AdverIntent-Agent: Adversarial Reasoning for Repair Based on Inferred Program Intent

HE YE, University College London, UK
AIDAN Z.H. YANG, Carnegie Mellon University, USA
CHANG HU, Macau University of Science and Technology, China
YANLIN WANG, Sun Yat-sen University, China
TAO ZHANG, Macau University of Science and Technology, China
CLAIRE LE GOUES, Carnegie Mellon University, USA

Automated program repair (APR) has shown promising results, particularly with the use of neural networks. Currently, most APR tools focus on code transformations specified by test suites, rather than reasoning about the program's intent and the high-level bug specification. Without a proper understanding of program intent, these tools tend to generate patches that overfit incomplete test suites and fail to reflect the developer's intentions. However, reasoning about program intent is challenging.

In our work, we propose an approach called AdverIntent-Agent, based on critique and adversarial reasoning. Our approach is novel to shift the focus from generating multiple APR patches to inferring multiple potential program intents. Ideally, we aim to infer intents that are, to some extent, adversarial to each other, maximizing the probability that at least one aligns closely with the developer's original intent. AdverIntent-Agent is a multi-agent approach consisting of three agents: a reasoning agent, a test agent, and a repair agent. First, the reasoning agent generates adversarial program intents along with the corresponding faulty statements. Next, the test agent produces adversarial test cases that align with each inferred intent, constructing oracles that use the same inputs but have different expected outputs. Finally, the repair agent uses dynamic and precise LLM prompts to generate patches that satisfy both the inferred program intent and the generated tests.

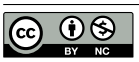
AdverIntent-Agent was evaluated on two benchmarks: Defects4J 2.0 and HumanEval-Java. AdverIntent-Agent correctly repaired 77 and 105 bugs in both benchmarks, respectively. Our work helps reduce the effort required to review patches by enabling developers to assess program intent in natural language, rather than reviewing code patches.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Program Repair, Large Language Models

ACM Reference Format:
He Ye, Aidan Z.H. Yang, Chang Hu, Yanlin Wang, Tao Zhang, and Claire Le Goues. 2025. AdverIntent-Agent: Adversarial Reasoning for Repair Based on Inferred Program Intent. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA062 (July 2025), 23 pages. https://doi.org/10.1145/3728939

Authors' Contact Information: He Ye, Department of Computer Science, University College London, UK, he.ye@ucl.ac.uk; Aidan Z.H. Yang, Software and Societal Systems Department, School of Computer Science, Carnegie Mellon University, USA; Chang Hu, School of Computer Science and Engineering, Macau University of Science and Technology, China; Yanlin Wang, School of Software Engineering, Sun Yat-sen University, China; Tao Zhang, School of Computer Science and Engineering, Macau University of Science and Technology, China; Claire Le Goues, Software and Societal Systems Department, School of Computer Science, Carnegie Mellon University, USA, clegoues@cs.cmu.edu.



ACM 2994-970X/2025/7-ARTISSTA062
https://doi.org/10.1145/3728939

1 Introduction

Automated program repair (APR) aims to reduce the manual and costly processes involved in software maintenance tasks related to bug fixing [12, 42]. A wide range of approaches have been proposed to modify programs at the source level to bring behavior in line with a given test suite. These techniques include search-based [13, 23, 26, 33, 37, 51, 64, 66, 79], semantics-based [10, 25, 41, 48, 49, 68], and learning-based [5, 6, 8, 35, 55, 74, 76, 77, 83] techniques. Not unsurprisingly, the emergence of Large Language Models (LLMs) has motivated growing interest in using program repair techniques, such as code generated by GitHub Copilot [58], Amazon Code Whisperer [78], and Replit [9]. Recent large-scale industrial deployments from Google [14] and Meta [36] provide promising initial validation that techniques developed in the lab can fruitfully move to industrial practice.

However, existing deployments (and research prototypes) remain limited in the types of bugs they successfully target. Meta's SapFix, for example, specifically targets certain types of null pointer exceptions. This allows tool designers to have more confidence that narrowly tailored patches are likely to be correct and acceptable. Acceptability in program repair has long been informed by program behavior on test suites, as tests are common in practice, and well-understood. However, overfitting to provided tests [50] is a known problem, resulting in patches that cause tests to pass, but do not generalize to the true specification, or developer intent. Researchers have proposed to tackle overfitting in traditional program repair through post-assessment approaches, such as additional heuristics [67], probability models [17, 54, 73], or test augmentation techniques [11, 75]. However, the problem is intrinsic in patch generation, and existing techniques remain limited in the complexity of defects they can empirically correctly tackle.

We argue that such traditional techniques are limited in particular by their focus on transformation types in their design (such as new templates, or new types of synthesis, to use to attempt various types of repair). They do not, by and large, attempt to reason about higher level intent of a given program. *An intent in this work is defined as the expected behavior of a function specification that developers aim to achieve.* LLM-driven techniques have recently offered an alternative perspective, potentially increasing APR expressiveness and efficiency by both (a) engaging an LLM to reason about a program's intended behavior, bringing developer intent more explicitly into the process, and (b) iteratively refining initial patch attempts towards more complex patches. This line of LLM-based work enables a self-repair loop to refine previously generated intermediate patches, which is considered state-of-the-art in the field [15, 62, 77].

This idea is promising, but incomplete: iterative, conversational repair approaches like ChatRepair [62], ITER [77], and Cigar [15] fundamentally assume that an initial repair attempt proposed by an LLM can be refined to an acceptable solution. However, LLMs often face challenges in reasoning, and the initial bug-fixing intent is not always correct [16]. In such cases, increasing the number of conversational iterations is unlikely to help [44]. Indeed, during such iterative processes, these approaches often cause conversational agents to issue refusal responses [56, 63].

In this work, we propose an approach, AdverIntent-Agent, consisting of three agents: one for reasoning about program intent, another for generating tests based on that intent, and a third for producing patches that satisfy the inferred program intent. The core idea behind AdverIntent-Agent is to adversarially reason through and construct evidence about the developer intent surrounding the buggy code, and use this reasoning to inform patch construction. At a high level, this means that AdverIntent-Agent uses LLM-based agents to (a) infer the developer intention and localize a defect, (b) generate tests to validate both those inferred intentions, and produced patches, supplementing developer-provided tests, and (c) iterates on generated patches conversationally to construct high-quality repairs. More specifically, AdverIntent-Agent:

Uses adversarial reasoning and testing to construct and identify the most likely correct intents. AdverIntent-Agent uses LLMs to infer multiple possible developer intentions with respect to a buggy function. These program intents are, by construction, intended to be *independent* and *adversarial*: that is, ideally, it should be impossible for the function, when repaired, to satisfy more than one. AdverIntent-Agent uses test generation to help demonstrate and validate that the inferred intentions are likely adversarial, assuming the intention is correct by construction and thereby sidestepping the oracle problem [1]. These tests allow for more effective automated reasoning about which intention is correct, and reduces the risk of overfitting to the developer-provided tests. The approach also increases the diversity of the considered patch pool by construction, increasing the likelihood of repair success. In practice, it is difficult to construct completely adversarial intents. Our work is an early exploration that uses test cases to quantify the degree of adversariality and to maximize the distinctiveness of intents within a limited number of attempts.

AdverIntent-Agent uses adversarial reasoning to explore multiple possible program intentions. However, multiple root causes are still possible, even given a single possible program intent. AdverIntent-Agent uses dynamic precise prompts to first ask LLMs to reason about the top-k root causes of bugs based on a given inferred intent. We use those root causes to seed additional prompts to request patches that address each root cause. This approach contrasts with previous uses of generic prompts for patch refinement [62], as it explicitly guides the generation of diverse solutions.

We implement AdverIntent-Agent and evaluate it against two benchmarks: Defects4J 2.0 [22] and HumanEval-Java [20]. On the Defects4J 2.0 benchmark, AdverIntent-Agent successfully repairs 77 bugs, outperforming related works [4, 61, 62, 74]. In HumanEval-Java, AdverIntent-Agent achieves 105 bug repairs, also surpassing the considered related works. These findings highlight the effectiveness of AdverIntent-Agent in addressing the challenges of automated program repair.

