STARS: Static Analysis-Guided Assertion Synthesis using Large Language Models

JIALUN CAO, Hong Kong University of Science and Technology, China and Guangzhou HKUST Fok Ying Tung Research Institute, China
HAOYU WANG*, Huazhong University of Science and Technology, China
HAORAN YAN*, Huazhong University of Science and Technology, China
MING WEN†, Huazhong University of Science and Technology, China
MICHAEL PRADEL, CISPA Helmholtz Center for Information Security, Germany

Automated unit test generation promises to reduce the cost of software quality assurance, and hence, is attracting attention from both academia and industry. Yet, generating assertions that are executable, meaningful to developers, and able to catch faults remains an unsolved challenge. Existing approaches either randomly enumerate assertions that are plausible based on static program analysis without considering whether they naturally fit the test prefix or query an LLM to generate assertions based on local context only, such as the test prefix and the focal method. However, we observe that local context alone is insufficient for LLMs to generate high-quality assertions because many desirable assertions are built from components that are almost impossible to guess for an LLM, such as sequences of multiple method calls. This paper presents STARS, a novel test assertion generation technique that combines the benefits of static program analysis and LLM-based synthesis. The key idea is to first gather a set of assertion components based on static program analysis and to then combine, concretize, prioritize, and improve them with an LLM. The resulting assertions go beyond what an LLM alone could realistically guess based on the test prefix and focal method, and they naturally fit the given test case. Empirical results show that STARS consistently outperforms the state-of-the-art baseline in five evaluation metrics. STARS achieves an exact-match rate of at most 83.4% using GPT-5.4. Compared with the baseline, STARS's mutation score nearly triples that of the baseline (10.97% vs. 3.97%), approaching that of developer-written assertions, while consuming 24.75% fewer tokens and 30.6% fewer LLM queries.

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

Additional Key Words and Phrases: Test Assertion Generation, Large Language Model

ACM Reference Format:
Jialun Cao, Haoyu Wang, Haoran Yan, Ming Wen, and Michael Pradel. 2026. STARS: Static Analysis-Guided Assertion Synthesis using Large Language Models. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA192 (October 2026), 25 pages. https://doi.org/10.1145/3832283

*Both authors contributed equally to this research.
†Ming Wen is the corresponding author.

Authors' Contact Information: Jialun Cao, Hong Kong University of Science and Technology, Hong Kong, China and Guangzhou HKUST Fok Ying Tung Research Institute, Guangzhou, China, jialuncao@ust.hk; Haoyu Wang, Haoran Yan, and Ming Wen, School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China, hy_wang@hust.edu.cn, haoran_yan@hust.edu.cn, and mwenaa@hust.edu.cn; Michael Pradel, CISPA Helmholtz Center for Information Security, Stuttgart, Germany, michael@binaervarianz.de.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA192
https://doi.org/10.1145/3832283

```
1  // Focal class
2  public class Cell {
3      protected List<TextChunk> textElements = new ArrayList<>();
4      // Focal Method
5      public List<TextChunk> getTextElements() {
6              return textElements;
7  }
8  ----------------------------------------------------------------
9  // Test Prefix
10 @Test
11 public void testGetTextElements() {
12     Cell cell = new Cell(0, 0, 0, 0);
13
14     TextElement tElement = new TextElement(0, 0, 0, 0,
15                                 new PDType1Font(Standard14Fonts.FontName.HELVETICA_BOLD), 10, "test", 5);
16     TextChunk tChunk = new TextChunk(tElement);
17     List<TextChunk> tList = new ArrayList<>();
18     tList.add(tChunk);
19     cell.setTextElements(tList);
20     // ✎ <Assertion Placeholder>
21 }
22 ----------------------------------------------------------------
23 // Assertions (Ground-truth)
24 assertEquals("test", cell.getTextElements().get(0).getText());
```

Listing 1. Motivating example with a complex assertion containing long method call sequences.

1 Introduction

Test oracles play an essential role in software testing by automatically determining whether a program behaves as expected [5, 43]. An effective test oracle can significantly reduce manual effort and improve testing accuracy [20]. However, automating the generation[1] of these oracles is challenging as it relies on a deep understanding of the software under test. As the software system grows in complexity, the problem becomes harder.

Various techniques have been proposed to generate test oracles.[2] Before the era of large language models (LLMs), static [8, 9, 20, 66] and dynamic [44, 61] approaches were extensively explored, but they usually suffer from a high false positive rate [20, 30] or produce weak true positive oracles [34, 43]. Recently, with the emergence of LLMs, more strategies have been developed. According to recent statistics [43], prompt engineering is the most common strategy among LLM-based approaches. With zero-shot [6, 16, 37, 50, 72], few-shot [45, 46], chain-of-thought [12, 30, 50, 75], or other sophisticated prompting strategies [50], more sources of information have been provided to LLMs. For test oracle generation, the sources of information for prompting [43] usually include the *focal method* (i.e., method under test) [6, 12, 16, 20, 30, 37, 50, 50, 66, 72], *test prefix* (i.e., test code before the assertion statements) [20, 30, 37, 50, 58], *focal class* (i.e., the class where the focal method belongs to) [12, 30, 46, 50, 63], and *documentation* [6, 33, 35, 37, 50].

However, although these approaches have effectively leveraged focal information (e.g., focal method/class), ***the information beyond focal code has not been fully exploited*** (e.g., code from other classes in the same repository). Take Listing 1 as an example:[3] given the focal method `getTextElements`, which returns a variable `textElements` defined in the focal class `Cell`, and the test prefix `testGetTextElements`, it is straightforward to infer that the assertion is likely to constrain over the return result of `cell.getTextElements()`. Yet, the ground truth assertion (Line 24) constrains over the method calls `cell.getTextElements().get(0).getText()`, which goes beyond the focal method/class. Especially, even though it is possible to infer `cell.getTextElements().get(0)` according to the definition of `textElements` in Line 3 (i.e., `get(int)` is a method in `ArrayList`), it is almost impossible to infer the method `getText()` without the context of this method, which is

