Improving Automated Patch Correctness Assessment by Designing LLM-Based Oracles

***Abstract*—Automated Program Repair (APR) suffers from the overfitting problem, in which generated patches pass the available test suite while remaining semantically incorrect. FixCheck, a recent Automated Patch Correctness Assessment (APCA) technique, aims to mitigate this issue by leveraging large language models (LLMs) to generate additional tests that expose incorrect patches. However, our empirical analysis shows that FixCheck's LLM-based assertion generation frequently produces semantically weak assertions or non-compiling tests, substantially limiting its effectiveness. To address these limitations, we propose six prompt engineering strategies and systematically evaluate them on 109 incorrect patches from the Defects4J benchmark using LLMs with varying capacities, including GPT-4o, GPT-4o-mini, and Llama 3.2 3B. Our results demonstrate that the effectiveness of prompt engineering is strongly model-dependent. For GPT-based models, explicitly aligning the APCA objective through role definition improves incorrect patch detection by up to 7.5%. In contrast, for Llama 3.2 3B, enforcing strict output format constraints with illustrative examples reduces non-compiling assertions and improves detection performance by up to 19.9%. Overall, this study provides practical, model-aware prompt design guidelines for building reliable LLM-based APCA systems.**

***Index Terms*—Automated Program Repair, Patch Correctness Assessment, Large Language Models, Prompt Engineering, Software Testing**

I. Introduction

Automated Program Repair (APR) techniques synthesize patches for defective programs by leveraging test suites that expose faults. [1] However, test suites typically provide only a partial specification of the intended program behavior [2]. As a result, APR-generated patches often overfit the given tests and fail to correctly resolve the underlying defect. [3] Such patches, commonly referred to as *plausible patches*, pass all available tests while remaining semantically incorrect. This overfitting problem has been widely recognized as a major obstacle to the practical adoption of APR in real-world software development [4].

To mitigate this limitation, a wide range of Automated Patch Correctness Assessment (APCA) techniques have been proposed to distinguish correct patches from overfitting ones [5]–[9]. Existing APCA approaches can be broadly categorized into dynamic techniques, which rely on runtime execution information, and static techniques, which analyze program structure without execution. While dynamic approaches often achieve higher accuracy, they incur substantial computational overhead. In contrast, static approaches are more scalable but frequently suffer from limited precision.

FixCheck, recently proposed by Molina et al. [7], represents state of the art approach to Automated Patch Correctness Assessment (APCA). It integrates static analysis, randomized testing, and large language models (LLMs) to enhance the detection of incorrect patches. Specifically, FixCheck mutates inputs of existing fault-revealing tests to generate diverse executions and leverages LLMs to infer assertions from observed runtime behaviors. These inferred assertions are subsequently used to construct new tests intended to provide concrete and interpretable evidence of patch incorrectness.

While FixCheck demonstrates the strong potential of LLM-augmented APCA and represents the current state of the art, our in-depth empirical analysis reveals inherent limitations associated with LLM-based assertion generation approaches more broadly. Rather than being specific to FixCheck, these limitations stem from fundamental challenges in using LLMs to infer semantically precise and executable assertions from dynamic program behaviors. We categorize these challenges into two primary failure modes: *Semantic Insufficiency* and *Syntactic Invalidity*.

Regarding *Semantic Insufficiency*, LLM-generated assertions frequently fail to capture the fault-triggering conditions that are essential for effective APCA. We identify two dominant manifestations of this issue. First, **shallow verification**, in which assertions encode input-dependent invariants or coincidental properties rather than true semantic correctness, allowing incorrect patches to pass when specific values happen to satisfy these invariants. Second, **semantic irrelevance**, where assertions fail at runtime but reflect expectations unrelated to the original defect, thereby producing misleading or noisy validation signals. These observations highlight a broader challenge in LLM-based APCA techniques: inferring assertions that are both semantically meaningful and directly aligned with the underlying fault remains a non-trivial problem.

Regarding *Syntactic Invalidity*, generated assertions frequently violate language-level or tool-level constraints, resulting in non-compiling or unusable tests. This issue arises recurrently across LLM-based approaches that synthesize executable code. We identify three common sources of such failures: (i) **assertion omission**, where helper method-based oracles are not recognized as valid assertions; (ii) **scope violations**, caused by extracting expressions from their valid lexical scope (e.g., loop-local variables); and (iii) **hallucinated API methods**, in which LLMs invoke non-existent but semantically plausible methods, leading to compilation errors. Collectively, these issues substantially undermine the robustness and reliability of the APCA pipeline.

Motivated by these observations, this paper investigates whether prompt engineering alone can systematically improve

LLM-based assertion generation for APCA, without modifying FixCheck's underlying architecture. We decompose prompt design into six strategies (H1–H6), each explicitly targeting one or more of the identified failure modes.

Group 1: Improving Verification Depth. The first group of strategies addresses *Semantic Insufficiency* by steering LLMs toward fault-revealing reasoning. These strategies include expanding bug-related contextual information (H1), inducing step-wise reasoning (H2), mitigating copying bias from existing assertions (H3), and explicitly aligning the LLM's role with APCA objectives through role specification (H4).

Group 2: Enhancing Code Validity. The second group of strategies targets *Syntactic Invalidity* by constraining the LLM's output space. This includes enforcing structured output formats with illustrative examples (H5) and strengthening compilation guarantees via scope-aware constraints (H6).

We evaluate these strategies on 109 incorrect patches from Defects4J using representative LLMs with varying capacities, including GPT-4o, GPT-4o-mini, and Llama 3.2 3B. Our results reveal a clear capacity-dependent pattern. For GPT-based models, explicit alignment with APCA objectives through role specification (H4) consistently improves verification depth and increases incorrect patch detection by up to 7.5%. In contrast, for the lower-capacity Llama 3.2 3B model, strict enforcement of output format constraints with examples (H5) yields the largest gains, improving detection performance by up to 19.9% while substantially reducing non-compiling assertions.

Overall, this study makes three key contributions. First, it systematically characterizes the root causes of assertion generation failures in LLM-based APCA, demonstrating that these issues arise from structural limitations of LLM inference rather than tool-specific design choices. Second, it shows that effective APCA requires explicit behavioral control beyond contextual enrichment alone. Third, it provides empirically grounded design insights indicating that optimal guidance strategies must be tailored to model capacity. Together, these findings advance the understanding of LLM-based APCA and offer practical principles for improving its robustness and effectiveness.

Contributions. This paper makes the following contributions:

- We identify and empirically characterize fundamental semantic and syntactic failure modes in LLM-based assertion generation for Automated Patch Correctness Assessment (APCA).
- We systematically design and evaluate six input guidance strategies that explicitly target these failure modes without requiring architectural changes to existing APCA frameworks or LLMs.
- We demonstrate that the effectiveness of such guidance strategies is strongly dependent on model capacity, with goal alignment via role specification benefiting GPT-based models, and structured output constraints yielding larger gains for lower-capacity models such as Llama 3.2 3B.
- We distill practical, model-aware design insights for improving the robustness and reliability of LLM-driven APCA systems.