We make the following contributions:

- We devise AdverIntent-Agent, an adversarial reasoning agent for program intents, to guide the generation of diverse and high-quality patches for program repair.
- AdverIntent-Agent is extensively evaluated on two datasets, including Defects4J v2.0 and HumanEval, achieving state-of-the-art performance.
- Our work explores an original technique of adversarial program intents, where the degree of adversarial is measured by the number of adversarial test cases that can be generated.
- We consolidate and provide access to our dataset and code for future exploration https://doi.org/10.5281/zenodo.15367930.

2 Illustration

In this section, we illustrate the workflow of AdverIntent-Agent with an example. Consider Figure 1a, showing a bug in the `CountUpper` program from the Humaneval-Java [20] dataset. We use this example to demonstrate AdverIntent-Agent's initial and adversarial approach to formulate program intention as part of a repair process.

Given the buggy program and a test failure that exposes the bug, as shown in Figure 1b, AdverIntent-Agent works as follows. First, a *reasoning* agent ($\text{Agent}_{\texttt{reason}}$) starts with an initial prompt to reason about the intent of program, and its faulty statements: *Summarize the program intent, and identify faulty locations.* These responses are shown in Figure 1c. This produces an *initial* inferred intent and potentially buggy lines (first box of Figure 1c), which pinpoints the `for` loop and the `if` condition are both incorrect, as it only checks every other letter with mixed lowercase and upper case vowels.

```java
1  public int countUpper(String s) {
2   for(int i=0; i<s.length(); i+=2){
3    char c = s.charAt(i);
4    if(c=='A'||c=='e'||c=='I'||c=='o'||c=='u') {
5       count += 1;
6    }
7   return count;
```

(a) Buggy program from HumanEval-Java

```java
1  //Test case to expose the bug
2  public void testCountUpper()  {
3   int result = countUpper("aBCdEf");
4   assertEquals(1, result);
5  }
6  AssertionError:
7  Expected: 1 Actual: 0
```

(b) Initial failing test case to expose the bug

Initial Program Intent 1:	Adversarial Program Intent 2:	Adversarial Program Intent 3
The program is to count **every uppercase vowels** in 's'. The faulty code is: - for (int i=0; i< s.length(); i += 2 { - if (c == 'A'||c == 'e'||c == 'I'||c == 'o'...	The program is to count uppercase vowels **at even indices** in string 's'. The faulty code is: - if (c == 'A'||c =='e'||c =='I'||c == 'o'...	The program is to count **uppercase and lowercase** vowels at even indices in string 's'. - if (c == 'A'||c == 'e'||c == 'I'||c == 'o'...

(c) AGENT$_{reason}$ reasons three program intents.

Initial Test Cases 1 :	Adversarial Test Cases 2 :	Adversarial Test Cases 3:
Input: s = "UNIvERsiTy", Expected output: 3 Input: s = "Apple", Expected output: 1 Input: s = "bAnaNa", Expected output: 1 Input: s = "AeioOU", Expected output: 3	Input: s = "UNIvERsiTy", Expected output: 3 Input: s = "Apple", Expected output: 1 Input: s = "bAnaNa", Expected output: 0 Input: s = "AeioOU", Expected output: 2	Input: s = "UNIvERsiTy", Expected output: 3 Input: s = "Apple", Expected output: 2 Input: s = "bAnaNa", Expected output: 0 Input: s = "AeioOU", Expected output: 4

(d) AGENT$_{test}$ generates three sets of adversarial tests to measure the degree of adversarial in different program intents based on different outputs over all generated tests: the adversarial score between the first and second intent is 50%, the adversarial score between first and third intent is 75%.

Patch for Intent 1 - Overfitting:	Patch for Intent 2 - Correct:	Patch for Intent 3 - non-plausible:

- for (int i=0; i< s.length(); i += 2 {
+ for (int i=0; i< s.length(); i += 1 {
- if (c=='A'||c=='e'||c =='I'||c=='o'||c=='u') {
+ if (c=='A'||c=='E'||c =='I'||c=='O'||c=='U') {
	- if (c=='A'||c=='e'||c=='I'||c=='o'||c=='u') {
	+ if (c=='A'||c=='E'||c=='I'||c=='O'||c=='U') {
		- if (c=='A'||c=='e'||c == 'I'||c=='o'||c=='u') {
		+ if (c=='A'||c=='E'||c == 'I'||c=='O'||c=='U'
		|| c=='a'||c=='e'||c == 'i'||c=='o'|| c=='u') {

(e) AGENT$_{repair}$ generates three sets of patches to satisfy program intents and pass adversarial tests

Fig. 1. An illustrative example ADVERINTENT-AGENT to show the difference between initial and adversarial reasoning and corresponding patches and test case generated.

AGENT$_{reason}$ also constructs an *adversarial* prompt. This allows ADVERINTENT-AGENT to simultaneously explore a second possible intent. It therefore asks the LLM, "If the previous intent is NOT correct,...", producing the *adversarial* intent to count uppercase vowels at every other position (shown in the second box of Figure 1c). This second prompt identifies only the `if` condition on line 4 as problematic. ADVERINTENT-AGENT continues this process to achieve another adversarial program intent to count both uppercase and lowercase vowels at even indices (shown in the third box of Figure 1c). Note that this differentiates our approach from prior agent-based work [4, 20] by considering different possibilities for program intents.

Next, Figure 1d illustrates how the *testing* agent, AGENT$_{test}$, uses the two adversarial intents to generate new differential test cases that reflect them. The goal of AGENT$_{test}$ is twofold. First, we use tests to measure the degree of adversarial behavior in the inferred program intents. The prompt instructs the LLM to construct tests with the *same inputs* but expected different outputs for each program intent. A greater number of differentially generated tests based on the same inputs indicates a higher degree of adversarial behavior in the program intents. When the number of tests falls below a predefined threshold, we reject and re-generate the inferred program intent to

ensure all intents are adversarial to each other. In our example in Figure 1d, the adversarial score between the first and second intents is 50%, as two tests produce different results out of four total tests. In contrast, the adversarial score between intent 1 and intent 3 is 75%. Under our settings, all three intents are considered adversarial, as the threshold is set to 33.3% (the metric is introduced in subsection 3.2. Otherwise, the intents will be re-generated until they achieve the expected degree of adversarialness.

The second goal of $\text{Agent}_{\texttt{test}}$ is to generate tests that reflect each intent and specify the correct behavior for patches. Since we assume each individual intent is correct, there is theoretically a solution to an oracle problem. However, practically, ensuring assertion correctness can be challenging. We apply criticism and ranking strategies to alleviate this issue, which are discussed in detail in subsection 3.2 approach. To the best of our knowledge, AdverIntent-Agent is the first technique to generate test cases at a general level during the patch generation process.

The repair agent, $\text{Agent}_{\texttt{repair}}$, takes the output of both other agents to generate different sets of patches in Figure 1e. All generated patches are validated by both the original test cases to assess their plausibility, and on the newly generated adversarial tests reflecting inferred program intent. Our observation is that since the patches are adversarial, it is unlikely that all of them will pass the original test suite. Although incomplete, the developer-provided tests are therefore useful for testing adversarial patch plausibility. In our example, the second patch reflecting adversarial intent (middle box of Figure 1c) is correct, while the other two are either overfitting (first box of Figure 1e) or non-plausible (third box of Figure 1e). If only one is a plausible patch, this increases confidence in the inferred intent, and argues for the use of the new tests to supplement the test suite moving forward. If, on the other hand, all patches are plausible, AdverIntent-Agent increases the likelihood that at least one of them will be correct due to adversarial.

3 Approach

Figure 2 provides an overview of AdverIntent-Agent. AdverIntent-Agent takes as input a program and a set of test cases that identify a bug. It also is able to interact with an *execution environment* (by, e.g., compiling, executing, or testing the generated code) to collect information as necessitated by the process, or in response to LLM requests.

AdverIntent-Agent then consists of a three-component multi-agent framework:

- Reasoning Agent ($\text{Agent}_{\texttt{reason}}$): Reasons about multiple program intents at the function level and locates fault statements that violate such intent.
- Test Agent ($\text{Agent}_{\texttt{test}}$): Generates test cases based on the inferred program intent, to verify the adversarial degree of generated intent and act as oracles on the repair process.
- Repair Agent ($\text{Agent}_{\texttt{repair}}$): Creates patches based on the located faulty statements and program intent.