[1]We use "generation" [30, 64] and "synthesis" [23] interchangeably.
[2]There are different kinds of test oracles: test assertions [20, 30, 33], contracts [1, 3, 7], and metamorphic relations [4, 9, 41, 68, 69]. In this paper, we focus on *test assertions*.
[3]Listing 1 and Listing 2 are adapted from https://github.com/tabulapdf/tabula-java.

```
1  // Focal class
2  public class TextElement extends Rectangle {
3      public TextElement(float y, float x,
4                         float width, float height,
5                         PDFont font, float fontSize,
6                         String c, float widthOfSpace) {
7          this(y, x, width, height, font, ...);
8      }
9      // Focal method is inherited from its Super class ↓
10 }
11 ----------------------------------------------------------------
12 // Superclass of the focal class
13 public class Rectangle {
14     public Rectangle(float y, float x,
15                      float width, float height) {
16         this(y, x, width, height);
17     }
18     // The actual implementation of the focal method
19     public float getArea() {
20         return this.width * this.height;
21     }
22 }
23 ----------------------------------------------------------------
24 // Test Prefix
25 @Test
26 public void testGetArea() {
27   Rectangle r = new Rectangle(5, 10, 50, 25);
28   TextElement textElement = new TextElement(5, 10, 50, 25, ...);
29   // ✎ <Assertion Placeholder>
30 }
31 ----------------------------------------------------------------
32 // Assertions (Ground-truth)
33 assertEquals(r.getArea(), textElement.getArea());
```

Listing 2. Motivating example with a complex assertion involving class inheritance analysis.

defined in class `TextChunk`. In other words, unless data leakage occurs, it is unlikely that LLMs can infer the correct assertion only with the given information.

Moreover, while existing approaches could effectively leverage LLMs' to understand the off-the-shelf information (i.e., inferring code intention and test oracle from identifier names), they tend to ***overlook implicit information*** (e.g., possible method invocation sequences, class inheritance hierarchies, interface implementations, and data types) ***that lies beyond what is readily available and requires program analysis to uncover***. Take Listing 2 as an example. From the given texts, one can easily identify the variable `textElement` at Line 28 and invoke the focal method via `textElement.getArea()`. However, from the textual information, one can hardly infer the relation between `textElement` (Line 28) and `r` (Line 27). Indeed, at the textual level, these two variables are unrelated in terms of their names (i.e., `textElement` and `r`) or the data types (i.e., `Rectangle` and `TextElement`) they belong to. Yet, in terms of class inheritance, `TextElement` inherits from its superclass `Rectangle`, which implies the relationship between these two variables.

To address the above two limitations (i.e., focal context could be insufficient while context beyond focal context may not be readily available), one straightforward solution is to provide LLMs with as much context as possible. However, supplying excessive input may exceed token limits. Additionally, when overwhelmed with too much information, LLMs tend to become lost in the middle [39] and may overlook critical details. An alternative approach is to retrieve similar test assertions from the same code repository [30, 71]. However, it is unlikely to retrieve all the assertions. According to our analysis (see Section 4.5), at least 33.91% of assertions cannot be directly retrieved from the source code. Moreover, if identical assertions can be retrieved from existing test cases, the generated assertions would become redundant.

To address the above challenges, we present STARS, which takes advantage of static analysis (e.g., class hierarchy analysis [17] and data type analysis) to guide LLM-based assertion generation. Our key insight is that ***the more precise assertion components*** (i.e., the variables or expressions that should be used in the assertion) we provide, ***the more likely LLMs are able to***

generate valid and desired test assertions. For example, in Listing 1, if we provide the expression `cell.getTextElements(). get(0).getText()` to an LLM, it is highly likely to produce the correct test assertion. Based on this insight, starting from a given test prefix, STARS first analyzes both the prefix and the entire project under test to synthesize relevant assertion components (e.g., method call sequences, pairs of variables). These components are then supplied to an LLM to guide the assertion generation process.

While this approach is promising, realizing it requires us to handle the huge search space of possible assertion components. Even if we manage to filter assertion components relevant to the test prefix, various implementation complexities (e.g., inheritance hierarchies, interface dependencies, cross-package invocations, Java standard library usage, and third-party libraries) further exacerbate the explosion of the search space. Furthermore, even after identifying related assertion components, filling in appropriate arguments for the methods invoked in the assertions (e.g., the argument "`0`" in `cell.getTextElements().get(0).getText()`) adds another layer of complexity. To address this challenge, we selectively explore and retain only those assertion components that are most relevant to focal code, then complete the assertions via a combination of static analysis and LLMs.

The effectiveness of STARS is evaluated on 500 test prefixes from 38 real-world Java projects used in prior work [30]. The experimental results demonstrate the effectiveness of our approach. It consistently outperforms the baseline [30] in terms of five evaluation metrics. In particular, STARS achieves the best exact-match rate of 66.2% (331/500), substantially outperforming the baseline, which reaches at best 44.2% (221/500). Its mutation score nearly triples that of the baseline (10.97 vs. 3.97), approaching the effectiveness of developer-written assertions. STARS also yields higher line and branch coverage, i.e., 21.86% (vs. 14.66%) and 15.74% (vs. 9.78%), respectively, further narrowing the gap with developer-written assertions. When a more powerful model, GPT-5.4, is adopted, STARS achieves the best exact match rate, 83.4% (= 417 / 500), with an average of eight assertions per test, while the direct prompting only hit 43.2% (= 216/500). Moreover, STARS is significantly more cost-efficient, reducing total token consumption by 24.75% (from 13.72M to 10.32M tokens) and the number of model queries by 30.6% (from 6,771 to 4,700). In summary, this paper makes the following contributions:

- ***Observation***. We observe in a preliminary study that real-world assertions predominantly involve method call sequences and variable references. This observation reveals a critical gap in existing assertion generation approaches, i.e., the limited incorporation of static analysis.
- ***Idea***. We present the first assertion generation technique that effectively combines static analysis-guided component synthesis with LLM-based assertion generation. Rather than relying solely on readily available information from the focal code, STARS empowers LLMs with static program analysis to generate more accurate and meaningful assertions by grounding them in semantically relevant program elements.
- ***Empirical evidence***. Our empirical results show that STARS consistently outperforms the baseline in all evaluation metrics, with 24.75% fewer tokens and 30.6% fewer LLM queries. STARS achieves an exact-match rate of at most 83.4% using GPT-5.4. Compared with baselines, STARS's mutation score nearly triples that of the baseline (10.97% *vs.* 3.97%), approaching that of developer-written assertions. The artifact and dataset are available at [11] to support reproduction.