II. Motivating Examples

This section presents representative examples observed in our evaluation that illustrate limitations of LLM-driven assertion generation for Automated Patch Correctness Assessment (APCA), thereby motivating the need for more systematic guidance mechanisms.

Figure 1 illustrates a case of shallow verification arising from input-sensitive assertions. In this example, the defect results in an incorrect computation of SSE when the precondition $y > x$ holds. The original test exposes this defect by producing a negative SSE, thereby violating the invariant SSE ≥ 0. In contrast, the automatically generated test preserves the same invariant while modifying the input values, yielding a non-negative SSE despite the defect remaining present. This example demonstrates that invariant-based assertions such as SSE ≥ 0 may pass spuriously for certain inputs and therefore fail to capture defect-relevant semantic behavior.

```
public void addData(double x, double y) {
  if (n == 0) { ...
    if (y <= x) { ybar = y; }
  } else { ... }
}
```

(a) Buggy code.

```
public void testSSENonNegative() {
  double[] y = {8915.102, ...};
  double[] x = {1.107178495E2, ...};
  ...
  assertTrue(reg.getSumSquaredErrors() >=
      0.0);
}
```

(b) Initial fault-revealing test.

```
public void testSSENonNegative() {
  double[] y = {8915.102, ...};
  double[] x = {0.7544106335656895, ...};
  ...
  assertTrue(reg.getSumSquaredErrors() >=
      0.0);
}
```

(c) Test produced by FixCheck.

Fig. 1: Shallow verification due to input-sensitive assertions.

Figure 2 illustrates a case of semantically irrelevant assertions. Inspection of the buggy code and corresponding patch reveals that the defect originates from improper sign checking using `fa / fb`, which leads to incorrect behavior when `fb` is zero. The original test exposes this defect by validating the correct output value. In contrast, the automatically generated assertion encodes an incorrect numerical expectation that is unrelated to the intended program semantics. Although the generated test fails at runtime, the failure does not correspond

to the underlying defect, thereby producing a misleading validation signal. This example underscores that assertion failures are informative only when they semantically target the fault under analysis.

```
if (fa / fb >= 0.0) {
  throw new ConvergenceException(...);
}
```

(a) Buggy code.

```
public void testMath280() {
  ...
  assertEquals(2.0, result, 1.0e-12);
}
```

(b) Initial fault-revealing test.

```
public void testMath280() {
  ...
  assertEquals(-1.645, result, 1.0e-3);
}
```

(c) Test produced by FIXCHECK.

Fig. 2: Semantically irrelevant assertions.

Figure 3 illustrates a case of assertion omission arising from limited oracle recognition in LLM-driven APCA pipelines. The original test validates program behavior using a helper method, `check(test, 10, 20)`, which implicitly encodes the expected object state. While the LLM preserves this helper-based oracle during test generation, the subsequent processing stage recognizes only explicit `assert*` statements as valid assertions and discards method-based checks. As a result, the generated test omits all verification logic. This example demonstrates that assertion omission can arise from post-processing constraints rather than deficiencies in the LLM's reasoning or generation capability.

Figure 4 illustrates a non-compiling test resulting from a scope violation introduced during assertion extraction and placement in an LLM-driven APCA pipeline. In the original test, assertions correctly reference the loop-local variable `sample` within its valid lexical scope. Although the LLM generates structurally well-formed assertions within the loop, the subsequent extraction and relocation of these assertions to the end of the test method places them outside the variable's scope. This relocation leads to a compilation error, indicating that the scope violation arises from assertion placement strategies rather than deficiencies in the LLM's code generation.

Finally, our evaluation also reveals non-compiling tests caused by hallucinated API usage. In such cases, the LLM generates assertions that appear semantically plausible but invoke methods that are not defined in the target API. For example, an assertion such as `assertFalse(dt.isLeapYear())` may be produced even when the corresponding object type does not provide an `isLeapYear()` method. Although such code is syntactically well-formed, it fails to compile, thereby reducing the practical usefulness of LLM-generated fault-revealing evidence. Taken together, these motivating examples reveal two fundamental sources of failure in LLM-driven assertion generation for APCA. First, assertions may lack semantic adequacy, either by relying on input-sensitive invariants or by targeting behavior that is unrelated to the underlying defect. Second, even when assertions are structurally reasonable, limitations in assertion extraction, placement, and output control can introduce syntactic invalidity, including assertion omission and scope violations, while LLMs may additionally produce non-compiling assertions due to API hallucinations.

These observations motivate a more principled approach to guidance design for LLM-based assertion generation. Rather than treating assertion generation as a generic test completion task, LLMs must be explicitly guided to (i) produce defect-

```
public void testWith3() {
  Partial test = createHourMinPartial();
  try {
    test.with(DateTimeFieldType
      .clockhourOfDay(), 6);
    fail();
  } catch (IllegalArgumentException ex) {}
  check(test, 10, 20);
}
```

(a) Initial fault-revealing test.

```
public void testWith3() {
  Partial test = createHourMinPartial();
  try {
    test.with(DateTimeFieldType
      .clockhourOfDay(), 6);
    fail();
  } catch (IllegalArgumentException ex) {}
  // check() omitted by the tool
}
```

(b) Test produced by FIXCHECK.

Fig. 3: Assertion omission caused by helper-based oracles.

```
public void testMath1021() { ...
  for (int i = 0; i < 100; i++) {
    final int sample = dist.sample();
    Assert.assertTrue(0 <= sample);
    Assert.assertTrue(sample <= n);
  }
}
```

(a) Initial fault-revealing test.

```
public void testMath1021() { ...
  for (int i = 65; i < 100; i++) {
    final int sample = dist.sample();
  }
  Assert.assertTrue(0 <= sample);
  Assert.assertTrue(sample <= n);
}
```

(b) Test produced by FIXCHECK.

Fig. 4: Non-compiling assertions caused by scope violations.

relevant verification logic and (ii) respect the structural and syntactic constraints imposed by the testing framework. In the next section, we formalize these insights and introduce a set of guidance strategies designed to systematically address both semantic and syntactic failure modes.

III. Approach

A. Overview of Guidance Strategy Design

Figure 5 provides an overview of the guidance strategy design space explored in this study. Building on the failure modes identified in the motivating examples, we organize our investigation around two recurring challenges in LLM-based assertion generation for APCA: *semantic insufficiency* and *syntactic invalidity*. Guided by these characteristics, we formulate six concrete hypotheses (H1–H6) that capture distinct ways of steering LLM behavior to mitigate the observed failures. These hypotheses are grouped into two complementary categories, reflecting whether they primarily target verification depth or code validity.

