRepair.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

References

Alhanahnah M, Stevens C, Bagheri H (2020) Scalable analysis of interaction threats in iot systems. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2020, p 272–285. https://doi.org/10.1145/3395363.3397347

AutoGPT (2024) AutoGPT. https://github.com/Significant-Gravitas/AutoGPT Accessed: 30-March-2024

Bagheri H, Wang J, Aerts J, Ghorbani N, Malek S (2021) Flair: efficient analysis of android inter-component vulnerabilities in response to incremental changes. Empir Softw Eng 26(3):54

Bagheri H, Kang E, Mansoor N (2020) Synthesis of assurance cases for software certification. In: ICSE-NIER 42nd international conference on software engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020, ACM, pp 61–64

Bagheri H, Sadeghi A, Jabbarvand R, Malek S (2016) Practical, formal synthesis and automatic enforcement of security policies for android. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp 514–525. https://doi.org/10.1109/DSN.2016.53

Bouzenia I, Devanbu P, Pradel M (2025) RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE Computer Society, Los Alamitos, CA, USA, pp 2188–2200. https://doi.org/10.1109/ICSE55347.2025.00157, URL https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00157

Brida SG, Regis G, Zheng G, Bagheri H, Nguyen T, Aguirre N, Frias M (2021) Bounded exhaustive search of alloy specification repairs. In: Proceedings of the 43rd international conference on software engineering, IEEE Press, ICSE '21, p 1135–1147, https://doi.org/10.1109/ICSE43902.2021.00105

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

Bubeck S, Chandrasekaran V, Eldan R, Gehrke J, Horvitz E, Kamar E, Lee P, Lee YT, Li Y, Lundberg S, Nori H, Palangi H, Ribeiro MT, Zhang Y (2023) Sparks of artificial general intelligence: Early experiments with gpt-4. https://arxiv.org/abs/2303.12712,2303.12712

Chase H (2022) LangChain. https://github.com/langchain-ai/langchain Accessed: 27-Feb-2024

Chen M, Tworek J, Jun H, Yuan Q, Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A (2021) Evaluating large language models trained on code. ArXiv abs/2107.03374. https://api.semanticscholar.org/CorpusID:235755472

Chowdhery A, Narang S, Devlin J, Bosma M, Mishra G, Roberts A, Barham P, Chung HW, Sutton C, Gehrmann S, et al. (2023) Palm: Scaling language modeling with pathways. J Mach Learn Res 24(240):1–113

Fan Z, Gao X, Mirchev M, Roychoudhury A, Tan SH (2023) Automated repair of programs from large language models. In: Proceedings of the 45th International Conference on Software Engineering, IEEE Press, ICSE '23, pp 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

Gutiérrez Brida S, Regis G, Zheng G, Bagheri H, Nguyen T, Aguirre N, Frias M (2023) Icebar: Feedback-driven iterative repair of alloy specifications. In: Proceedings of the 37th IEEE/ACM international conference on automated software engineering, association for computing machinery, New York, NY, USA, ASE '22. https://doi.org/10.1145/3551349.3556944

Hasan MR, Li J, Ahmed I, Bagheri H (2025) Automated repair of declarative software specifications in the era of large language models. https://doi.org/10.21227/yt7g-bk72

Jackson D (2006) Software abstractions: logic, language, and analysis. The MIT Press

Jain N, Vaidyanath S, Iyer A, Natarajan N, Parthasarathy S, Rajamani S, Sharma R (2022) Jigsaw: large language models meet program synthesis. In: Proceedings of the 44th international conference on software engineering, Association for Computing Machinery, New York, NY, USA, ICSE '22, pp 1219–1231. https://doi.org/10.1145/3510003.3510203

Kang S, Chen B, Yoo S, Lou J (2025) Explainable automated debugging via large language model-driven scientific debugging. Empir Softw Eng 30(2). https://doi.org/10.1007/s10664-024-10594-x

Khurshid S, Marinov D (2004) Testera: Specification-based testing of java programs using sat. Automated Software Engineering 11(4):403–434. https://doi.org/10.1023/B:AUSE.0000038938.10589.b9