2 Problem Statement and Motivation

2.1 Terminology and Problem Formulation

A unit test verifies individual pieces of code (e.g., methods or classes) using testing frameworks like JUnit. A *test prefix* refers to the portion of the test where objects are initialized and dependencies are set up before invoking the ***focal code*** (i.e., the method/class under test). In addition to the test

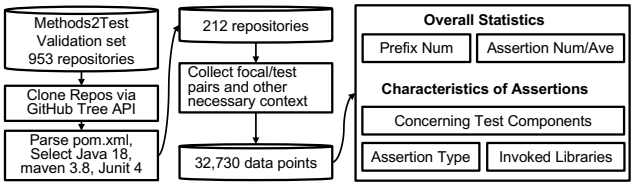

Fig. 1. Preparation for assertion characteristics analysis.

Table 1. Statistics of test assertions.

Dataset Statistics			
Total Projects	212	Total Prefix	10,357
Average Assertions per Test Prefix	3.16	Total Asserts	32,730
Characteristics of Assertions			
		Num	Ratio
Assertions with **Method Invocations**		25,325	77.40%
➥ Assertions with Method Invocations (w/ Arguments)		13,590	41.50%
➥ Assertions Invoking Standard Library Methods		14,555	44.47%
➥ Assertions invoking Static Method Calls		6,625	20.24%
Assertions with **Variables** only		3,701	11.31%

prefix, there is often a separate *test setup* phase that initializes shared objects or resources accessed by the test. For simplicity in presentation, we refer to the combination of the test prefix and test setup collectively as the ***test code***.

The *test oracle* defines the expected outcomes against which actual results are validated, and it is implemented through assertions (e.g., `assertEquals`, `assertTrue`). The ***assertion components*** (i.e., the object, method, or output value being verified by the assertion) is determined by the test oracle and represents the core behavior under test [24, 25, 42]. For example, the assertion component in Listing 2 line 35 is `textElement.getArea()`.

Problem Formulation – The problem of *test assertion generation* addressed in this paper follows the formulation established by prior work [20, 30, 33, 47, 64, 71]: *Given a test prefix that lacks assertion statements, the goal is to generate assertions that are as valid and accurate as possible.* In addition to the test prefix, supplementary context from the project, such as the *test setup* (responsible for initializing objects used in the test) [30], or *example tests* similar to the given test prefix [30, 71], may also be incorporated as input.

2.2 Study on Characteristics of Test Assertions

To better understand the problem of assertion generation, we begin by investigating the characteristics of real-world assertions. Specifically, we analyze the Methods2Test dataset [62], which contains 780,944 pairs of test cases and their corresponding focal methods, collected from 9,410 GitHub repositories and partitioned into training, validation, and testing subsets. We intentionally choose Methods2Test rather than the datasets used in prior assertion generation studies [30, 47] to separate the dataset used for characteristics analysis from that used for methodology evaluation, thereby reducing the risk of method-specific overfitting to the evaluation dataset.

For this study, the overall data preparation pipeline is illustrated in Figure 1. We began with the validation set, which contains 78,534 test–method pairs from 953 repositories, to analyze the characteristics of assertions. For each repository, we cloned the latest branch using the GitHub REST API to retrieve the corresponding Git trees. We then parsed the pom.xml files to retain only those projects that depend on Java 18, Maven 3.8, and JUnit 4. This collection procedure follows prior work [66], and the version selection is consistent with recent assertion-generation works [30, 47].

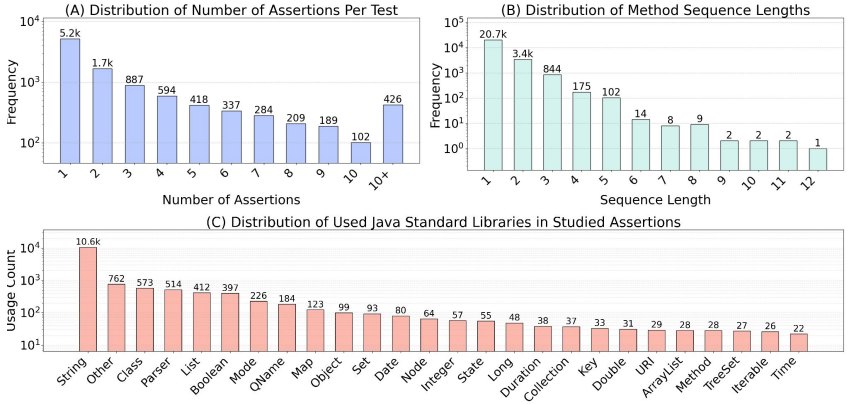

Fig. 2. Characteristics of assertions.

Table 1 shows that among 32,730 assertions analyzed, each test contains an average of 3.16 assertions. This observation echoes previous work [64]. Among these assertions, 77.4% of studied assertions involve at least one method invocation (e.g., `assertTrue(x.getText())`. Notably, 41.5% of invoked methods take arguments (e.g., `assertNotNull(x.getIndex(0))`), 44.47% of assertions invoke methods from Java standard library such as `String` and `Array`, and 20.24% invoke static methods. In contrast, 11.31% of assertions involve only variables without any method calls (e.g., `assertNotNull(result)`).

To enable deeper analysis, we present the distribution of (A) the number of assertions per test prefix, (B) the length of method call sequences within assertions, and (C) the standard library classes invoked in assertions, as shown in Figure 2. Notably, Figure 2(A) shows that 75.27% of test prefixes contain between one and three assertions, while around 4.12% contain more than 10. In Figure 2(B), more than 98.75% of assertions involve method call sequences ranging in length from 1 to 3. This finding echoes a recent study [64] that points out there are usually multiple assertions in one test prefix. Lastly, Figure 2(C) indicates that standard library classes such as `String` are among the most commonly used in assertions. These observations deepen our understanding of the assertion generation task by highlighting the complexity and diversity of real-world assertions, thereby motivating the design of our methodology.