Group 1: Improving Verification Depth. This group targets *semantic insufficiency* in LLM-generated assertions, including shallow verification driven by input-sensitive invariants and semantically irrelevant assertions that fail to capture the underlying defect. The proposed strategies aim to steer LLMs toward a more faithful interpretation of bug semantics and patch intent during assertion generation, thereby improving the relevance and discriminative power of the resulting tests. This group comprises the following hypotheses: (H1) *Expansion of Bug-Related Context*, (H2) *Inducement of Step-wise Reasoning*, (H3) *Mitigation of Copying Bias*, and (H4) *Explicit Alignment with APCA Objectives*.

Group 2: Enhancing Code Validity. This group addresses *syntactic invalidity* in generated assertions. It encompasses failures arising from limitations in assertion extraction and placement, such as assertion omission in the presence of helper-based oracles and references to variables outside their valid lexical scope. It also includes errors introduced by the LLM itself, such as hallucinated API invocations. While such issues could, in principle, be resolved through substantial redesign of the underlying APCA pipeline, doing so would require non-trivial architectural changes. Instead, we examine whether these failures can be mitigated through guidance strategies that constrain the LLM's output toward syntactically valid and compilable assertions. This group includes the following hypotheses: (H5) *Output Format Enforcement* and (H6) *Scope-Aware Constraints*.

The following subsections describe each hypothesis and its corresponding guidance strategy in detail.

B. Group 1: Improving Verification Depth

1) Hypothesis 1: Expansion of Bug-Related Context: Providing explicit bug-related contextual information enables LLMs to better interpret patch semantics, thereby increasing the likelihood of generating defect-relevant assertions. In typical LLM-based APCA settings, assertion generation is guided primarily by test code structure, often constructed from a fault-revealing test and a passing test without assertions. While such prompts convey syntactic information about the test harness, they provide limited insight into why the test failed or which program components were affected by the patch. As a result, the model may lack sufficient semantic grounding and produce shallow or semantically irrelevant assertions, a phenomenon we characterize as *semantic insufficiency*. Prior work suggests that enriching prompts with bug-related contextual information can improve LLM reasoning about patch behavior in APCA tasks. Motivated by these findings, we hypothesize that augmenting assertion generation with explicit bug-related context can steer LLMs toward assertions that more directly target the underlying defect. To operationalize this hypothesis, we augment the assertion generation prompt with two forms of bug-related contextual information. First, we include **failure details** that describe how the original fault-revealing test manifested the defect. Second, we provide information about **patch-modified files**, highlighting program components directly affected by the patch. In our evaluation, both forms of information are obtained from metadata available in the Defects4J benchmark. In practical development settings, analogous information can be readily derived from standard artifacts such as version control diffs and test failure traces.

```
==== Failure in the original buggy version
    ====
- org.jfree.chart.renderer.category.junit.
    AbstractCategoryItemRendererTests
  ::test2947660
  --> junit.framework.AssertionFailedError:
      expected:<1> but was:<0>

==== Files modified by the patch ====
org.jfree.chart.renderer.category.junit.
    AbstractCategoryItemRendererTests
```

Fig. 6: Example of an assertion generation prompt augmented with bug-related contextual information (H1).

By explicitly associating failure symptoms with patch-modified code regions, this strategy is expected to reduce reliance on shallow, input-sensitive assertions and promote verification logic that more accurately reflects defect-relevant semantics, while remaining applicable across both benchmark-driven evaluations and real-world development workflows.

2) Hypothesis 2: Inducement of Step-wise Reasoning: Encouraging structured internal reasoning enables LLMs to better interpret patch semantics, thereby increasing the likelihood of generating defect-relevant assertions. In many LLM-based APCA settings, assertion generation is framed as a direct code synthesis task, with limited guidance on how the model should reason about test failures or patch behavior. As a result, LLMs may rely on shallow heuristics or surface-level patterns, leading to assertions that lack semantic depth. Prior work in APCA and related program analysis tasks

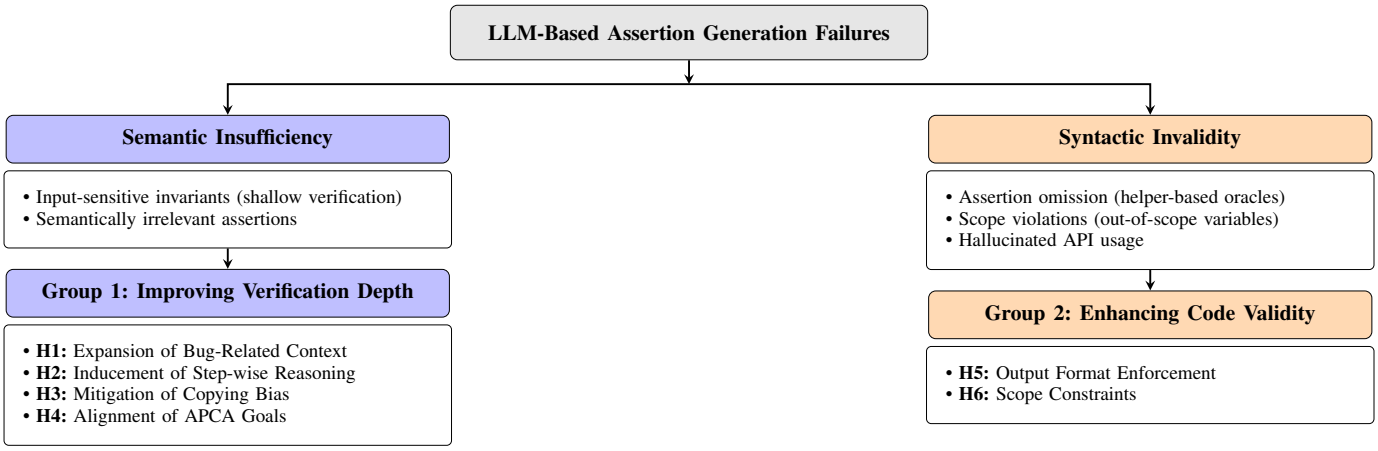

Fig. 5: Overview of prompt engineering strategies addressing FIXCHECK's failure modes.

suggests that inducing structured reasoning can improve an LLM's ability to analyze patch behavior and its semantic implications. Motivated by these findings, we hypothesize that explicitly encouraging step-wise internal reasoning can help LLMs form a more coherent understanding of defect semantics before producing assertions. To instantiate this hypothesis, we introduce an instruction that encourages the model to internally reason about the relationship between the observed failure and the patch behavior prior to assertion generation, while preserving the requirement that the final output consists solely of executable test code. This design allows structured reasoning to inform generation without altering the external format or execution semantics of the produced assertions.

```
Think step-by-step internally,
but do NOT reveal your reasoning.
```

Fig. 7: CoT-inducing instruction added to the system prompt (H2).

We evaluate two variants of this strategy. (i) **CoT-only** (H2-a), which introduces step-wise reasoning guidance on top of the baseline assertion generation prompt, and (ii) **Context + CoT** (H2-b), which combines step-wise reasoning guidance with the inclusion of explicit bug-related context (H1).