A core element of AdverIntent-Agent's approach, as illustrated in the motivating example, is the use of *adversarial* reasoning. The key to achieving reasoning is through criticism prompts to tell LLMs the previous answer is insufficient and explicitly to explore diversity, e.g., *The previous answer is not correct, consider alternatives*. Each agent thus simultaneously explores solutions to the initial repair task, as well as different adversarial formulations of the task. This approach is novel compared to related work as follows: 1) AdverIntent-Agent is the first work to reason about multiple, partially adversarial program intents, increasing the likelihood that at least one aligns with the correct intent. 2) AdverIntent-Agent is the first work, to our knowledge, to augment test generations during patch generation process, on the opposite of post patch assessments [17, 67] or specific oracle construction (e.g., crash [11]. 3) AdverIntent-Agent is explicitly guided to explore

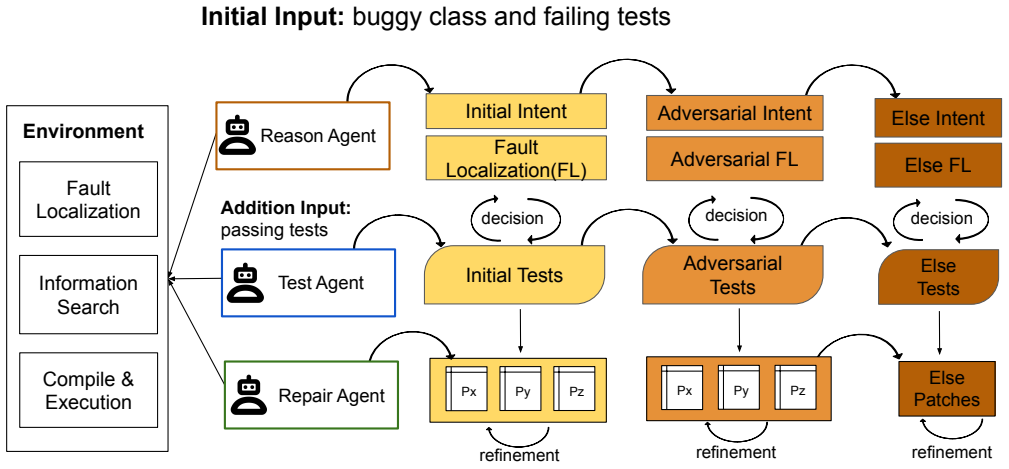


Fig. 2. Overview of AdverIntent-Agent that shows three agents that are responsible for reasoning - $\text{Agent}_{\text{reason}}$, test generation - $\text{Agent}_{\text{test}}$, and patch generation - $\text{Agent}_{\text{repair}}$. AdverIntent-Agent begins with a buggy class and failing tests as inputs, and its outputs include new intents, tests, and candidate patches.

differences and alternatives, especially compared with prior conversational approaches [62, 77], that limited themselves to iteratively refine generated patches on previous answers.

There are two kinds of *interactions* in AdverIntent-Agent: (1) Multi-Agent interactions and (2) Agent-Environment interactions. In Multi-Agent interactions, $\text{Agent}_{\text{reason}}$ infers the function intent for $\text{Agent}_{\text{test}}$ and identifies faulty statements for $\text{Agent}_{\text{repair}}$. $\text{Agent}_{\text{test}}$ generates tests based on the inferred intent to: 1) measure the degree of adversarial behavior between two inferred intents and reject one if the adversarial score falls below a predefined threshold; and 2) use these newly generated adversarial tests to validate the patches produced by $\text{Agent}_{\text{repair}}$. As to Agent-Environment interactions: $\text{Agent}_{\text{reason}}$ retrieves bug information from the file system to provide additional information requested by the LLMs. $\text{Agent}_{\text{test}}$ may retrieve sample test cases from the file system, while $\text{Agent}_{\text{repair}}$ requires the environment to validate the generated patches and collect execution feedback.

The rest of this section describes each agent in AdverIntent-Agent and Table 1 shows prompt examples throughout.

3.1 $\text{Agent}_{\text{reason}}$

The goal of $\text{Agent}_{\text{reason}}$ is two-fold: (1) infer program intent to understand expected behavior and (2) locate the buggy statements. As shown in Figure 2, $\text{Agent}_{\text{reason}}$ infers program intents sequentially, first generating an initial intent and then continuously generating adversarial ones based on previous intents, rather than generating them in parallel, to control each subsequent intent differs from the previous one.

$\text{Agent}_{\text{reason}}$ takes three inputs: (a) the source code for the presumed-buggy class; (b) the failing test cases; (c) the error messages, which identify the buggy class from the failing test cases where the function is tested but causes test failures. $\text{Agent}_{\text{reason}}$ works in two steps. First, it narrows down the buggy class to a finer granularity (functions, constructors, and variables). Second, it infers

Table 1. Prompts used in AdverIntent-Agent.

Agent	Description	Prompts
$\text{Agent}_{\text{reason}}$	Fault Localization (from class level to finer granularity)	• P1: Locate the top-3 faulty functions from the given buggy class. • P2: If the fault does not exist in the function, could you locate the variable definition or other potentially buggy classes? • P3: What if the previous answers are incorrect? What alternatives are available?
	Reason Program Intent	• P4: Given the buggy code snippet, can you reason about the program intent and locate the corresponding fault statements? • P5: What if the previous intents are incorrect? What alternatives are available?
$\text{Agent}_{\text{test}}$	Initial Test Generation	• P6: Generate N tests based on provided program intent.
	Criticism Assertions	• P7: Are there any assertions that could be wrong?
	Adversarial Test Generation	• P8: Modify the previous tests based on this intent to maintain the same inputs but generate different outputs.
	Test Prioritization	• P9: Rank tests based on confidence of assertion correctness.
$\text{Agent}_{\text{repair}}$	Dynamic Precise Prompts	• P10: What are the top-3 most likely root causes of this bug-breaking program intent? Answer in X, Y, Z. • P11: Generate a patch to repair the bug caused by X. • P12: Generate a patch to repair the bug caused by Y. • P13: Generate a patch to repair the bug caused by Z.
	Patch Refinement	• P14: Refine the patches based on the compilation and execution errors.

the intended program behavior and further refines the faulty statements based on this finer code, such as the function level. As output, $\text{Agent}_{\text{reason}}$ provides an initial set of program intents and faulty statements that exist in the current version. Each program intent is expected to be explored adversarially and distinctly.

Adversarial fault localization refined from the class level to statement granularity. $\text{Agent}_{\text{reason}}$ adversarially explores potential fault localizations by critiquing previous answers and guiding alternative exploration through three adversarial prompts. As shown in the first row of Table 1, $\text{Agent}_{\text{reason}}$ iteratively narrows the scope of faulty statements from the entire buggy class to finer granularity levels, such as functions (**P1**). Additionally, $\text{Agent}_{\text{reason}}$ explicitly guides LLMs to explore statements outside the function level, including constructors and variables, to achieve statement-level fault localization (**P2**). The LLM is also directed to examine related statements from the initially identified buggy class, as similar issues may exist in other classes, thereby enabling multi-location reasoning (**P3**). Initially, only a single class file is provided. After adversarial prompting and evaluating the error message, the LLM may request additional information. We then supply the requested files to facilitate cross-class fault localization. This design explicitly

and systematically explores a diverse set of faulty statements, complementing previous answers. Note that AGENT$_{\texttt{reason}}$ is novel in integrating LLM-based fault localization with program repair and reasoning about program behavior. Prior work either assumed perfect fault localization (e.g., RepairAgent [4]) and others [8, 21, 35, 61, 62]. Other techniques integrate with spectrum-based fault localization (see, e.g., GenProg [27] and related techniques [74, 76]). This includes agent-like AutoCodeRover [82], which considers fault localization and repair as two separate tasks and increases configuration effort.

With the identified faulty functions and statements from the first step, AGENT$_{\texttt{reason}}$ begins by generating the initial program intent by asking the LLM, "Can you reason about the program intent and locate the corresponding fault statements?" (**P4**). Based on the response, AGENT$_{\texttt{reason}}$ explicitly guides the LLM to reason about alternative program intents by posing the critical prompt, "What if the previous intents are incorrect? What alternatives are available?" (**P5**). These critical prompts can be repeated K times until K adversarial intents are generated. An example is provided earlier in Figure 1c, where the response summarizes the program intents in natural language along with the corresponding faulty statements that violate those intents. Our insights are twofold. First, AGENT$_{\texttt{reason}}$ provides multiple program intents, which are, to some degree, adversarial to each other, thus covering the entire search space of possible intents. This increases the likelihood that at least one of them is correct or close to the original developer's intent. Second, by assuming each individual intent is correct, this approach partially addresses the oracle problem, enabling test oracles to be constructed as long as they satisfy the inferred program intent.