3 Approach

The workflow of STARS is illustrated in Figure 3 and consists of four main steps: ❶ STARS begins by analyzing the input test prefix to extract relevant project context, which is then used to synthesize candidate assertion components, e.g., method call sequences, individual variables, and variable pairs (Section 3.1). ❷ Once the assertion components are identified, STARS prompts a large language model (LLM) to generate assertions that reflect the expected behavior of these assertion components (Section 3.2). ❸ Since this stage may produce a large number of candidate assertions, the next step (Section 3.3) filters and retains those most likely to validate the focal method. Assertions that fail to compile or cause runtime errors are automatically repaired. ❹ Finally, the LLM ranks the validated assertions based on their relevance and informativeness, resulting in a prioritized assertion set suitable for integration into the test prefix. In the following, we use Figure 3 as a running example to illustrate the input and output of each step.

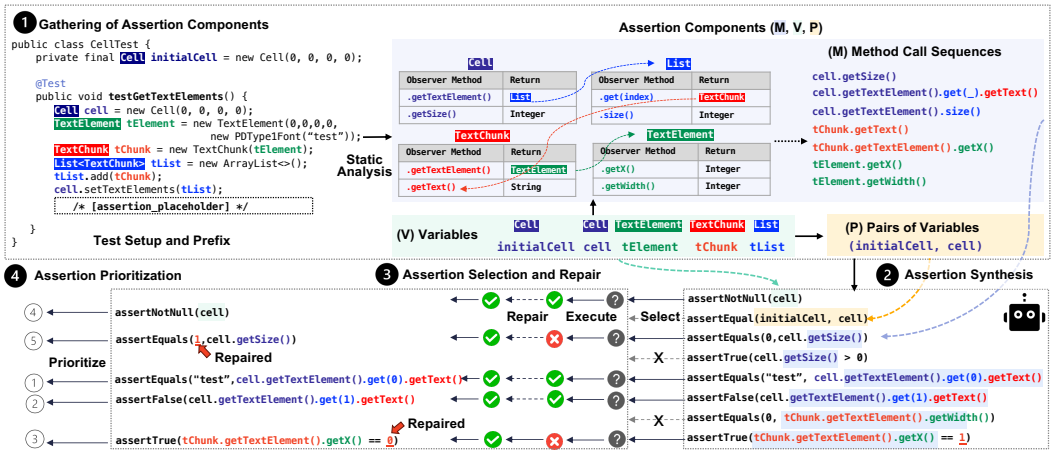

Fig. 3. Workflow of STARS.

3.1 Step 1. Gathering of Assertion Components

Given a test prefix, STARS begins by synthesizing possible assertion components to guide the following assertion generation. The entire algorithm is shown in Algorithm 1.

Step 1.1 - Gather Variables and Pairs of Variables (Alg 1, line 1) According to the observation made in Section 2.2, assertions mostly assert over method calls or variables. Thus, STARS first *extracts the variables* together with their data types declared in the test code. For example, in Figure 3 ❶, the test code includes a private member variable `initialCell` and a prefix of the test method `testGetTextElements`, STARS first extracts variables (e.g., `cell`) together with their types (e.g., `Cell`) from the given test code, as shown in green. To implement this step, STARS first parses the test prefix into an abstract syntax tree (AST) [10] and extracts variable names along with their types from assignment and expression statements, forming the set V. Note that since a test prefix could contain a high number of variables, we further query LLMs to select the variables related to the given test method. The prompt template can be found at [11].

Then, STARS *extracts the pairs of variables*, i.e., variables in V of the same type that appear more than once are grouped, and all unique pairs of such variables are gathered to form the set of variable pairs P. For example, the pair of variables collected in Figure 3 ❶ is `initialCell` and `cell` because they are both `Cell` type.

Step 1.2 Gather Method Sequences (Alg 1, lines 2-26) After extracting variables, STARS further synthesizes method call sequences by exploring the entire project using the variable information obtained in the previous step. As shown in Algorithm 1, given the variables along with their declared types extracted from Step 1.1, STARS generates a set of method call sequences M, where each sequence is composed in the form `m1.m2.`$\cdots$`.mk`, with a maximum length of k_m, a configurable parameter whose setup can be found in Section 4. To construct these call chains, STARS applies a Depth-First Search (Algorithm 2), initiating the search from all public methods of the classes corresponding to types in V, to find all valid method sequences up to length K, then add these sequences to M. During this process, any method that is not immutable (see below for details) is skipped. Additionally, the search path is terminated if the return type of a method does not belong to the focal classes or if it is the same as the method's declaring class, to avoid redundant exploration. For example, as illustrated in Figure 3, starting from the type `Cell`, which has two observer methods `getTextElement` and `getSize` (with return types `List` and `int`, respectively), the

search continues from `List`, which has an observer method `get` that takes an integer argument and returns a `TextChunk` object. Following this chain, STARS ultimately collects the method call sequences shown in the top-right corner of Figure 3 ❶ (in blue).

Observer Method Identification (Algorithm 2, lines 9–10). Rather than treating all methods equally, STARS focuses on identifying ***observer methods*** (also known as immutable methods) [2, 61] which are more likely to appear in assertions. In contrast, mutable methods that modify object state are typically not used in assertions and are therefore excluded. To determine whether a method is immutable, we refer to prior work [61], which defines a method f as an observer method if it satisfies the following conditions: (i) f returns a number or a Boolean, (ii) f does not directly or indirectly execute the `putfield` or `putstatic` bytecode instructions, and (iii) f has no parameters other than the object receiver. However, this definition is too restrictive for identifying method chains commonly used in test assertions. Specifically, for condition (i), methods that return types other than numbers or Booleans can still be part of meaningful chains; for condition (iii), as observed in Section 2.2, 41.5% of assertions invoke methods that take arguments. Therefore, we preserve condition (ii) only as the sole criterion for identifying immutable methods.