Kong J, Xie X, Cheng M, Liu S, Du X, Guo Q (2025) Contrastrepair: enhancing conversation-based automated program repair via contrastive test case pairs. ACM Trans Softw Eng Methodol. https://doi.org/10.1145/3719345

Langroid (2024). https://github.com/langroid/langroid Accessed: 27-Feb-2024

Liu Z, Tang Y, Li M, Jin X, Long Y, Zhang LF, Luo X (2024) Llm-compdroid: repairing configuration compatibility bugs in android apps with pre-trained large language models. 2402.15078

Macedo N, Cunha A, Pereira J, Carvalho R, Silva R, Paiva AC, Sozinho Ramalho M, Silva D (2021) Experiences on teaching alloy with an automated assessment platform. Sci Comput Program 211(C). https://doi.org/10.1016/j.scico.2021.102690

Madaan A, Tandon N, Gupta P, Hallinan S, Gao L, Wiegreffe S, Alon U, Dziri N, Prabhumoye S, Yang Y, Gupta S, Majumder BP, Hermann K, Welleck S, Yazdanbakhsh A, Clark P (2023) Self-refine: iterative refinement with self-feedback. In: Proceedings of the 37th International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA, NIPS '23

Mirzaei N, Garcia J, Bagheri H, Sadeghi A, Malek S (2016) Reducing combinatorics in gui testing of android applications. In: Proceedings of the 38th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '16, pp 559–570. https://doi.org/10.1145/2884781.2884853

OpenAI, Achiam J, Adler S, Agarwal S, Ahmad L, Akkaya I, Aleman FL, Almeida D, Altenschmidt J, Altman S, Anadkat S, Avila R, Babuschkin I, Balaji S, Balcom V, Baltescu P, Bao H, Bavarian M, Belgum J, Bello I, Berdine J, Bernadett-Shapiro G, Berner C, Bogdonoff L, Boiko O, Boyd M, Brakman AL, Brockman G, Brooks T, Brundage M, Button K, Cai T, Campbell R, Cann A, Carey B, Carlson C, Carmichael R, Chan B, Chang C, Chantzis F, Chen D, Chen S, Chen R, Chen J, Chen M, Chess B, Cho C, Chu C, Chung HW, Cummings D, Currier J, Dai Y, Decareaux C, Degry T, Deutsch N, Deville D, Dhar A, Dohan D, Dowling S, Dunning S, Ecoffet A, Eleti A, Eloundou T, Farhi D, Fedus L, Felix N, Fishman SP, Forte J, Fulford I, Gao L, Georges E, Gibson C, Goel V, Gogineni T, Goh G, Gontijo-Lopes R, Gordon J, Grafstein M, Gray S, Greene R, Gross J, Gu SS, Guo Y, Hallacy C, Han J, Harris J, He Y, Heaton M, Heidecke J, Hesse C, Hickey A, Hickey W, Hoeschele P, Houghton B, Hsu K, Hu S, Hu X, Huizinga J, Jain S, Jain S, Jang J, Jiang A, Jiang R, Jin H, Jin D, Jomoto S, Jonn B, Jun H, Kaftan T, Lukasz Kaiser, Kamali A, Kanitscheider I, Keskar NS, Khan T, Kilpatrick L, Kim JW, Kim C, Kim Y, Kirchner JH, Kiros J, Knight M, Kokotajlo D, Lukasz Kondraciuk, Kondrich A, Konstantinidis A, Kosic K, Krueger G, Kuo V, Lampe M, Lan I, Lee T, Leike J, Leung J, Levy D, Li CM, Lim R, Lin M, Lin S, Litwin M, Lopez T, Lowe R, Lue P, Makanju A, Malfacini K, Manning S, Markov T, Markovski Y, Martin B, Mayer K, Mayne A, McGrew B, McKinney SM, McLeavey C, McMillan P, McNeil J, Medina D, Mehta A, Menick J, Metz L, Mishchenko A, Mishkin P, Monaco V, Morikawa E, Mossing D, Mu T, Murati M, Murk O, M´ely D, Nair A, Nakano R, Nayak R, Neelakantan A, Ngo R, Noh H, Ouyang L, O'Keefe C, Pachocki J, Paino A, Palermo J, Pantuliano A, Parascandolo G, Parish J, Parparita E, Passos A, Pavlov M, Peng A, Perelman A, de Avila Belbute Peres F, Petrov M, de Oliveira Pinto HP, Michael, Pokorny, Pokrass M, Pong VH, Powell T, Power A, Power B, Proehl E, Puri R, Radford A, Rae J, Ramesh A, Raymond C, Real F, Rimbach K, Ross C, Rotsted B, Roussez H, Ryder N, Saltarelli M, Sanders T, Santurkar S, Sastry G, Schmidt H, Schnurr D, Schulman J, Selsam D, Sheppard K, Sherbakov T, Shieh J, Shoker S, Shyam P, Sidor S, Sigler E, Simens M, Sitkin J, Slama K, Sohl I, Sokolowsky B, Song Y, Staudacher N, Such FP, Summers N, Sutskever I, Tang J, Tezak N, Thompson MB, Tillet P, Tootoonchian A, Tseng E, Tuggle P, Turley N, Tworek J, Uribe JFC, Vallone A, Vijayvergiya A, Voss C, Wainwright C, Wang JJ, Wang A, Wang B, Ward J, Wei J, Weinmann C, Welihinda A, Welinder P, Weng J, Weng L, Wiethoff M, Willner D, Winter C, Wolrich S, Wong H, Workman L, Wu S, Wu J, Wu M, Xiao K, Xu T, Yoo S, Yu K, Yuan Q, Zaremba W, Zellers R, Zhang C, Zhang M, Zhao S, Zheng T, Zhuang J, Zhuk W, Zoph B (2024) Gpt-4 technical report. URL https://arxiv.org/abs/2303.08774,2303.08774