3.2 AGENT$_{\texttt{test}}$

AGENT$_{\texttt{test}}$ takes as input program intents descriptions inferred by AGENT$_{\texttt{reason}}$, and produces as output executable test cases to reflect those behaviors. As shown in Figure 2, test generation is a sequential process in which initial tests based on the initial program intent are generated first, followed by tests based on adversarial intents in sequence. Note that each set of test cases is targeted toward validating a particular hypothesis about the function intent.

There are two objectives in AGENT$_{\texttt{test}}$. First, AGENT$_{\texttt{test}}$ generates tests to measure the adversarial degree of inferred intents. If the adversarial degree of an intent is low, that intent will be removed, and a new adversarial intent will be generated. Second, AGENT$_{\texttt{test}}$ generates additional tests to address the overfitting problem by validating the patches generated by AGENT$_{\texttt{repair}}$. Specifically, we validate whether the generated patches are overfitting to a specific intent. When one of the intents is close to the developer's intent, the generated patches are considered correct rather than overfitting. Note that this use of tests generated as an integral part of repair is, to our knowledge, novel. Overfitting patch assessment tends to be considered a post-processing step in patch generation, or left to manual effort (as in previous agent-based approaches [4, 28, 82], which conclude repair attempts upon finding plausible test-passing patches).

Initial test generation and assertion criticism. In the initial test generation, AGENT$_{\texttt{test}}$ prompts the LLMs to generate N tests based on the initial program intent (**P6**), establishing a baseline for adversarial tests. However, before modifying these initial tests to create adversarial ones, we use a critique prompt to encourage the LLMs to double-check the correctness of assertions by asking "Are there any assertions that could be wrong?" (**P7**). To ensure accurate oracles, we remove any tests for which the LLMs are uncertain about their correctness. Since not all LLM-generated test cases are executable. AGENT$_{\texttt{repair}}$ attempts to iteratively repair generated tests until they compile, up to three times. Test cases that still do not compile after three repair attempts are discarded.

Adversarial test generation. Recall that $\text{Agent}_{\text{reason}}$ produces multiple adversarial program intents. Similarly, $\text{Agent}_{\text{test}}$ aims to generate *adversarial* tests for these inferred intents. In this context, adversarial tests refer to those with the same input but with expected outputs that vary depending on which intent the program satisfies. Given the initial set of tests, adversarial prompting directs the LLM to "Modify the expected output according to the adversarial function intent" (**P8**). This allows us to align all test oracles to the same inputs and count the number of differing outputs. Inspired by mutation testing [18], we measure the degree of adversarial conflict in each intent with $adversarial\_score = \frac{different\_tests}{all\_tests}$. We define an adversarial threshold as $thres = \frac{100\%}{K}$, where K is the number of inferred intents. For example, in Figure 2, $intents\_1$ and $intents\_2$ produce different outputs in two out of four generated tests, resulting in an $adversarial\_score$ of 50%.

This design is based on the need to ensure that the test cases are theoretically mutually exclusive, covering the entire intent space, which is our core idea. We use the number of adversarial tests to measure the quality of the newly generated inferred intents. If any inferred intent falls below the threshold, we repeat the generation process until it exceeds the threshold. Consequently, we increase the likelihood that at least one intent aligns with the correct solution. In this work, we simplify the process by comparing each newly inferred intent with the first generated one.

Test Prioritization. We conduct test prioritization for each set of test cases based on the confidence of LLM assertions by asking the LLMs to "rank test cases based on confidence of assertion correctness" (**P9**). From the ranked list, we select only the top percentage of tests and filter out the rest according to the project's test configuration requirements.

3.3 $\text{Agent}_{\text{repair}}$

$\text{Agent}_{\text{repair}}$ takes as input the program intents inferred by $\text{Agent}_{\text{reason}}$ and the tests generated by $\text{Agent}_{\text{test}}$. It aims to generate patches applicable to the input program that fix the bug by ensuring that both the original and adversarial test cases pass. The key difference between $\text{Agent}_{\text{repair}}$ and prior conversation-based repair approaches is that $\text{Agent}_{\text{repair}}$ explicitly explores the diversity and differences in the root causes of the bug, whereas prior work either limits themselves in terms of diversity by iterative refining based on a single previous patch [62, 77] or increases randomness and cost by using random samples in patch generation through adjusting LLM temperatures [15, 24].

Dynamic Adversarial Prompting and Patch Refinement. Building upon the previously inferred program intent, our goal is to precisely identify the root causes of bugs to guide repair actions. The purpose is to diversify repair actions: even with the same program intent, multiple different factors may cause a bug. By precisely identifying these root causes, we ensure diversity in repair actions and prevent all generated patches from converging on a single repair strategy. To this end, $\text{Agent}_{\text{repair}}$ employs a novel dynamic and precise prompt construction (**P10**) to explicitly guide patch generation. To identify the root causes of a bug, $\text{Agent}_{\text{repair}}$ first asks the LLMs to identify the top three most likely issues, denoted as X, Y, and Z. These root causes could include "Null Checks," "Floating Point Precision Issues," "Array Index Errors," "Type Casting," and "Negative Numbers." These root cause keywords are then passed to the next phase to guide patch generation, ensuring that the process remains explicit and controlled.

Next, based on these identified root causes, $\text{Agent}_{\text{repair}}$ sends three separate prompts to guide the LLM in fixing the bug according to each root cause (**P11**, **P12**, and **P13**). Each program's intent leads to three initial patches. However, not all correct patches can be generated in one attempt. Following prior work on conversational LLM approaches [62, 77], $\text{Agent}_{\text{repair}}$ refines previous patches by executing them against the test cases. If errors are found, the error messages are collected

Table 2. Evaluation benchmarks.

Datasets	Projects	# Bugs
Defects4J 2.0 [22]	17	835
HumanEval-Java [20]	164	164
Total	181	999

and provided to the LLM for further patch refinement (**P14**). The process is configured to allow a maximum of refinement configuration reaches.

4 Evaluation

In this section, we present the experimental setup to evaluate AdverIntent-Agent. Our experiments address the following research questions:

- **RQ1** (Effectiveness): How well does AdverIntent-Agent perform overall in fixing bugs, including as compared to prior techniques?
- **RQ2** (Adversarial Reasoning): To what extent does adversarial reasoning contribute to AdverIntent-Agent's effectiveness in fault localization and patch generation?
- **RQ3** (Test Cases): To what extent do the test cases generated by $\text{Agent}_{\texttt{test}}$ contributes to the quality of the generated patches?
- **RQ4** (Cost): What is the token cost of AdverIntent-Agent in generating plausible patches?

We first evaluate the overall effectiveness of AdverIntent-Agent in repairing software bugs by comparing its performance with prior related work. Next, we conduct an ablation study to assess the contribution of individual components, such as adversarial reasoning and newly generated test cases, to the overall effectiveness. Finally, we analyze the cost of our approach to provide a comprehensive evaluation of its efficiency.

4.1 Experimental Setup

AdverIntent-Agent's core functionality is implemented in Python, interfacing with the GPT-4o API endpoint. We use a default temperature of 1 to promote patch diversity. We evaluate all generated patches on a 12-core Intel E5-2690V3 CPU at 3.50 GHz with 32GB of RAM, running on Ubuntu 20.04.3 LTS and utilizing OpenJDK Java 64-Bit Server version 1.8.0_312.

In the experiment configuration, we set the number of inferred intents to $K = 3$. We aim to locate the top three faulty functions in the provided buggy class and two alternative answers in the constructions, definitions, and other classes, for a total of five. For test generation, we take 70% of the ranked adversarial tests based on the confidence of assertions and filter out the remaining ones. For patch generation, we identify the top three causes that are inconsistent with the inferred program intent, and we allow each initial patch three rounds of refinement.

4.1.1 Benchmarks. We evaluate AdverIntent-Agent on the widely used benchmarks Defects4J 2.0 [22] and HumanEval-Java [19]. Table 2 summarizes details of the considered benchmarks. Defects4J 2.0 consists of 835 bugs from 17 real-world projects and has been widely used in related work, enabling us to conduct a fair comparison. Additionally, we evaluate HumanEval-Java. In HumanEval-Java, developers converted Python programs and their associated test cases from HumanEval into Java programs and JUnit test cases, as well as some bugs were intentionally introduced into the correct Java programs. HumanEval-Java was released after the data collection used for training GPT-3.5, providing a new benchmark for evaluating the model's ability to handle Java coding tasks.