Method from Java Standard Libraries Identification. In addition to project-specific classes, methods from Java standard libraries such as `size()` in `ArrayList` are frequently used in test assertions, as shown by the statistics in Section 2.2. Therefore, we also incorporate standard library methods in our approach (lines 6–7 in Algorithm 2). From a practical view, it is infeasible to include all observer methods across all Java standard libraries. Consequently, we focus on the libraries and methods that are ranked top in our empirical study (Section 2.2). A complete list of the selected libraries and methods is available at [11]. Note that the datasets used for the study and for evaluation are different, thereby minimizing the risk of method-specific overfitting to the evaluation dataset.

Step 1.3 – Method Sequence Filtering (Alg 1, lines 9–25) After collecting the initial set of method call sequences M, STARS applies a two-stage filtering process to refine the sequences and retain non-redundant ones. First, STARS filters out any method call sequences that contain a method whose parameter types are neither among the types of V nor primitive types (lines 12–15 in Algorithm 1). This is because such methods cannot be meaningfully invoked within the current variable context, making the sequence inapplicable to the test code. Next, STARS compares method call sequences pairwise to remove redundancy (lines 16–23, Algorithm 1). Shorter subsequences like `m1.m2` are discarded in favor of longer, more complete ones like `m1.m2.m3.m4`. This filtering reduces overlap and focuses on richer, more informative sequences. Overall, the two-stage process shrinks M while keeping its expressiveness and relevance to the test context.

3.2 Step 2. Assertion Synthesis

With the assertion components collected in the previous step, including variable pairs and method sequences, STARS proceeds to generate actual assertion statements in source code using LLMs (as illustrated in Figure 3 ❷). To guide the LLM effectively, STARS constructs structured prompts that provide the necessary context for each assertion. Take the method sequence shown in Figure 3 ❷ as an example: for the component `cell.getTextElement().get(_).getText()`, the LLM is prompted to generate an appropriate assertion for it, e.g., `assertEquals("test",cell.getTextElement().get(0).getText())`. With these concrete components provided, the LLM no longer needs to infer or guess valid method call sequences. **Instead, it can focus on generating correct oracles or selecting more suitable assertion types for the given components.** Similarly, for the variable `cell`, the LLM may generate an assertion like `assertNotNull(cell)`, also shown in Figure 3 ❷. Note that for each assertion component, the LLM may generate one or more corresponding assertions. At the same time, some components may be skipped if the LLM deems them less relevant to the focal method, performing an implicit selection.

Algorithm 1: Gathering of assertion components.

Input: Focal Context *FC*, Test Prefix *Prefix*, Entire Project *P* a parameter *K* determining the maximum lengths of method call sequences
Output: A set of test components *TCompts*

```
V, P ← Extract_Variables_and_Pairs(FC) ;                    // See Section 3.1, Step 1.1
M ← ∅ ;                                                     // initialize the set storing method calls
type_groups ← GroupByClassType(V);
for type in type_groups do
    vars ← type_groups[type];
    method_seq, current_depth, min_seq ← ∅, ∅, ∅;
    initial_seq ← [ ];
    method_seq ← SynM_DFS(type, current_depth, initial_seq, K) ;          // See Algorithm 2
    for seq in method_seq do
        param_types ← ExtractParamTypes(seq) ;
        valid ← True;
        for t in param_types do
            if t ∉ type_groups and IsNotPrimitiveType(t) then
                valid ← False;
                break;
        for saved_seq in min_seq do
            if IsSubSequence(seq, saved_seq) then
                valid ← False;
                break
            if IsSubSequence(saved_seq, seq) then
                valid ← False;
                min_seq.replace(saved_seq, seq);
                break
        if valid then
            min_seq ← min_seq ∪ {seq};
    M ← M ∪ {(vars, min_seq)} ;
    TCompts ← M ∪ V ∪ P
return TCompts
```

Batch Processing – Given that some test methods can contain a large number of assertion components, STARS employs a data split strategy to manage the input size effectively. When the number of components exceeds a manageable limit for the LLM input, STARS divides them into smaller subsets. Assertions are then generated for each subset separately, after which all generated assertions are merged to form the complete assertion set for the following steps.

Prompt Design — The prompt used for assertion generation includes a problem description, detailed instructions, the test prefix, the focal method, and a set of assertion components. To manage token limits and avoid overwhelming the LLM with excessively long input, which can lead to performance degradation as noted in prior work [39], we then restrict each generation pass to take no more than k_{ai} assertion components as input components, and generate k_{ao} as output assertions. Note that k_{ai} and k_{ao} are configurable, and we show the token consumption in Section 4.6. Moreover, since the assertion components are of different types (i.e., method sequences, variables, and pairs of variables), STARS generates assertions independently for each type. This separation enables STARS to tailor the prompt format and generation strategy to the specific characteristics of each component type, thereby improving both the quality and the relevance of the generated assertions. For assertion components like method call sequences, prompts emphasize valid arguments; for variables, they should focus on value validity.

Algorithm 2: Gathering of method call sequences (line 8 in Alg. 1).

Input: The current type to search *type*, the current depth of method calls *cur_depth*, the current method sequence, *cur_seq*, a parameter K determining the maximum lengths of method calls
Output: A set of the recorded method sequences *method_sequences*

```
1  method_sequences ← ∅ ;                                  // initialize a set storing method sequences
2  if cur_depth ≥ K then
3      return { cur_seq };
4  class_code ← GetClassCode(type);
5  methods ← ParseMethods(class_code);
6  if methods is None then
7      methods ← ParseJavaStandardLibs(type);
8  for method in methods do
9      if not IsObserverMethod(method) ;                   // See Section 3.1, Step 1.2
10     then
11         continue
12     method_info ← ExtractMethodInfo(method);
13     return_type ← method_info.return_type;
14     if not IsSearchableType(return_type) then
15         new_seq ← current_sequence + [method_info];
16         method_sequences ← method_sequences ∪ {new_seq};
17         continue
18     new_seq ← current_sequence + [method_info];
19     child_seqs ←SynM_DFS(return_type, cur_depth + 1, cur_seq, K);
20     method_sequences ← method_sequences ∪ {child_seqs};
21 return method_sequences
```

3.3 Step 3. Assertion Selection and Repair

After generating a comprehensive set of assertions using LLMs in Step 2, STARS performs a critical refinement phase to ensure the quality, relevance, and correctness of these assertions. This phase consists of two main substeps: selection and repair.