By promoting structured internal reasoning, this strategy is expected to mitigate shallow, input-sensitive assertions and improve the semantic relevance of generated oracles, particularly when combined with richer bug-related contextual information.

3) Hypothesis 3: Mitigation of Copying Bias: Removing the original fault-revealing test from the prompt reduces copying bias and encourages LLMs to generate more novel, defect-sensitive assertions, albeit at the potential cost of losing useful structural guidance. We observe that when the original fault-revealing test is included in the prompt, LLMs frequently reproduce assertions that are identical or nearly identical to the original oracle. This tendency, which we refer to as *copying bias*, limits verification novelty and often fails to introduce new evidence of patch incorrectness, thereby contributing to semantic insufficiency. To isolate the impact of copying bias from other confounding factors, we explore the effect of removing the original fault-revealing test from the assertion generation context. To mitigate copying bias, we remove the entire original fault-revealing test from the test prefix, including its setup, execution logic, and assertions. The LLM is then tasked with generating assertions independently, which are appended to the synthesized test. To disentangle the effect of copying bias mitigation from the loss of structural and semantic guidance, we evaluate three variants: (i) **Fault-Revealing Test Removal Only** (H3-a), (ii) **Removal with Output Constraints** (H3-b, incorporating H5), and (iii) **Removal with Bug-Related Context** (H3-c, incorporating H1).

This hypothesis aims to assess whether the inclusion of fault-revealing tests primarily induces copying bias or instead provides essential structural and semantic scaffolding for effective assertion generation. The results help clarify the trade-off between encouraging assertion novelty and preserving informative guidance.

4) Hypothesis 4: Alignment of APCA Goals via Role Definition: Explicitly aligning the LLM's assigned role with the objective of Automated Patch Correctness Assessment (APCA) increases the likelihood of generating fault-revealing assertions, rather than conservative assertions that trivially pass. In many LLM-based assertion generation settings, the model is instructed to synthesize executable unit tests without an explicit specification of the APCA objective. As a result, the LLM tends to favor execution-safe assertions that minimize the risk of test failures. While such behavior is desirable in general test generation, it directly conflicts with APCA's goal of intentionally exposing incorrect patches. Without explicit goal alignment, the model lacks a clear incentive to generate aggressive, defect-sensitive assertions, often resulting in shallow verification. To address this misalignment, we redefine the system-level instruction to explicitly specify the APCA task context and the desired failure semantics, emphasizing

```
Output ONLY valid Java JUnit assertions.
No comments, imports, or extra text.

Examples:
assertEquals(expected, actual);
assertTrue(condition);
assertFalse(condition);
```

Fig. 9: Output format constraints with examples (H5).

that assertion generation should prioritize detecting incorrect patches rather than ensuring test passability. This role specification serves to align the model's generation behavior with the underlying objective of APCA.

```
You are an expert Java unit-test assistant
for Automated Patch Correctness Assessment.
Your goal is to generate fault-revealing
assertions that FAIL when the bug exists
and PASS when it is fixed.
```

Fig. 8: APCA-aligned role definition (H4).

By shifting the optimization focus from execution safety toward defect detection, this strategy is expected to encourage more aggressive, defect-sensitive assertions and thereby mitigate semantic insufficiency in LLM-generated verification logic.

C. Group 2: Enhancing Code Validity

1) Hypothesis 5: Output Format Enforcement: Imposing strict output format constraints reduces syntactic errors and increases the proportion of compilable assertions generated by LLMs. As demonstrated in the motivating examples, LLM-based assertion generation pipelines may produce tests that lack effective verification logic due to limitations in downstream processing or ambiguity in output structure, even when the model captures relevant oracle information. Moreover, LLM outputs often contain extraneous text, malformed syntax, or non-code elements that lead to compilation failures. These issues are particularly pronounced for lower-capacity models, where loosely specified output requirements tend to exacerbate syntactic invalidity and reduce the usability of generated tests. To address these challenges, we enforce a strict output format that explicitly constrains the structure of generated assertions and provide illustrative examples of valid JUnit assertion statements. This design narrows the LLM's output space toward syntactically well-formed and directly executable code.

By constraining generation to valid assertion constructs, this strategy is expected to mitigate assertion omission and formatting-related compilation failures, thereby improving the robustness and reliability of LLM-generated assertions.

2) Hypothesis 6: Scope-Aware Constraints: Explicitly constraining variable scope and permissible API usage reduces non-compiling assertions caused by out-of-scope references and hallucinated methods. As illustrated in the motivating examples, non-compiling tests frequently arise when assertions reference variables outside their valid lexical scope or invoke methods that are not defined in the target API. A representative failure occurs when assertions that are valid within a loop are relocated to a different context, rendering previously in-scope variables inaccessible. More broadly, such scope and API violations constitute a major source of syntactic invalidity in LLM-generated assertions and substantially limit their practical usefulness. To mitigate these issues, we introduce explicit negative constraints that restrict assertion generation to variables within the current lexical scope and disallow the use of API methods that are not present in the available program context. These constraints are designed to guide the LLM away from common sources of compilation failure without requiring changes to the underlying assertion extraction or placement mechanisms.

```
Use only variables in scope at the end
of the test method.
Do not invent methods or APIs.
Do not reference uninitialized objects.
```

Fig. 10: Scope-aware constraints (H6).

Expected Effect: By preventing out-of-scope references and hallucinated API usage, this strategy is expected to substantially reduce compilation failures and improve the reliability of generated assertions, while remaining compatible with existing APCA pipelines.

IV. Evaluation

A. Experimental Setup

1) Dataset: We evaluate our guidance strategies on a Defects4J [10]-based dataset of APR-generated patches that has been widely adopted in prior studies on Automated Patch Correctness Assessment (APCA), including PATCH-SIM and Shibboleth. This dataset was originally curated for evaluating PATCH-SIM and subsequently reused in Shibboleth, and thus consists of patches whose correctness has been assessed using established APCA techniques. The dataset contains a total of 139 patches, comprising 30 correct and 109 incorrect patches. In this study, we focus exclusively on the **109 incorrect patches**. This choice aligns with the common APCA evaluation setting, where the primary objective is to assess a technique's ability to expose incorrect patches by generating additional fault-revealing evidence, rather than to independently classify patch correctness. The patches correspond to 74 real-world bugs from four Defects4J projects: `chart`, `lang`, `math`, and `time`. They were generated by five widely used Automated Program Repair (APR) tools: jGenProg [11] and jKali [12], Nopol [13] (2015 and 2017 versions), ACS [14], and HDRepair [15]. Together, these tools cover a diverse range of repair paradigms, including search-based, constraint-based, and statistical approaches.

TABLE I: Distribution of incorrect patches and detection results by PATCH-SIM and Shibboleth.