4.1.2 Patch Quality Evaluation. We compare AdverIntent-Agent against the state-of-the-art program repair baselines on open-source projects. We measure AdverIntent-Agent and baseline effectiveness at bug repair according to standard metrics in the APR literature:

- Plausible: the number of bugs that include at least one patch that makes the original developer-written test cases pass.
- Correct: the number of bugs that include at least one correct patch, as discussed below.
- *Top@N*: A metric that evaluates whether at least one of the top-*n* generated patches is correct. A patch is considered correct if it passes both the original test cases and the automatically generated test cases.

We measure APR effectiveness in terms of the number of both plausible (test-passing) patches and correct patches. Patches that overfit to the tests but fail to generalize to the desired specification are a well-known problem in automatic program repair [45, 50]. We take multiple approaches to evaluate patch correctness. First, we assess whether the patch is plausible that makes the original test cases pass [45]. If a patch is plausible, we further identify whether it is an exact match to the developer-written ground truth patch with string comparison, ignoring whitespace differences.

For the remaining plausible patches that do not exactly match the ground truth, we employ an LLM to assess their correctness based on whether they pass LLM-generated tests for based on one specific program intent. If a patch passes its intended test case, it is considered a likely-correct patch; otherwise, it is considered likely-overfitting. Finally, we manually assess patch correctness following prior work [21, 59, 62, 68]. Two authors independently cross-check the correctness of these patches [4, 20, 21, 74]. If both authors agree that a patch is semantically equivalent to the ground truth, it is considered believed-correct. These, along with the exact match patches, form the set of correct patches.

4.1.3 Methodology. We discuss our methodology for each research question below.

Overall effectiveness (RQ1). We compare AdverIntent-Agent with related work across different repair families, including fine-tuning-based methods: SelfAPR [74], AlphaRepair [61], ITER [77]; conversation-based methods: ChatRepair [62], Cigar [15], ContrastRepair [24], and a baseline from GPT-4; as well as the agent-based approach: RepairAgent [4]. We reimplemented experiments for ChatRepair [62] based on GPT-4 and also conducted a baseline experiment using GPT-4 without execution feedback. For the remaining approaches, we report quantitative results from the corresponding papers and repositories. Plausible and correct repair results are reported as discussed above. In the experiment assuming perfect fault localization, we follow prior work [4, 24, 62] by providing the buggy statements and their contextual code to the LLMs. For a more realistic comparison, we provide a complete buggy class tested by the failing test cases, allowing the LLM to achieve finer granularity at both the function and statement levels. Existing related work evaluated on HumanEval-Java is mostly based on LLM approaches. This is because HumanEval-Java was released recently, after the data collection used for training GPT-3.5, meaning that earlier approaches were not able to evaluate it at the time they published their paper.

Adversarial Reasoning (RQ2). We study the effectiveness of adversarial reasoning by: 1) Measuring the adversarial degree between inferred intents to assess how different and distinct they are. This is based on the number of distinct test oracles generated from the same inputs. The evaluation is performed during the initial intent generation, without regenerating intents if the score falls below the threshold. 2) Comparing the alignment between inferred intents and ground-truth intents extracted from the human-written patch. We first use an LLM to generate the ground-truth intent from the patch, then assess semantic equivalence using the GPT-4o. 3) Evaluating its effectiveness as a fault localization and patch generation technique. For buggy statements spanning multiple

Table 3. Comparison with state-of-the-art on three evaluation benchmarks: Defect4J 2.0 and HumanEval-Java. A '*' indicates our reimplementation results, and a '–' indicates that the result is not available in the literature. The best performance is shown in bold.

Approach		Perfect FL		Realistic FL	
		plausible	correct	plausible	correct
Defects4J 2.0 (835 bugs)					
Fine-tuning-based	SelfAPR [74]	130	110	107	67
	AlphaRepair [61]	109	79	90	50
	ITER [77]	-	-	119	74
LLM Conversation-based	ChatRepair [62]	-	162	-	-
	Cigar [15]	-	-	**185**	69
	ContrastRepair [24]	**201**	143	-	-
Agent-based	RepairAgent [4]	186	**164**	-	-
	AdverIntent-Agent	180	141	135	**77**
HumanEval-Java (164 bugs)					
LLM Conversation-based	GPT-4 *	136	127	124	87
	ChatRepair * [62]	137	130	126	88
	ContrastRepair [24]	151	137	-	-
	Cigar [15]	-	-	**152**	102
	AdverIntent-Agent	**154**	**140**	146	**105**

locations, identifying any part of the buggy statement is considered correct, as it provides a strong starting point for patch generation. This is justified by prior work [77], which shows that subsequent buggy statements can be discovered iteratively. For patch generation, we compare the number of successful patches generated from the initial intent and the two adversarial intents to evaluate how adversarial reasoning promotes diversity. All three analyses involve substantial manual effort. To make this feasible, we conduct the study on 300 randomly selected bugs from Defects4J 2.0.

Test Generation (RQ3). In this research question, we explore to what extent newly generated tests can successfully identify and filter out overfitting patches. To evaluate this, we adopt a two-step approach that combines LLM-based selection with manual confirmation. Specifically, we employ ChatGPT-4o, which takes as input a set of plausible patches and a set of newly generated tests. First, we perform a sanity check on the generated tests by assessing their correctness and removing those with low-confidence oracles. Next, we rank the tests and select the top 70% from the ranked list. These selected tests are then executed against the plausible patches to identify likely overfitting patches, defined as those that fail the tests. For all patches flagged as likely overfitting by the LLM, we conduct a final manual analysis to determine the actual number of true positives.

Cost (RQ4). In RQ4, we conduct a detailed analysis of the cost associated with AdverIntent-Agent by comparing it to several related methods, including ChatRepair[62], Cigar [15], and RepairAgent [4], all of which rely heavily on large language model (LLM) APIs for automated program repair tasks. Our comparison focuses specifically on the token cost, which refers to the average number of tokens consumed by each approach to repair a bug in the Defect4J benchmark. We evaluate the token cost in real-world software projects, thus We do not evaluate the token cost on HumanEval-Java. By evaluating token usage in this context, we aim to provide a clear understanding of the computational resources required by each method and to highlight the efficiency and practicality of AdverIntent-Agent in real-world software engineering scenarios.

4.2 RQ1: *AdverIntent-Agent* Effectiveness

Table 3 compares the effectiveness of AdverIntent-Agent with state-of-the-art in two considered benchmark Defects4J 2.0 and HumanEval-Java. The second column lists the existing APR approaches from three categories: fine-tuning-based, LLM-based and Agent-based approaches. Their effectiveness results respectively based on perfect fault localization and realistic fault localization are given in the third and fourth column groups. The best performance number is highlighted in bold in the table. For example, in the last row of the Defects4J 2.0 benchmark, AdverIntent-Agent generated plausible patches for 135 bugs and correctly repaired 77 of them with realistic fault localization.

AdverIntent-Agent achieves the highest number of correct repairs in both the Defects4J 2.0 benchmark (77 of 835 bugs repaired) and the HumanEval-Java benchmark (105 of 164 bugs repaired) under the realistic fault localization configuration. For Defects4J 2.0 benchmark, these numbers increase to 180 and 141, respectively, when assuming perfect fault localization is known. For HumanEval-Java benchmark, these numbers increase to 154 and 140, respectively, when assuming perfect fault localization is known. Our result shows the overall effectiveness of AdverIntent-Agent.

While AdverIntent-Agent does not produce the highest number of plausible patches, this is due to its stronger validation criteria: patches are assessed against both the original and adversarial test cases, where adversarial tests effectively filter out overfitting patches. In contrast, prior work only validates against original test cases, resulting in a higher number of overfitting patches that cannot be eliminated.

We present a case study to demonstrate the effectiveness of AdverIntent-Agent. Figure 3 shows patches generated based on the same program intent for the `add_elements` program in HumanEval-Java. Figure 3a provides an initial inferred program intent and identifies the faulty statement in which the given code checks for at least two digits, which violates the inferred intent of at most two digits (see the comment in Figure 3a).

Figure 3b shows the initial patch generated by $\text{Agent}_{\texttt{repair}}$. Although this patch is intended to fix the issue, it still contains two bugs. First, the patch incorrectly identifies all numbers under *100* as having two digits, failing to account for negative numbers that may have more than two digits. Second, the initial intent and its patch fail to correct the buggy code that subtracts from, rather than adds to, the final result.