Step 3.1 Assertion Selection. During the selection process, STARS filters the generated assertions to identify those most likely to test the focal methods identified in Step 2. Using the method call information and variable context, STARS evaluates each assertion's relevance using LLMs by matching it to the focal method or method sequences that it targets, then selecting assertions based on their relevance to the focal method or method sequences it targets. This ensures that only assertions directly related to the intended behaviors are retained. This step is performed using the LLM. The prompt includes the set of candidate assertions, the focal method, and a configurable maximum number of assertions to select (k_s). The complete prompt template is available at [11].

Step 3.2 Assertion Repair. Following selection, STARS applies an iterative repair process to enhance the syntactic correctness and runtime validity of the assertions. First, the selected assertions are compiled together with the test code. If compilation errors occur, STARS feeds the error messages back to the LLM, which generates repaired assertions addressing the reported issues. This cycle repeats until the assertions compile successfully or a predefined iteration limit (k_{r1}) is reached. Subsequently, STARS executes the test suite, including the generated assertions. If runtime errors, primarily assertion failures, are detected, the system again invokes iterative feedback and repair, leveraging the LLM to fix faulty assertions. This process continues until all assertions pass or the maximum number of iterations (k_{r2}) is exhausted. Note that such a repair is under a regression testing setting, which, regarding the focal code is correct, following prior work [30, 47, 53, 56].

3.4 Step 4. Assertion Prioritization

After obtaining a refined set of compiled and passing assertions from the previous step, STARS performs a final prioritization step using LLMs. The goal of this step is to rank the assertions so that the most relevant ones are surfaced first. This helps developers focus on high-value assertions, especially in cases where only a limited number of assertions can be retained or reviewed.

This step is straightforward: STARS feeds the LLM with all surviving assertions from the previous step, and prompts it to select and prioritize a certain number (k_p) of assertions. The prompt includes a set of assertions, test code, and focal code. The rationale behind the prioritization is mainly based on the LLMs' understanding of the assertions' relevance to the test prefix (e.g., the test method name) and the common order in tests (e.g., checking the validity of an object first, then invoking its methods). The prompt template can be found at [11].

Notably, in cases where the number of surviving assertions is fewer than k_p, we observe that the LLM may proactively generate several assertions to meet the quota. These supplementary assertions, while not part of the original set, are often relevant and can enrich the test suite further. Section 4.5 presents such an observation.

4 Evaluation

Our evaluation addresses four research questions (RQs):

- **RQ1. Effectiveness.** We evaluate the effectiveness of STARS under various setups (e.g., LLM choice, in-context learning strategy, the number of assertion candidates, and repair iterations).
- **RQ2. Comparison with Baseline.** We compare our STARS with ChatAssert [30], the SOTA baseline, under one of their original setups using the same LLM (i.e., gpt-3.5-turbo).
- **RQ3. Ablation Study.** We analyze the contribution of each component in STARS's pipeline through a step-by-step ablation study. This helps isolate the role of individual components and assess their importance in the overall system performance.
- **RQ4. Cost Analysis.** Since the pricing of LLM APIs varies, we measure the total number of tokens and queries to provide a model-agnostic estimate of resource consumption. This analysis helps evaluate the feasibility of using STARS at scale or in time-sensitive development workflows.

4.1 Dataset Preparation

4.1.1 Evaluation on Existing Benchmark. We begin by analyzing the limitations of recent datasets [30, 47], followed by a detailed introduction of how we augment the existing data.

***Limitation 1**: Only the first assertion is considered as ground truth.* To simplify dataset construction and evaluation, prior work [20, 30] considers the first assertion as the ground truth, ignoring subsequent ones. However, as shown in Table 1, a unit test contains an average of three assertions, making this assumption potentially inaccurate. In tests with multiple assertions, the first often checks the validity of objects, while the remaining assertions verify the actual functionality. Listing 3 presents an example unit test containing six assertions (in green). Based on the method name `testsetcustom`, the focal method under test is `setCustom` (in yellow). The first assertion verifies the validity of `result`, which is unrelated to the focal method. In contrast, the assertion in line 14 tests the expected behavior of `setCustom`. However, in the existing dataset where only the first assertion is retained as ground truth, the remaining assertions, though correctly generated, would be mistakenly considered incorrect, leading to false negatives.

***Limitation 2**: Abstracted String Literals in the ground-truth assertions.* To simplify exact match evaluation (a generated assertion is compared textually against the ground truth), existing methods [30, 47] replace concrete string literals with placeholder tokens, e.g., `assertEquals("STR",`

```
@Test
public void testsetcustom(){
 String className ="testsetCustom";
 OClass clazz = db.getMetadata().getSchema().createClass(className);
 OProperty prop = clazz.createProperty("name", OType.STRING);
 prop.setCustom("foo", "bar");
 OResultSet result = db.command("alter property "+ className + ".name custom foo='baz'");
 Object currentValue = prop.getCustom("foo");
 Assert.assertNotNull(result);
 Assert.assertTrue(result.hasNext());
 OResult next = result.next();
 Assert.assertNotNull(next);
 Assert.assertEquals("bar", next.getProperty("oldValue"))
 Assert.assertEquals("baz", currentValue);
 Assert.assertEquals(currentValue,next.getProperty("newValue"));
 result.close();
}
```

Listing 3. A test with multiple assertions.

`method.getName()`). While this abstraction streamlines evaluation, it removes meaningful semantic content, making the generated assertions less practical.

4.1.2 Dataset Augmentation. In view of these limitations, we augment the dataset constructed by the baseline [30] via collecting all assertions from the subject tests and replacing placeholder tokens for string literals with their original concrete values. This enhancement preserves the full semantic content of all assertions. As a result, the original dataset, which contained 500 assertions for the 500 test prefixes, is expanded to 1,683 assertions in the same 500 test prefixes. For instance, all assertions highlighted in green in Listing 3 are treated as ground truth for the corresponding test prefix, rather than only the first assertion. Note that the assertion involving `next` is unlikely to be generated, as the variable is not present in the test prefix, which is expected. Nevertheless, including such assertions as ground truth does not introduce false positives. With ground-truth augmented, the evaluation allows more flexibility, reducing the false negatives. After augmentation, each test prefix has 3.4 assertions, on average, echoing the observation in Section 2.2.