Project	Incorrect	Detected (PATCH-SIM)	Detected (Shibboleth)
Chart	23	13	21
Lang	10	5	10
Math	63	35	58
Time	13	9	12
Total	109	62	101

As shown in Table I, PATCH-SIM detects 62 out of 109 incorrect patches (56.9%), while Shibboleth detects 101 out of 109 (92.6%). In our evaluation, we apply the LLM-based APCA approach to the subsets of incorrect patches identified by each existing APCA technique, and assess whether it can generate at least one additional fault-revealing test that provides actionable evidence of patch incorrectness beyond what is already detected.

2) Baseline: Our baseline follows the original assertion generation configuration proposed in prior work. For each potentially incorrect patch, the input consists of an initial fault-revealing test t associated with the corresponding Defects4J bug, along with its test prefix tp. As in the original setup, the failure trace of t is used during the final test selection and prioritization stage. However, the baseline prompt itself does not incorporate additional semantic context, explicit goal alignment, or constraints on output format or compilation validity.

3) Guidance Strategies: Starting from the baseline configuration, we augment the prompt with two groups of guidance strategies (H1–H6), each designed to address a distinct class of failure modes identified in Section II. Group 1 targets *semantic insufficiency*, including shallow or defect-irrelevant assertions, whereas Group 2 targets *syntactic invalidity*, such as non-compiling assertions.

Group 1: Improving Verification Depth.: This group focuses on improving the semantic relevance and discriminative power of generated assertions. It includes:

- (H1) *Expansion of Bug-Related Context*
- (H2) *Inducement of Step-wise Reasoning*
- (H3) *Mitigation of Copying Bias*
- (H4) *Explicit Alignment with APCA Objectives*

Group 2: Enhancing Code Validity.: This group focuses on improving the syntactic correctness and executability of generated assertions. It includes:

- (H5) *Output Format Enforcement*
- (H6) *Scope-Aware Constraints*

4) Models and Runs: We evaluate three representative large language models with different capacities: GPT-4o [16], GPT-4o-mini [17], and Llama 3.2 3B [18]. For each patch, the assertion generation procedure produces $K = 10$ candidate test prefixes. Each experimental configuration is repeated three times using different random seeds, and we report the mean results across runs.

B. Evaluation Metrics

Let $\mathcal{P}_A$ denote the set of patches classified as *incorrect* by a given Automated Patch Correctness Assessment (APCA) technique A (e.g., PATCH-SIM or Shibboleth). Following common APCA evaluation practice, only these patches are considered as candidates for complementary analysis. In our dataset, $|\mathcal{P}_{\text{PATCH-SIM}}| = 62$ and $|\mathcal{P}_{\text{SHIBBOLETH}}| = 101$.

For a patch p, let $\mathcal{T}(p)$ denote the set of tests generated under a given guidance strategy and LLM configuration. Each generated test is executed on the patched program and classified into one of three mutually exclusive outcomes: PASS, AF (assertion-failing), or NC (non-compiling).

Failing-Test Selection and Prioritization. Assertion-failing tests are prioritized based on the similarity between their failure behavior and that of the initial fault-revealing test t. Let f_t and f_τ denote the failure traces of t and a generated failing test τ, respectively. Each trace is converted into a string representation, and a normalized similarity score is computed using the Levenshtein distance:

$$\text{score}(\tau) = 1 - \frac{\text{Levenshtein}(s_\tau, s_t)}{m}, \tag{1}$$

where $m = \max(|s_\tau|, |s_t|)$. A higher score indicates greater similarity between the failure behavior of τ and that of the original fault-revealing test.

Failing tests whose similarity score falls below a predefined threshold κ (set to $\kappa = 0.4$) are discarded. The remaining tests are ranked in descending order of $\text{score}(\cdot)$, and the highest-ranked test is reported as evidence of patch incorrectness.

Fault-Revealing Patch. A patch p is considered *fault-revealing* if at least one generated assertion-failing test exhibits a failure trace sufficiently similar to that of the initial fault-revealing test. Formally,

$$\text{FR}(p) \triangleq \exists\, \tau \in \mathcal{T}(p) \text{ s.t. } \tau \in \text{AF} \wedge \text{score}(\tau) \geq \kappa. \tag{2}$$

1) APCA Complementation Effectiveness: Following prior work on LLM-based APCA, we measure the ability of an LLM-driven approach to complement an existing APCA technique A as the proportion of patches in $\mathcal{P}_A$ for which at least one additional fault-revealing test is generated.

$$\text{Comp}(A) = \frac{|\{\, p \in \mathcal{P}_A \mid \text{FR}(p) \,\}|}{|\mathcal{P}_A|} \times 100. \tag{3}$$

2) Precision and Recall Analysis: To assess whether improvements in fault-revealing capability introduce false positives, we additionally report patch-level precision and recall over the full set of Defects4J patches. A patch is treated as a positive prediction if the LLM-based APCA approach generates fault-revealing evidence for that patch.

Let $\mathcal{P}_{\text{INC}}$ and $\mathcal{P}_{\text{COR}}$ denote the sets of incorrect and correct patches, respectively. Recall and precision are defined as:

$$\text{Recall} = \frac{|\{\, p \in \mathcal{P}_{\text{INC}} \mid \text{FR}(p) \,\}|}{|\mathcal{P}_{\text{INC}}|}, \tag{4}$$

$$\text{Precision} = \frac{|\{\, p \in \mathcal{P}_{\text{INC}} \mid \text{FR}(p) \,\}|}{|\{\, p \mid \text{FR}(p) \,\}|}. \tag{5}$$

TABLE II: Effectiveness of prompt strategies across LLMs. Parentheses denote absolute change from the FixCheck baseline. **Bold** indicates the best score per model and metric.

Group	Strategy	GPT-4o-mini		GPT-4o		Llama 3.2 3B	
		PATCH-SIM	**Shibboleth**	**PATCH-SIM**	**Shibboleth**	**PATCH-SIM**	**Shibboleth**
	Baseline	47.87	55.97	46.77	52.27	27.93	38.83
Group 1 Verification Depth	H1	48.93 (+1.06)	55.67 (-0.30)	46.27 (-0.50)	50.80 (-1.47)	26.87 (-1.06)	36.70 (-2.13)
	H2-a	48.40 (+0.53)	55.67 (-0.30)	45.13 (-1.64)	50.73 (-1.54)	29.57 (+1.64)	37.30 (-1.53)
	H2-b	49.47 (+1.60)	56.57 (+0.60)	45.70 (-1.07)	50.47 (-1.80)	25.80 (-2.13)	36.70 (-2.13)
	H3-a	36.57 (-11.30)	41.93 (-14.00)	36.03 (-10.74)	45.27 (-7.00)	24.73 (-3.20)	33.93 (-4.90)
	H3-b	47.87 (+0.00)	50.80 (-5.17)	43.57 (-3.20)	49.23 (-3.04)	34.40 (+6.47)	44.63 (+5.80)
	H3-c	36.03 (-11.84)	44.03 (-11.94)	41.37 (-5.40)	45.90 (-6.37)	24.73 (-3.20)	34.56 (-4.27)
	H4	**55.37 (+7.50)**	**62.40 (+6.43)**	**49.47 (+2.70)**	**58.13 (+5.86)**	28.47 (+0.54)	38.50 (-0.33)
Group 2 Code Validity	H5	47.33 (-0.54)	53.20 (-2.77)	42.43 (-4.34)	48.60 (-3.67)	**47.87 (+19.90)**	**55.03 (+16.20)**
	H6	47.33 (-0.54)	54.73 (-1.24)	43.00 (-3.77)	47.10 (-5.17)	25.27 (-2.66)	35.77 (-3.06)