The adversarial intent, shown in Figure 3c, critiques the initial intent and patch and infers a new intent that differs from the initial intent. This leads to the generation of an adversarial patch, shown in Figure 3d. Based on the inferred program intent, AdverIntent-Agent also generates an additional test case Figure 3e to reveal the behavioral difference between the initial patch and the adversarial patch: the expected output is '-4' (making the adversarial patch pass), while the initial patch's actual output is '-16' (which fails this test case).

We can see AdverIntent-Agent provides rich information, including inferred program intent and newly generated test cases, which offer valuable resources for developers to understand the patches. This is a significant distinction from prior work, which typically provides only patches. To our knowledge, AdverIntent-Agent is the only framework that integrates inferred program intents, tests, and patches. This paradigm shift enhances how developers engage with APR patches, enabling them from understanding the patch and choosing one program intent that most closely aligns with their original intent.

An inferred program intent and located faulty statements:

Program Intent: Given an array of integers arr and an integer k, the program should return the sum of elements that have at most two digits, within the first k elements of arr. For this calculation, any negative sign should count as a digit, so negative numbers up to -99 (e.g., -5, -23) qualify as "two-digit numbers" under this rule.

Suspicious Faulty Statements

```
- if( Integer.toString (arr.get(i)).length() >= 2) { //bug: at least instead of at most
- result -= arr.get(i);
- }
```

(a) An inferred program intent and faulty statements

An initial patch:

```
//Bug: -10 will be incorrectly count
- if( Integer.toString (arr.get(i)).length() >= 2) {
+ int num = arr.get(i);
+ if (Math.abs(num) < 100) {
```

(b) An initial patch

An adversarial inferred program intent:

```
//bug 1: at least instead of at most
- if( Integer.toString (arr.get(i)).length() >= 2) {
//bug 2: Subtraction instead of addition
- result -= arr.get(i);
```

(c) An adversarial inferred program intent and faulty statements

An adversarial patch:

```
+ int num = arr.get(i);
+ String numStr = Integer.toString(num);
+ int digitCount = numStr.length();
//Check at most two digits
+ if (digitCount <= 2) {
+ result += num; //Addition instead of subtraction
+ }
```

(d) An adversarial patch

A generated test case:

```
@Test
public void testAddElements() {
ArrayList arr = new ArrayList(Arrays. asList(-1, -12, -3));
int k = 3;
// Expected output: -1 + (-3)
assertEquals(-4, add_elements(arr, k));
}
```

(e) A test case that makes (b) initial patch fail and (d) adversarial patch pass

Fig. 3. A correct patch generated by AdverIntent-Agent.

Answer to RQ1:

AdverIntent-Agent repairs 77 in Defects4J and 105 in HumanEval-Java benchmarks. This success is due to effective adversarial reasoning on program intents and dynamic, root-cause-specific prompt construction. Additionally, AdverIntent-Agent offers comprehensive resources for function reasoning, bug analysis, and test cases, providing a unique package in program repair approaches. This design shifts the paradigm by enabling developers to understand APR patches and select program intents more effectively.

4.3 RQ2: Adversarial Intent Effectiveness

We analyze the effectiveness of adversarial intents based on 300 randomly selected bugs from the Defects4J benchmark. First, we quantify the adversarial score between different inferred intents. As shown in Figure 4, the majority of adversarial scores fall within the range of 28.6% to 60.0%. The average adversarial score ranges from 43.8% to 46.5%. This data confirms the adversarial nature between different intents.

Second, we measure the extent to which the generated intents align with the expected program intent (ground truth intent). Table 4 presents the analysis, showing how well the inferred intents match the expected ground truth intent. As shown in the two last rows of Table 4, our analysis indicates that 81.7% of cases with at least one inferred intent are aligned with the ground-truth intent, while only 18.3% of cases have no correctly inferred program intents.

Specifically, it highlights the significant benefits of using multiple intents. If only a single inferred intent is used, the best single-intent match (the first intent) achieves an accuracy of 62.0% (the sum

Table 4. Program intent alignment statistics.

Scenario	Count	Percentage
All three intents are aligned	75	25.0 %
Only the first intent is aligned	38	12.7 %
Only the second intent is aligned	21	7.0 %
Only the third intent is aligned	21	7.0 %
Both the first and second are aligned	56	18.6 %
Both the first and third are aligned	17	5.7 %
Both the second and third are aligned	17	5.7 %
At least one intent is considered aligned (total)	245	81.7 %
None of the three intents is considered aligned	55	18.3 %

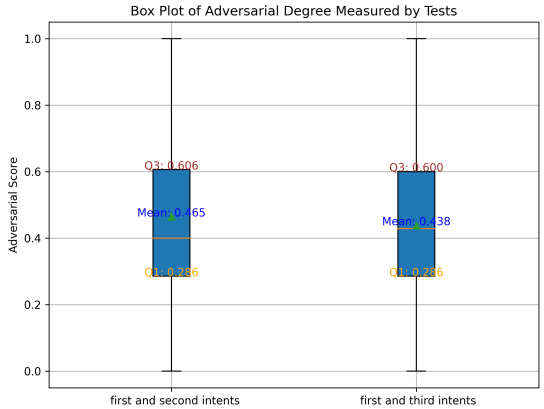


Fig. 4. Adversarial degree between the inferred intents.

of 25.0%, 12.7%, 18.6%, and 5.7%). In contrast, when leveraging multiple intents, at least one correct match is found in 81.7% of cases. The second and third inferred intents complement the first intent, accounting for the remaining 19.7% of matches.

Table 5 presents an ablation study on the effectiveness of AdverIntent-Agent with and without adversarial intents in fault localization and patch generation. The last row provides a summary of the study. We can see that with adversarial reasoning, fault localization precision increases by 13.8%, and patch generation improves by 19.4%. This confirms the effectiveness of reasoning adversarial intents for both fault localization and patch generation. We make the following implications.

Fault localization performance improves as adversarial intents encourage exploration at both function and statement levels, extending beyond the provided class. First, adversarial reasoning examines a broader and more varied search space for faulty code, not merely by generating more candidate faulty statements. Our experiment shows that simply increasing the number of candidates does not necessarily improve the final result. Second, adversarial reasoning guides the LLM to locate buggy statements outside functions, such as variable definitions, constructors, and other classes.

Patch generation performance improves as adversarial intents increase the likelihood of the search space containing a correct patch. We observe that successful patches benefit from prompts that are both adversarial and precise. The number of correctly repaired bugs increased from 36 to

Table 5. Adversarial intent effectiveness.

	Fault Localization		Patch Generation	
Projects	Initial Intent	+ Adversarial Intents	Initial Intent	+ Adversarial Intents
Total	64	75 (13.8%)	36	43 (+19.4%)

43 by incorporating adversarial intents. Adversarial intents explore a diverse range of expected program behaviors, avoiding repeated flawed attempts and effectively pinpointing root causes with accuracy.

Answer to RQ2:

Adversarial intent reasoning significantly boosts fault localization precision by 13.8% and enhances patch generation by 19.4%. Adversarial reasoning expands the search space by encouraging exploration beyond function-level constraints, effectively locating buggy statements and enhancing patch accuracy. Additionally, the agent-driven loop facilitates broader exploration by providing context, helping avoid dead-end responses.

4.4 RQ3: Tests and Overfitting

Table 6 shows the number of tests generated by $\text{AGENT}_{\texttt{test}}$ and the number of bugs with overfitting patches discarded by adversarial tests, where overfitting is associated with the original tests. The second column shows the initial number of generated tests, while the third column indicates the number of low-confidence assertions, as identified by evaluating the correctness of assertions in generated patches, and we prioritize tests based on the LLM's assertion confidence. The number of bugs with likely-overfitting patches that are discarded are given in the last column

We make the following observations from Table 6. First, AdverIntent-Agent discards likely-overfitting patches for 12 bugs in Defects4J 2.0 and 7 bugs in HumanEval-Java, confirming the effectiveness of adversarial testing in alleviating overfitting. Second, using criticism prompts to ask LLMs to double-check assertions and identify potentially incorrect ones is effective, helping us filter out 23.8% to 28.4% of low-confidence tests.