4.2 Experimental Setup

Evaluation Metrics: We evaluate the effectiveness of the generated assertions using five metrics:

- **Exact Match@K**: checks whether the generated assertion matches the ground truth exactly purely in the text level within the top K generated results, following the baseline [30]. We did not adopt the embedding-based or LLM-based approaches because they tend to be lenient, tolerating subtle differences. While other static metrics, such as edit distance, BLEU [52], CodeBLEU [55], and ROUGE [13], have been used in prior work [30, 56, 64, 71], we exclude them due to their limited semantic reliability [22]. To enable more accurate evaluations, four equivalence transformation rules were adopted:
  1 `assertTrue`: If it contains equality `==`, transform `assertTrue(a == b)` into `assertEquals(a, b)`; Otherwise, transform `assertTrue(expr)` into `assertEquals(true, expr)`.
  2 `assertFalse`: Transform `assertFalse(expr)` into `assertEquals(false, expr)`.
  3 `assertNull`: Transform `assertNull(obj)` into `assertEquals(null, obj)`.
  4 `assertSame`: Transform `assertSame(a, b)` into `assertEquals(a, b)`.
- **Mutation Score@K**: measures the proportion of injected faults (i.e., mutants) that are killed by the test suite over all the generated mutants, where each test contains Top-K generated assertions (i.e., k assertions for each test prefix):

$$Mutation_Score = (\#\ Killed\ Mutants)\ /\ (\#\ Mutants) \tag{1}$$

  This metric reflects the adequacy of the tests and their effectiveness in detecting faults [19, 29, 36]. In our implementation, the mutation score is calculated by PITest [15]. It operates directly on

Java bytecode and generates mutants by applying small, predefined transformations (mutators) to it. The default mutators include arithmetic operations (e.g., `IADD`, `ISUB`), comparisons and conditionals (e.g., `IF_ICMPGE`, `IFNULL`), switch statements, etc. We use these built-in mutators to generate mutants for the project under test. The number of mutants is not fixed; it varies from code under test. By default, PITest considers all applicable mutators when generating mutants, and we used the default setting.

- **Test Strength@K**: measures the ratio between the number of mutants killed by the test suite and the total number of mutants that are covered (executed) by the test, where each test contains the Top-K generated assertions:

$$Test_Strength = (\#Killed\ Mutants)\ /\ (\#Mutants\ Covered\ By\ Tests) \tag{2}$$

Unlike Mutation Score@K, whose denominator includes all generated mutants, Test Strength considers only those mutants exercised by the tests, thereby isolating the fault-detection capability from code coverage effects. We obtain this score using PITest [15].

- **Line/Branch Coverage@K**: We report the ***line coverage*** and ***branch coverage*** to assess overall code exploration. They measure the proportion of executable source lines/branches in the focal code that are exercised by the test suite when considering the Top-K returned assertions. We measure two coverages using JaCoCo [21].

Baseline Selection – We select ChatAssert [30] as our baseline, as it is a state-of-the-art method for test assertion generation. Earlier approaches such as Teco [47], ATLAS [66], and TOGA [20] were based on small fine-tuned models such as CodeT5, which were hardly comparable to powerful LLMs such as ChatGPT. To alignment with the baseline's evaluation, we reuse the assertion list provided in the their open-source artifact, preventing performance fluctuations introduced by LLM randomness. Then, we run our STARS under the same setup to ensure a fair comparison.

Note that we exclude DeepAssert [64], a related method that generates multiple assertions, as it relies on training on GraphCodeBERT, whose maximum sequence length is 512. In comparison, the LLMs we used allow many more tokens. Similarly, another related work, ALLMAssert [40], finetuned a 34b Code Llama model to generate multiple assertions, which may not be fair to compare with frontier models, making the comparison unfair to these baselines.

Studied LLMs – We select four LLMs as underlying models: (1) GPT-3.5-turbo [48], chosen to ensure a fair comparison with the existing baseline [30]; and (2) DeepSeek-V3-671B [18], selected for its strong performance as an open-source model across a range of code generation tasks; (3) GPT-5-mini [59].and (4) GPT-5.4 [49], selected as two of the latest frontier closed-source models. The hyperparameters for LLMs are set following prior work [30]: for LLM inference, we use a commonly used nucleus sampling [32], where 1, 5, and 10 solution samples are randomly generated with a temperature of 0.7. Moreover, for most RQs, we use zero-shot since in-context learning is not our focus, while we involve few-shot in RQ1 to show its impact.

Implementation Details (1) ***Environment***: The experiments are conducted on a Linux server equipped with two Intel(R) Xeon(R) Gold 6248R CPUs and 256GB of RAM. (2) ***LLM Usage***: LLMs, including GPT-3.5-turbo [48], GPT-5-mini and DeepSeek-V3 [18], are accessed via APIs provided by OpenAI and Siliconflow. (3) ***Implementation***: To clone and process GitHub repositories, we use the GitHub Tree API. For compiling and executing the Java projects, we employ Java 18, Maven 3.8, and JUnit 4, following the setup of the baseline method [30]. During evaluation, we observed that some projects in the dataset use JUnit 5; in those cases, we use JUnit 5 to ensure compatibility. For evaluation metrics, mutation score and test strength are calculated using PITest [15], while line and branch coverage are reported using JaCoCo [21]. (4) ***Parameters***: STARS's parameters are configured as follows: k_m=2 (the max method call depth in step 1); k_{ai} = 100, k_{ao}=20 (max number of input assertion components and output generated assertions per batch in step 2); k_s=40

Table 2. RQ1 (Effectiveness): The Effectiveness of STARS in terms of five evaluation metrics under three different LLMs, zero-/few-shot ("Shot"), iterations of repair ("Iteration"), and the number of candidates ("@K" where K=1/5/10). Higher values are better.