3) Assertion Generation Quality: To analyze the factors underlying improvements or degradations in $\mathrm{Comp}(A)$, we examine the distribution of outcomes among the generated tests. Let N denote the total number of generated tests, and let N_{PASS}, N_{AF}, and N_{NC} denote the number of tests classified as PASS, AF (assertion-failing), and NC (non-compiling), respectively. We compute:

$$\mathrm{PassRate} = \frac{N_{\text{PASS}}}{N} \times 100, \tag{6}$$

$$\mathrm{AFRate} = \frac{N_{\text{AF}}}{N} \times 100, \tag{7}$$

$$\mathrm{NCRate} = \frac{N_{\text{NC}}}{N} \times 100. \tag{8}$$

PASS is typically associated with semantic insufficiency, AF includes both defect-relevant and irrelevant assertion failures, and NC reflects syntactic invalidity such as malformed assertions, hallucinated APIs, or scope violations. Together, these metrics provide both outcome-level performance and mechanism-level diagnostics for evaluating FIXCHECK's APCA complementation capability.

C. RQ1: Overall Impact of Prompt Design Elements

Table II summarizes the overall impact of guidance design elements across different large language models (LLMs). The results reveal a clear model-dependent pattern: no single strategy consistently outperforms the baseline across all models, and the effectiveness of guidance varies substantially with model capacity. For GPT-based models (GPT-4o and GPT-4o-mini), explicit alignment with the APCA objective (H4) yields the most consistent improvements. In particular, GPT-4o-mini achieves the largest gains under H4, with increases of +7.50 for PATCH-SIM and +6.43 for Shibboleth, while GPT-4o exhibits the same trend with more moderate improvements (+2.70 and +5.86, respectively). These results indicate that high-capacity models benefit most from guidance that clarifies the fault-revealing objective, enabling them to generate more defect-relevant assertions.

In contrast, for the lower-capacity Llama 3.2 3B model, output format enforcement with illustrative examples (H5) leads to substantially larger improvements than semantic or reasoning-oriented strategies, yielding gains of +19.9% for PATCH-SIM and +16.2% for Shibboleth. Other strategies show limited or inconsistent effects for this model. This suggests that, for lower-capacity models, constraining the output space to ensure syntactic validity plays a more critical role than providing additional semantic guidance. Taken together, these findings demonstrate that effective guidance for LLM-based APCA is inherently model-aware. High-capacity models primarily benefit from explicit goal alignment that leverages their semantic reasoning capabilities, whereas lower-capacity models benefit more from strong syntactic and structural constraints. This result underscores the importance of tailoring guidance strategies to model characteristics, rather than adopting a one-size-fits-all approach.

D. RQ2: Effects on Verification Depth

This research question investigates how guidance design elements influence the *depth of verification* achieved by generated tests. Motivated by the shallow, input-sensitive verification illustrated in Figure 1, we operationalize verification depth using the outcome distribution of generated tests: PASS, AF (assertion-failing), and NC (non-compiling) in Table III.

Based on the results of RQ1, we focus on explicit alignment with the APCA objective (H4), which yields the most consistent improvements for GPT-based models. Under H4, both GPT-4o-mini and GPT-4o exhibit a clear behavioral shift: the proportion of PASS outcomes decreases, while the proportion of AF outcomes increases. For GPT-4o-mini, the AF rate increases by 7.84 points and the PASS rate decreases by 23.01 points; for GPT-4o, AF increases by 11.44 points and PASS decreases by 13.37 points. Relative to the invariant-driven behavior observed in Figure 1, this shift indicates that H4 promotes assertions that more directly target defect-relevant semantics.

TABLE III: Outcome distribution of generated assertions. Parentheses denote absolute change from the FixCheck baseline.

Group	Strategy	GPT-4o-mini			GPT-4o			Llama 3.2 3B		
		Pass	NC	AF	Pass	NC	AF	Pass	NC	AF
	Baseline	38.65	6.96	28.65	43.60	4.53	26.37	55.44	7.52	11.90
Group 1 Verification Depth	H1	38.96 (+0.31)	9.62 (+2.66)	25.55 (-3.10)	41.20 (-2.40)	5.91 (+1.38)	27.98 (+1.61)	54.24 (-1.20)	8.01 (+0.49)	12.60 (+0.70)
	H2-a	38.63 (-0.02)	6.46 (-0.50)	29.94 (+1.29)	43.42 (-0.18)	4.21 (-0.32)	27.25 (+0.88)	55.90 (+0.46)	6.90 (-0.62)	12.34 (+0.44)
	H2-b	41.67 (+3.02)	7.63 (+0.67)	24.94 (-3.71)	41.26 (-2.34)	5.61 (+1.08)	28.51 (+2.14)	53.45 (-1.99)	9.47 (+1.95)	11.87 (-0.03)
	H3-a	30.15 (-8.50)	25.82 (+18.86)	18.60 (-10.05)	31.26 (-12.34)	20.18 (+15.65)	23.30 (-3.07)	52.91 (-2.53)	14.62 (+7.10)	7.34 (-4.56)
	H3-b	36.99 (-1.66)	12.02 (+5.06)	26.49 (-2.16)	42.39 (-1.21)	5.78 (+1.25)	27.56 (+1.19)	16.99 (-38.45)	43.10 (+35.68)	14.71 (+2.81)
	H3-c	31.93 (-6.72)	20.38 (+13.42)	22.19 (-6.46)	32.83 (-10.77)	18.62 (+14.09)	23.19 (-3.18)	54.03 (-1.41)	14.24 (+6.72)	6.34 (-5.56)
	H4	15.64 (-23.01)	20.67 (+13.71)	36.49 (+7.84)	30.23 (-13.37)	6.70 (+2.17)	37.81 (+11.44)	55.96 (+0.52)	5.17 (-2.35)	13.98 (+2.08)
Group 2 Code Validity	H5	43.74 (+5.09)	4.30 (-2.66)	26.93 (-1.72)	46.46 (+2.86)	3.30 (-1.23)	24.83 (-1.54)	33.01 (-22.43)	21.05 (+13.53)	20.97 (+9.07)
	H6	39.39 (+0.74)	5.88 (-1.08)	29.85 (+1.20)	47.22 (+3.62)	3.39 (-1.14)	24.68 (-1.69)	54.42 (-1.02)	9.59 (+2.07)	10.64 (-1.26)

```
public void testSSENonNegative() {
  SimpleRegression reg = new SimpleRegression
      ();
  assertTrue(reg.getSumSquaredErrors() >= 0.0)
      ;
  assertEquals(0.0, reg.getSumSquaredErrors(),
       1e-10); // added by H4
}
```

Fig. 11: Test produced by H4.