Answer to RQ3:

AdverIntent-Agent effectively discards overfitting patches for 12 bugs in Defects4J 2.0 and 7 bugs in HumanEval-Java. Test generation faces a low compilation rate, with variation between complex and simpler programs, largely due to missing dependencies.

4.5 RQ4: Cost of AdverIntent-Agent

Table 7 compares the average token cost across four approaches: Cigar, RepairAgent, ChatRepair, and AdverIntent-Agent. On average, AdverIntent-Agent uses 438K tokens per bug. Its token usage is higher than both RepairAgent and Cigar but remains lower than ChatRepair.

The main reason for the increased cost compared to RepairAgent is that AdverIntent-Agent performs realistic fault localization and automated test generation for each bug, with test generation representing the primary contributor to the overall token cost, a step that is not included in RepairAgent's workflow.

Compared to Cigar, AdverIntent-Agent 's higher token cost results from its use of advanced prompts for reasoning about program intent and generating tests, both of which are key innovations in our approach. However, if we consider only the patch generation phase, AdverIntent-Agent is actually more cost-effective than Cigar because it requests three patches per program intent and root cause, whereas Cigar samples 50 candidate patches per iteration, leading to a much greater token expenditure for Cigar in this stage.

Table 6. Generated test cases and overfitting detection.

	Test Cases			Likely-overfitting
	Initial	Low-Confidence	Rank Threshold	
Defects4J 2.0	25050	28.4%	70%	12
HumanEval-Java	4920	23.8%	70%	7

Despite these additional costs, AdverIntent-Agent remains more efficient than ChatRepair. ChatRepair's iterative approach incorporates all historical information into each prompt, causing token consumption to increase rapidly as iterations accumulate, sometimes reaching into the hundreds. By contrast, AdverIntent-Agent is designed to limit itself to three rounds of test and patch refinements, which manages overall token usage

AdverIntent-Agent achieves a balance between realism and efficiency, using more tokens for enhanced reasoning and test generation than some baselines, but remaining substantially more cost-effective than highly iterative methods like ChatRepair.

Answer to RQ4:

AdverIntent-Agent 's cost increases due to program intent reasoning and test generation, yet remains manageable because of the fewer number of iterations.

5 Related Work

5.1 Automated Program Repair

Code Transformation. Source code transformation to alter program behavior has been the main research focus of many search-based [27, 32–34, 38, 47, 59, 66, 68, 79, 80], constraint-based [40, 41, 43, 48, 51, 69], and learning-based repair [5, 8, 20, 21, 30, 31, 35, 76] techniques for a decade. The emphasis lies in constructing syntactically and semantically correct code transformations, taking into account their expressiveness and effectiveness.

Rich Information Beyond Code. Recently, more work considered information beyond code transformation, by including commit message [6], compiler and test feedback [24, 74], domain knowledge [74, 84] and iterative patch refinement [77]. Chen et al. [7] improves LMs code generation accuracy by injecting feedback messages generated by the LM itself based on its code execution results.

Conversational-based Repairs. Recent advancements in Large Language Models (LLMs) have paved the way for innovative approaches to automated program repair (APR). ChatRepair[62] pioneered a conversational repair method based on LLMs, iteratively generating patches by collecting bug context information and patch execution results. Cigar[15] further optimized the repair process by designing prompts and reboot strategies to generate diverse patches while reducing token consumption. ContrastRepair [24] takes an LLM-based iterative approach for APR. Their prompt includes a failing test and a similar passing test, where a passing test is either selected from the existing suite or generated using type-aware mutation. CHATDBG [29] shifted the focus from fully automated repair to a co-pilot for debugging, integrating LLMs with standard debuggers to enhance their capabilities.

Patch Correctness Assessment. Existing methods for assessing overfitting patches typically classify these patches separately and address them in post-processing. Prior work mainly focuses on static code analysis [52, 54, 57, 73] and test generation and execution analysis [11, 53, 65, 67].

Our work falls into the category of conversational-based repairs. Unlike previous prompting-based repair approaches, ours is the first to address the overfitting patch problem during the patch generation phase rather than as a post-generation step.

Table 7. Token cost analysis of AdverIntent-Agent.

	Cigar	RepairAgent	ChatRepair	AdverIntent-Agent
Tokens	127K	270K	467K	438K

5.2 LLM-Based Agent for Code

There is a set of parallel LLM-based agents designed for repairing software issues related to our research [71, 81, 82]. Yang et al. [71] propose SWE-Agent for repairing GitHub issues. AutoCodeRover [82] combines LLMs with advanced code search techniques to address GitHub issues through program modification or patch generation. Zhang et al. [81] introduce CodeAgent, a novel LLM-based agent framework that employs external tools for repository-level code generation, integrating five programming tools for information retrieval, code symbol navigation, and code testing. Bouzenia et al. [4] propose RepairAgent, which treats the LLM as an agent capable of autonomously planning and executing actions to fix bugs by invoking suitable tools. Our work similarly uses a decision-action-planning loop with multiple iterations and dynamic prompts based on command outputs, with the novel component of reasoning about multiple adversarial intents to explore diverse solutions.

5.3 Program Reasoning

Program intent reasoning is a critical area of research within software engineering, encompassing various techniques and methodologies aimed at understanding, analyzing, and improving software programs. One notable approach is symbolic execution [39, 60], which systematically explores program paths to identify potential errors or vulnerabilities. Formal methods provide mathematical techniques for reasoning about program correctness and behavior [2]. SpecRover by Ruan et al. [46] is a closely related work that extracts code intents via LLMs. This work shares a similar idea with ours by first extracting program intent and then using it to guide patch generation. However, our approach conceptually generates multiple diverse program intents adversarially, maximizing the likelihood that at least one inferred program is correct, whereas SpecRover infers only the most likely program intent.

In the general setting of machine learning for program reasoning, several works propose Chain-of-Thought [70], Tree-of-Thought [72] and Graph-of-Thought [3] to logical reasoning code generation tasks by breaking them down into understandable intermediate steps, enabling LLMs to handle each step individually. Unlike prior work, our approach reasons about program intent at the function specification level and incorporates multiple adversarial intents to ensure diverse solutions are explored, which is novel to our knowledge.

6 Threats to Validity

We acknowledge several potential threats to validity: 1) `Manual patch validation`. The first internal threat arises from the manual validation of believed-correct patches against the reference developer patch. To address this, we carefully examined and discussed each patch, and we make the patches publicly available. 2) `Data leakage`. Using GPT-4o for AdverIntent-Agent raises data leakage concerns due to its undisclosed training data; however, evaluation on the HumanEval-Java dataset helps strengthen internal validity. 3) `Non-deterministic of LLMs.` Non-deterministic language model outcomes pose a threat. We mitigate this by evaluating a substantial number of bugs and providing interaction logs for reproducibility and transparency. 4) `Limited programming languages.` Our experiments are limited to Java projects, which threatens external validity, as results may not generalize to other programming languages or projects. To mitigate this, we include two benchmarks, Defects4J 2.0 and HumanEval-Java, covering 181 diverse projects with various bug

types, lengths, and complexities, which strengthens the validity of our findings. 5) `Hallucinations and mistakes of LLMs.` We use LLMs for test generation based on specific program intent, but the generated oracles can sometimes be incorrect and difficult to detect. LLMs are also used to identify overfitting patches, which may result in further mistakes. We mitigate this risk by manually reviewing the results of LLM-based detection.

7 Conclusion

We present AdverIntent-Agent, a multi-agent system that advances automated program repair by focusing on inferred program intent through adversarial reasoning. Unlike traditional APR approaches, AdverIntent-Agent generates diverse program intents, adversarial tests, and patches guided by precise root causes to reduce overfitting and improve alignment with developer intent. Evaluations on Defects4J 2.0 and HumanEval-Java show that AdverIntent-Agent repairs more bugs than previous methods, underscoring its potential to enhance APR with intent-driven, developer-oriented solutions. Our work suggests a new paradigm in patch assessment that considers developer acceptance of APR patches, shifting to asking developers to analyze the patches and select the inferred program intent that best aligns with their original intent.

Acknowledgments

This work was partially supported by The Wallenberg Foundation and KAW Postdoctoral Scholarship Program - KAW 2022.0368. The computations and data handling were enabled by the supercomputing resource Berzelius provided by National Supercomputer Centre at Linköping University and the KAW foundation: 2024 NAISS GPU Grant Berzelius-2024-131.

References

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 05 (may 2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[2] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. 2009. Satisfiability modulo theories. (2009), 1–885.