OpenAI (2023b) New models and developer products announced at DevDay — openai.com. https://openai.com/blog/new-models-and-developer-products-announced-at-devday. Accessed 29-March-2024

Paul R, Hossain MM, Siddiq ML, Hasan M, Iqbal A, Santos JCS (2023) Enhancing automated program repair through fine-tuning and prompt engineering. URL https://arxiv.org/abs/2304.07840,2304.07840

Salerno F, Al-Kaswan A, Izadi M (2025) How much do code language models remember? An investigation on data extraction attacks before and after fine-tuning. In: 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), IEEE Computer Society, Los Alamitos, CA, USA, pp 465–477. https://doi.org/10.1109/MSR66628.2025.00080, URL https://doi.ieeecomputersociety.org/10.1109/MSR66628.2025.00080

Schäfer M, Nadi S, Eghbali A, Tip F (2024) An empirical evaluation of using large language models for automated unit test generation. IEEE Trans Softw Eng 50(1):85–10. https://doi.org/10.1109/TSE.2023.3334955

Shinn N, Cassano F, Gopinath A, Narasimhan K, Yao S (2023) Reflexion: language agents with verbal reinforcement learning. In: Proceedings of the 37th International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook, NY, USA, NIPS '23

Shin T, Razeghi Y, Logan IV RL, Wallace E, Singh S (2020) AutoPrompt: eliciting knowledge from language models with automatically generated prompts. In: Webber B, Cohn T, He Y, Liu Y (eds) Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Online, pp 4222–4235. https://doi.org/10.18653/v1/2020.emnlp-main.346, URL https://aclanthology.org/2020.emnlp-main.346/

Thoppilan R, Freitas DD, Hall J, Shazeer N, Kulshreshtha A, Cheng HT, Jin A, Bos T, Baker L, Du Y, Li Y, Lee H,Zheng HS, Ghafouri A, Menegali M, Huang Y, Krikun M, Lepikhin D, Qin J, Chen D, Xu Y, Chen Z, Roberts A, Bosma M, Zhao V, Zhou Y, Chang CC, Krivokon I, Rusch W, Pickett M, Srinivasan P, Man L, Meier-Hellstern K, Morris MR, Doshi T, Santos RD, Duke T, Soraker J, Zevenbergen B, Prabhakaran V, Diaz M, Hutchinson B, Olson K, Molina A, Hoffman-John E, Lee J, Aroyo L, Rajakumar R, Butryna A, Lamm M, Kuzmina V, Fenton J, Cohen A, Bernstein R, Kurzweil R, Aguera-Arcas B, Cui C, Croak M, Chi E, Le Q (2022) Lamda: Language models for dialog applications. URL https://arxiv.org/abs/2201.08239,2201.08239