This increase in verification depth is accompanied by a higher proportion of NC outcomes, reflecting a trade-off between aggressiveness and compilability. For GPT-4o, the NC rate increases modestly (+2.17), whereas GPT-4o-mini exhibits a larger increase (+13.71), consistent with failures such as hallucinated API invocations, malformed assertions, or scope violations. Accordingly, the rise in NC should be interpreted as a consequence of more aggressive, defect-sensitive assertion generation rather than an improvement in code validity.

TABLE IV: Patch-level precision and recall under the FIXCHECK baseline and role alignment (H4).

Model	Setting	Precision	Recall
GPT-4o-mini	Baseline	0.90	0.58
GPT-4o-mini	H4 (Role Alignment)	0.88	0.65
GPT-4o	Baseline	0.91	0.54
GPT-4o	H4 (Role Alignment)	0.90	0.60

Despite this trade-off, patch-level effectiveness improves. As shown in Table IV, precision remains high under H4 (0.88 for GPT-4o-mini and 0.90 for GPT-4o), while recall increases from 0.58 to 0.65 and from 0.54 to 0.60, respectively. These results indicate that the increase in AF outcomes translates into additional fault-revealing evidence without a substantial increase in false positives.

This shift is also evident at the test level. Unlike the example in Figure 1, where the generated test preserves the same invariant by modifying input values, H4 introduces an explicit constraint that is aligned with the patched semantics (Figure 11). As a result, the generated test becomes less dependent on specific input configurations and more sensitive to semantic deviations introduced by incorrect patches.

Overall, these results indicate that role alignment (H4) deepens verification for GPT-based models by prioritizing defect-targeting assertions over input-sensitive invariants, albeit at the cost of a moderate increase in non-compiling tests.

E. RQ3: Effects on Code Validity

This research question examines how guidance strategies influence *code validity*, measured by the proportion of non-compiling (NC) tests. For **GPT-based models**, explicit structural constraints consistently improve code validity. As shown in Table III, output format enforcement (H5) reduces the NC rate from 6.96% to 4.30% for GPT-4o-mini and from 4.53% to 3.30% for GPT-4o, while scope-aware constraints (H6) further stabilize compilation behavior. In contrast, the effect of structural constraints on **Llama 3.2 3B** differs markedly. Although output format enforcement (H5) yields the largest gains in APCA complementation performance (Table II), it also increases the NC rate from 7.52% to 21.05%. This increase should not be interpreted solely as degraded code validity. Rather, it reflects a shift from conservative, uninformative outputs toward more aggressive assertion generation, some of which fail to compile due to syntactic or scoping errors. Overall, these results highlight a fundamental trade-off between assertion strength and compilability. High-capacity models leverage structural guidance to suppress malformed outputs, whereas for lower-capacity models the same guidance expands the generation frontier at the cost of increased compilation failures.

```
public void testWith3() {
  Partial test = createHourMinPartial();
  ...
  assertTrue(true);
  assertEquals(0, 6);
}
```

Fig. 12: Example of assertion generation under output format enforcement (H5) for Llama 3.2 3B.

Figure 12 illustrates a representative example. Compared to the baseline behavior, in which meaningful assertions are often

absent, H5 induces the model to generate semantically relevant checks that are aligned with the APCA objective. While the resulting expansion of the generation space increases the likelihood of non-compiling outputs, the net effect is positive, as the newly generated assertions contribute to substantially improved patch-level effectiveness.

Overall, these results indicate that code validity is primarily influenced by explicit structural constraints, but that their impact is strongly model-dependent. For GPT-based models, such constraints reliably suppress malformed outputs, whereas for lower-capacity models they expand the generation frontier, albeit at the cost of increased compilation failures.

F. RQ4: Negative Effects and Failure Modes

Several guidance strategies exhibit negative or counterproductive effects, highlighting the risks of naively modifying assertion generation prompts. Removing assertion examples without compensatory guidance (H3-a) consistently degrades both verification depth and code validity across all models. This variant substantially increases non-compiling rates (by up to 15–19%) while reducing assertion-failing outcomes, indicating that assertion examples serve not only as oracle references but also as implicit structural templates. Their removal destabilizes generation rather than promoting principled novelty. Adding output format enforcement after assertion removal (H3-b) partially mitigates this degradation, particularly for lower-capacity models, suggesting that the performance loss in H3-a stems primarily from the absence of structural guidance rather than copying bias alone. In contrast, augmenting the prompt with bug-related context after assertion removal (H3-c) fails to recover performance, indicating that semantic guidance alone cannot compensate for missing syntactic exemplars. More broadly, context expansion (H1) and step-wise reasoning induction (H2) exhibit inconsistent effects across models, sometimes increasing assertion-failing rates but also introducing noise or compilation failures when not paired with explicit goal alignment or output constraints. Overall, these results reveal strong model-dependent trade-offs: high-capacity models benefit most from explicit alignment with the APCA objective (H4), whereas lower-capacity models benefit more from strict output constraints (H5). This finding underscores that effective guidance design for LLM-based APCA is not universally transferable and must be tailored to model capacity and robustness.

V. Discussion

Our results indicate that the effectiveness of guidance strategies for Automated Patch Correctness Assessment (APCA) is strongly model-dependent. High-capacity models benefit most from explicit semantic alignment, particularly role definition (H4), whereas lower-capacity models gain limited benefit from semantic cues alone and instead improve substantially under strict output format constraints (H5). These findings reveal inherent trade-offs between verification depth and robustness and suggest that effective APCA requires model-aware guidance rather than a single universal configuration. We note that LLM-based assertion generation is inherently stochastic, and that our evaluation is conducted on Java programs from Defects4J using a limited set of representative models. Moreover, assertion-failing behavior, while useful as an indicator of aggressive verification, does not always imply strong semantic alignment with the underlying defect. These factors may affect the generalizability and fine-grained interpretation of our results.

VI. Related Work

Automated Program Repair. Automated Program Repair (APR) [19]–[24] aims to automatically generate patches that fix software defects, reducing the burden of manual debugging. Representative approaches include search-based techniques, constraint-based methods [25], and statistical or learning-based systems. Most APR systems follow a generate-and-validate paradigm, where candidate patches are validated against existing test suites [26]–[29]. Despite substantial progress, APR techniques frequently produce *plausible but incorrect* patches that overfit tests without resolving the underlying defect, motivating the need for effective patch correctness assessment.

Automated Patch Correctness Assessment. Automated Patch Correctness Assessment (APCA) [30]–[45] aims to distinguish correct patches from overfitting or incorrect ones, complementing APR. Existing approaches can be broadly categorized as dynamic, static, or hybrid. Dynamic techniques, such as PATCH-SIM and DiffTGen [46], execute patched programs with generated tests to detect behavioral differences, but often incur high overhead or depend on strong test oracles. Static techniques analyze patch properties without execution, offering better scalability but limited recall. Hybrid approaches, most notably Shibboleth, combine static and dynamic signals to improve assessment accuracy.