[3] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 16 (March 2024), 17682–17690. https://doi.org/10.1609/aaai.v38i16.29720

[4] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. arXiv:2403.17134 [cs.SE]

[5] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. 2020. CODIT: Code Editing with Tree-Based Neural Models. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3020502

[6] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=KuPixIqPiq

[8] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2019).

[9] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*.

[10] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-Avoiding Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA)*. 8–18. https://doi.org/10.1145/3293882.3330558

[11] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 8–18.

[12] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* (2017).

[13] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA*

2019). 19–30. https://doi.org/10.1145/3293882.3330559
[14] Google. 2024. Large sequence models for software development activities. *Google* (2024).
[15] Dávid Hidvégi, Khashayar Etemadi, Sofia Bobadilla, and Martin Monperrus. 2024. CigaR: Cost-efficient Program Repair with LLMs. arXiv:2402.06598 [cs.SE]
[16] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large Language Models Cannot Self-Correct Reasoning Yet. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=IkmD3fKBPQ
[17] Elkhan Ismayilzada, Md Mazba Ur Rahman, Dongsun Kim, and Jooyong Yi. 2023. Poracle: Testing Patches under Preservation Conditions to Combat the Overfitting Problem of Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 44 (dec 2023), 39 pages. https://doi.org/10.1145/3625293
[18] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.
[19] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. arXiv:2302.05020 [cs.SE]
[20] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*.
[21] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering*.
[22] Rene Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.
[23] YoungJae Kim, Seungheon Han, Askar Yeltayuly Khamit, and Jooyong Yi. 2023. Automated Program Repair from Fuzzing Perspective. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 854–866. https://doi.org/10.1145/3597926.3598101
[24] Jiaolong Kong, Mingfei Cheng, Xiaofei Xie, Shangqing Liu, Xiaoning Du, and Qi Guo. 2024. ContrastRepair: Enhancing Conversation-Based Automated Program Repair via Contrastive Test Case Pairs. arXiv:2403.01971 [cs.SE]
[25] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. 376–379. https://doi.org/10.1145/3092703.3098225
[26] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104
[27] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/TSE.2011.104
[28] Cheryl Lee, Chunqiu Steven Xia, Jen tse Huang, Zhouruixin Zhu, Lingming Zhang, and Michael R. Lyu. 2024. A Unified Debugging Approach via LLM-Based Multi-Agent Synergy. arXiv:2404.17153 [cs.SE]
[29] Kyla Levin, Nicolas van Kempen, Emery D. Berger, and Stephen N. Freund. 2024. ChatDBG: An AI-Powered Debugging Assistant. arXiv:2403.16354 [cs.SE]
[30] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. 602–614. https://doi.org/10.1145/3377811.3380345
[31] Changshu Liu, Pelin Cetin, Yogesh Patodia, Baishakhi Ray, Saikat Chakraborty, and Yangruibo Ding. 2024. Automated Code Editing with Search-Generate-Modify. *IEEE Transactions on Software Engineering* (2024), 1–12. https://doi.org/10.1109/TSE.2024.3376387
[32] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42. https://doi.org/10.1145/3293882.3330577
[33] X. Liu and H. Zhong. 2018. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
[34] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo Italy, 166–178. https://doi.org/10.1145/2786805.2786811
[35] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair *(ISSTA 2020)*.

[36] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: automated end-to-end repair at scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice* (Montreal, Quebec, Canada) *(ICSE-SEIP '19)*. IEEE Press, 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[37] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*.

[38] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 441–444.

[39] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article 15 (oct 2018), 37 pages. https://doi.org/10.1145/3241980

[40] S. Mechtaev, J. Yi, and A. Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 448–458. https://doi.org/10.1109/ICSE.2015.63

[41] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*.

[42] Martin Monperrus. 2017. Automatic Software Repair: a Bibliography. *ACM Computing Surveys* 51 (2017), 1–24. https://doi.org/10.1145/3105906

[43] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[44] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is Self-Repair a Silver Bullet for Code Generation?. In *International Conference on Learning Representations (ICLR)*.

[45] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. 24–36. https://doi.org/10.1145/2771783.2771791

[46] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2025. SpecRover: Code Intent Extraction via LLMs. In *Proceedings of the 47th International Conference on Software Engineering (ICSE 2025)*. arXiv:2408.02232 https://arxiv.org/abs/2408.02232

[47] Seemanta Saha et al. 2019. Harnessing evolution for multi-hunk program repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 13–24.

[48] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[49] Ridwan Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 1–36.

[50] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*.

[51] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 471–482. https://doi.org/10.1109/ICSE.2015.65

[52] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in Search-based Program Repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*.

[53] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kaboré, Kui Liu, Jacques Klein, and Tegawendé F. Bissyande. 2022. Checking Patch Behaviour against Test Specification. *ACM Trans. Softw. Eng. Methodol.* (2022).

[54] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *ASE*. IEEE, 981–992. https://doi.org/10.1145/3324884.3416532

[55] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 25–36. https://doi.org/10.1109/ICSE.2019.00021

[56] Bing Wang, Yan Gao, Zhoujun Li, and Jian-Guang Lou. 2023. Know What I don't Know: Handling Ambiguous and Unknown Questions for Text-to-SQL. In *Findings of the Association for Computational Linguistics: ACL 2023*, Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, Toronto, Canada, 5701–5714. https://doi.org/10.18653/v1/2023.findings-acl.352

[57] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *ASE*. ACM.

[58] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the Copilots: Fusing Large Language Models with Completion Engines for Automated Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 172–184. https://doi.org/10.1145/3611643.3616271

[59] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[60] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing Edit Expressiveness and Search Effectiveness in Automated Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 354–366. https://doi.org/10.1145/3468264.3468600

[61] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. 959–971. https://doi.org/10.1145/3540250.3549101

[62] Chunqiu Steven Xia and Lingming Zhang. 2024. Keep the Conversation Going: 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 (ISSTA)*.

[63] Tinghao Xie, Xiangyu Qi, Yi Zeng, Yangsibo Huang, Udari Madhushani Sehwag, Kaixuan Huang, Luxi He, Boyi Wei, Dacheng Li, Ying Sheng, Ruoxi Jia, Bo Li, Kai Li, Danqi Chen, Peter Henderson, and Prateek Mittal. 2024. SORRY-Bench: Systematically Evaluating Large Language Model Safety Refusal Behaviors. arXiv:2406.14598 [cs.AI] https://arxiv.org/abs/2406.14598

[64] Qi Xin and Steven Reiss. 2019. Better Code Search and Reuse for Better Program Repair. In *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*. 10–17. https://doi.org/10.1109/GI.2019.00012

[65] Qi Xin and Steven P. Reiss. 2017. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *ISSTA*.

[66] Q. Xin and S. P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[67] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*.

[68] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 416–426. https://doi.org/10.1109/ICSE.2017.45

[69] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* (2016).

[70] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. 2023. Chain-of-Thought in Neural Code Generation: From and For Lightweight Language Models. arXiv:2312.05562 [cs.SE]

[71] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent Computer Interfaces Enable Software Engineering Language Models.

[72] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. arXiv:2305.10601 [cs.CL]

[73] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *IEEE Transactions on Software Engineering* (2021). https://doi.org/10.1109/tse.2021.3071750

[74] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. *arXiv preprint arXiv:2203.12755* (2022).

[75] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (2021), 20. https://doi.org/10.1007/s10664-020-09920-w

[76] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*.

[77] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Article 10, 13 pages. https://doi.org/10.1145/3597503.3623337

[78] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. 2023. Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. *arXiv preprint arXiv:2304.10778* (2023).

[79] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. In *IEEE Transactions on Software Engineering*.

[80] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward Better Evolutionary Program Repair: An Integrated Approach. *ACM Trans. Softw. Eng. Methodol.* 29, 1, Article 5 (jan 2020), 53 pages. https://doi.org/10.1145/3360004

[81] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. arXiv:2401.07339 [cs.SE]
[82] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. arXiv:2404.05427 [cs.SE]
[83] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. 341–353. https://doi.org/10.1145/3468264.3468544
[84] Armin Zirak and Hadi Hemmati. 2024. Improving Automated Program Repair with Domain Adaptation. *ACM Trans. Softw. Eng. Methodol.* 33, 3, Article 65 (mar 2024), 43 pages. https://doi.org/10.1145/3631972

Received 2024-10-31; accepted 2025-03-31