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Lu, Polosukhin I (2017) Attention is all you need. In: Guyon I, Luxburg UV, Bengio S, Wallach H, Fergus R, Vishwanathan S, Garnett R (eds) Advances in Neural Information Processing Systems, Curran Associates, Inc., vol 30. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Wang L, Ma C, Feng X, Zhang Z, Yang H, Zhang J, Chen Z, Tang J, Chen X, Lin Y, et al. (2024) A survey on large language model based autonomous agents. Front Comput Sci 18(6):186345. https://doi.org/10.1007/s11704-024-40231-1

Wang K, Sullivan A, Khurshid S (2019) Arepair: a repair framework for alloy. In: 2019 IEEE/ACM 41st international conference on software engineering: companion proceedings (ICSE-Companion), pp 103–106. https://doi.org/10.1109/ICSE-Companion.2019.00049

Wu Y, Li Z, Zhang JM, Liu Y (2024) Condefects: A complementary dataset to address the data leakage concern for llm-based fault localization and program repair. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, FSE 2024, pp 642–646. https://doi.org/10.1145/3663529.3663815

Xia CS, Paltenghi M, Tian JL, Pradel M, Zhang L (2024) Fuzz4ALL: Universal Fuzzing with Large Language Models. In: 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), IEEE Computer Society, Los Alamitos, CA, USA, pp 1547–1559. URL https://doi.ieeecomputersociety.org/

Xia CS, Wei Y, Zhang L (2023) Automated program repair in the era of large pre-trained language models. In: Proceedings of the 45th international conference on software engineering, IEEE Press, ICSE '23, p 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

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. https://doi.org/10.1145/3540250.3549101

Xia CS, Zhang L (2023a) Conversational automated program repair. URL https://arxiv.org/abs/2301.13246,2301.13246

Xia CS, Zhang L (2024) Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2024, pp 819–831. https://doi.org/10.1145/3650212.3680323

Zhang Q, Fang C, Ma Y, Sun W, Chen Z (2023) A survey of learning-based automated program repair. ACM Trans Softw Eng Methodol 33(2). https://doi.org/10.1145/3631974

Zhang Y, Ruan H, Fan Z, Roychoudhury A (2024) Autocoderover: Autonomous program improvement. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2024, pp 1592–1604. https://doi.org/10.1145/3650212.3680384

Zheng G, Nguyen T, Brida SG, Regis G, Aguirre N, Frias MF, Bagheri H (2022) Atr: Template-based repair for alloy specifications. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2022, pp 666–677. https://doi.org/10.1145/3533767.3534369

Zhou W, Jiang YE, Li L, Wu J, Wang T, Qiu S, Zhang J, Chen J, Wu R, Wang S, Zhu S, Chen J, Zhang W, Tang X, Zhang N, Chen H, Cui P, Sachan M (2023a) Agents: An open-source framework for autonomous language agents. https://arxiv.org/abs/2309.07870,2309.07870

Zhou W, Jiang YE, Li L, Wu J, Wang T, Qiu S, Zhang J, Chen J, Wu R, Wang S, Zhu S, Chen J, Zhang W, Zhang N, Chen H, Cui P, Sachan M (2023b) Agents: an open-source framework for autonomous language agents

Authors and Affiliations

Mohannad Alhanahnah[1] · Md Rashedul Hasan[2] · Lisong Xu[2] · Hamid Bagheri[2]

✉ Mohannad Alhanahnah
Mohannad.alhanahnah@chalmers.se

Md Rashedul Hasan
mhasan6@cse.unl.edu

Lisong Xu
xu@unl.edu

Hamid Bagheri
bagheri@unl.edu

[1] Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden

[2] School of Computing, University of Nebraska-Lincoln, Lincoln, NE, USA