VII. Conclusion

We presented a systematic analysis of prompt engineering strategies for LLM-based assertion generation in Automated Patch Correctness Assessment. Our study identifies two fundamental failure modes, semantic insufficiency and syntactic invalidity, and demonstrates that the effectiveness of prompt design is strongly model-dependent. We find that explicit alignment with the APCA objective significantly improves verification depth for large-capacity models, while strict output constraints are critical for enhancing robustness when using smaller models. These findings suggest that effective LLM-based APCA requires model-aware behavioral control rather than uniform prompt designs. As future work, we plan to explore adaptive prompting strategies that dynamically balance verification aggressiveness and robustness, integrate semantic relevance metrics to better assess the fault-targeting quality of generated assertions, and extend our evaluation to additional benchmarks, programming languages, and emerging LLM architectures.

References

[1] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[2] Z. Qi, F. Long, S. Achour, and M. Rinard, "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*, pp. 24–36, 2015.

[3] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "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*, pp. 532–543, 2015.

[4] H. Tian, Y. Li, W. Pian, A. K. Kabore, K. Liu, A. Habib, J. Klein, and T. F. Bissyandé, "Predicting patch correctness based on the similarity of failing test cases," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–30, 2022.

[5] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th international conference on software engineering*, pp. 789–799, 2018.

[6] A. Ghanbari and A. Marcus, "Patch correctness assessment in automated program repair based on the impact of patches on production and test code," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 654–665, 2022.

[7] F. Molina, J. M. Copia, and A. Gorla, "Improving patch correctness analysis via random testing and large language models," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 317–328, IEEE, 2024.

[8] X. Zhou, B. Xu, K. Kim, D. Han, H. H. Nguyen, T. Le-Cong, J. He, B. Le, and D. Lo, "Leveraging large language model for automatic patch correctness assessment," *IEEE Transactions on Software Engineering*, 2024.

[9] M. Fuster-Pena, D. de Fitero-Dominguez, A. Garcia-Cabot, and E. Garcia-Lopez, "Repaca: Leveraging reasoning large language models for static automated patch correctness assessment," *arXiv preprint arXiv:2507.22580*, 2025.

[10] R. Just, D. Jalali, and M. D. Ernst, "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*, pp. 437–440, 2014.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[12] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[13] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

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

[15] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1, pp. 213–224, IEEE, 2016.

[16] OpenAI, "Gpt-4o: System card." https://openai.com/research/gpt-4o, 2024. Accessed: 2025-12-17.

[17] OpenAI, "'gpt-4o mini: Efficient multimodal language model." https://openai.com/research/gpt-4o-mini, 2024. Accessed: 2025-12-17.

[18] M. AI, "'the llama 3.2 language model." https://ai.meta.com/llama/, 2024. Accessed: 2025-12-17.

[19] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1–12, IEEE, 2019.

[20] Y. Liu, P. Gao, X. Wang, J. Liu, Y. Shi, Z. Zhang, and C. Peng, "Marscode agent: Ai-native automated bug fixing," *arXiv preprint arXiv:2409.00899*, 2024.

[21] O. I. Al-Bataineh, "Automated repair of multi-fault programs: Obstacles, approaches, and prospects," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 2215–2219, IEEE, October 2024.

[22] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pp. 103–111, ACM, July 2024.

[23] Y. Peng, S. Gao, C. Gao, Y. Huo, and M. Lyu, "Domain knowledge matters: Improving prompts with fix templates for repairing python type errors," in *Proceedings of the 46th ieee/acm international conference on software engineering*, pp. 1–13, 2024.

[24] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang, and F. Liu, "Enhancing automated program repair with solution design," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1706–1718, 2024.

[25] A. Name(s), "Semantic-guided search for efficient program repair with large language models," *Journal Name*, vol. Volume Number, p. Page Range, Month Year.

[26] H. V. Pandhare, "From test case design to test data generation: How ai is redefining qa processes," *International Journal Of Engineering And Computer Science*, vol. 13, no. 12, 2024.

[27] W. Wang, C. Yang, Z. Wang, Y. Huang, Z. Chu, D. Song, L. Zhang, A. R. Chen, and L. Ma, "Testeval: Benchmarking large language models for test case generation," in *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3547–3562, 2025.

[28] K. Liu, Y. Liu, Z. Chen, J. M. Zhang, Y. Han, Y. Ma, G. Li, and G. Huang, "Llm-powered test case generation for detecting tricky bugs," *arXiv e-prints*, pp. arXiv–2404, 2024.

[29] S. Gao, C. Wang, C. Gao, X. Jiao, C. Y. Chong, S. Gao, and M. Lyu, "The prompt alchemist: Automated llm-tailored prompt optimization for test case generation," *arXiv preprint arXiv:2501.01329*, 2025.

[30] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 968–980, 2020.

[31] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 524–535, IEEE, 2019.

[32] T. Le-Cong, D.-M. Luong, X. B. D. Le, D. Lo, N.-H. Tran, B. Quang-Huy, and Q.-T. Huynh, "Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3411–3429, 2023.

[33] X. Zhou, B. Xu, K. Kim, D. Han, T. Le-Cong, J. He, B. Le, and D. Lo, "Patchzero: Zero-shot automatic patch correctness assessment," *arXiv preprint arXiv:2303.00202*, 2023.

[34] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–29, 2022.

[35] Q. Gu, "Llm-based code generation method for golang compiler testing," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 2201–2203, 2023.

[36] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[37] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "Llm-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Transactions on Software Engineering*, 2024.

[38] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pp. 284–289, IEEE, 2023.

[39] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, and G. Li, "A survey on code generation with llm-based agents," *arXiv preprint arXiv:2508.00083*, 2025.

[40] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "Llm is like a box of chocolates: the non-determinism of chatgpt in code generation," *arXiv e-prints*, pp. arXiv–2308, 2023.

[41] L. Zhong and Z. Wang, "Can llm replace stack overflow? a study on robustness and reliability of large language model code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, pp. 21841–21849, 2024.

[42] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 572–576, 2024.

[43] V. Akuthota, R. Kasula, S. T. Sumona, M. Mohiuddin, M. T. Reza, and M. M. Rahman, "Vulnerability detection and monitoring using llm," in *2023 IEEE 9th International Women in Engineering (WIE) Conference*

on Electrical and Computer Engineering (WIECON-ECE), pp. 309–314, IEEE, 2023.

[44] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "Llms in software security: A survey of vulnerability detection techniques and insights," *ACM Computing Surveys*, vol. 58, no. 5, pp. 1–35, 2025.

[45] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability detection with code language models: How far are we?," *arXiv preprint arXiv:2403.18624*, 2024.

[46] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, pp. 226–236, 2017.