LLM	Shot	Iteration	Exact Match @1	Exact Match @5	Exact Match @10	Mutation Score @10 (%)	Test Strength @10 (%)	Line Coverage @10 (%)	Branch Coverage @10 (%)
gpt-3.5-turbo	Zero	1	128	223	235	11.1	61.81	21.23	15.39
		5	146	250	260	10.97	58.69	21.85	15.73
	Few	5	134	233	244	11.04	57.41	**22.08**	15.78
deepseek-v3	Zero	1	221	300	310	11.01	57.73	21.31	15.61
		5	**230**	311	329	11.97	63.08	21.53	**15.81**
	Few	5	225	**314**	**331**	11.96	63.38	20.56	15.13
gpt-5-mini	Zero	1	128	271	287	**13.87**	73.01	21.31	15.64
		5	139	278	293	**13.87**	**73.18**	21.48	15.79
	Few	5	134	279	289	11.86	69.44	19.96	14.74
Average			165	273	268	11.96	64.19	21.21	15.51

(max selected assertions in step 3); k_{r1}=k_{r2}=5 (the max repair iterations for compilation/runtime errors in step 3); and k_p=20 (max number of prioritized assertions in step 4). In addition, we set a 15-minute timeout for assertion generation per test prefix, a maximum of 3 hours for evaluation (e.g., calculating mutation scores) to balance performance with computational efficiency, ensuring scalability while limiting overhead.

4.3 RQ1: Effectiveness

Setup – Table 2 presents the effectiveness of STARS under different settings. For few-shot prompting, we follow existing work [30], using UniXCoder [28] to encode the code and select the most similar test in the same file based on cosine similarity (threshold 0.6). If no match is found, no example is used. The retrieved examples are adopted in the final step of STARS.

Analysis – Regarding **exact match**, our tool generates exact matching assertions for 165 – 268 out of 500 test prefixes on average. The best configuration achieves an exact-match rate of 331/500 (= 66.2%) (DeepSeek-V3, few-shot, five repair iterations). Even under the strict Top-1 setting, our approach reaches up to 230/500 (46%) exact matches. For **mutation score** and **test strength**, performance varies slightly across LLMs, ranging from 11.01%–13.87% and 57.41%–73.18%, respectively. STARS (GPT-5-mini) consistently attains the highest mutation score (13.87%) and test strength (73.18%). **Coverage metrics** remain stable across configurations, averaging 21.21% line coverage and 15.51% branch coverage, indicating that our STARS is generally stable across model choice or prompting setup.

We also observe an intriguing effect of few-shot learning on model performance. Specifically, STARS shows a slight drop (Exact Match@1 (gpt-3.5): 146 → 134). We analyzed the results and found that in-context examples may bias LLMs toward generating or prioritizing assertions involving variables that appear in the example assertions. Listing 4 illustrates this effect. In the zero-shot setting (i.e., without example code), STARS correctly generates the ground-truth assertion and ranks it at the top (line 7). However, when two example tests are provided, the LLM is misled into overemphasizing the variable `macAddress` in two example assertions (lines 13 and 19), causing it to prioritize the assertion involving `macAddress` (line 23). Although these assertions are semantically equivalent to the ground truth, since `macAddress.compareTo(macAddress)` evaluates to the same value as `comparison` due to the assignment in line 4, they do not exactly match the expected assertion. As a result, few-shot learning does not necessarily boost STARS's exact match rate. On the other hand, it also indicates that STARS relies less on example-based prompting, likely because assertion components's guidance lets STARS not only mimic the existing assertions.

```
@Test
public void testAddressCompareToEqual( ){
    final MacAddress macAddress = new MacAddress(MULTICAST_ADDRESS) ;
    final int comparison = macAddress.compareTo(macAddress);

    // ✔ Generated by STARS  in a zero-shot setting, the same as ground truth
    assertEquals(0,comparison);

    // Retrieved Example 1:
    @Test
    public void testAddressMulticast ( ) {
        final MacAddress macAddress = new MacAddress(MULTICAST_ADDRESS);
        assertTrue(macAddress.isMulticast());
    }
    // Retrieved Example 2:
    @Test
    public void testAddressHashCode( ){
        final MacAddress macAddress = new MacAddress(MULTICAST_ADDRESS);
        assertNotEquals(0, macAddress.hashCode());
    }

    // ✗ Generated by STARS  in a few-shot setting with the above retrieved examples
    assertEquals(0, macAddress.compareTo(macAddress));
}
```

Listing 4. Example of the impact of few-shot setting

Notably, STARS demonstrates substantial performance gains when paired with more capable LLMs. For example, the exact match@10 under zero-shot with one iteration: 235 (gpt-3.5) → 287 (gpt-5-mini) → 310 (deepseek-v3). The reason may because the advanced LLMs could better utilize the assertion components provided by STARS. As a result, more valid and meaningful assertions could be generated, as further discussed in Section 4.5.

4.4 RQ2: Comparison with Baseline

Setup – Table 3 shows the comparison with the baseline ChatAssert in terms of the five metrics over 500 test prefixes across 38 projects, including 32 single-package projects and 6 multiple-package projects. For each package, we report the number of test prefixes accordingly. The repair process is configured to allow up to five iterations for compilation errors and an additional five iterations for runtime assertion errors following the baseline setting [30]. For the baseline assertions, we reuse the assertion lists provided in their release artifact to prevent performance fluctuations introduced by LLM randomness. Accordingly, for a fair comparison, our approach uses the same setup (gpt-3.5-turbo, zero-shot and five iterations).

Analysis – Overall, according to the last row in Table 3, STARS outperforms the baseline across all evaluation metrics. For exact match 1, 5, 10, the results achieved by STARS are 146, 250, and 260, compared with 119, 194, and 209. For mutation score, STARS achieves 10.97% (Mutation Score@10), which is more than three times the baseline (3.97%) and closely approaches the ground truth (12.29%). Similarly, STARS's test strength outperform the baseline. In terms of test strength@10, STARS achieves 49.41%, representing a 39.66% improvement over the baseline (35.38%). For coverages, STARS also consistently surpass the baseline scores (line coverage @10: 21.86% *vs.* 15.15%, branch coverage@10: 15.74% *vs.* 9.78%). Additionally, it is noteworthy that the mutation score and code coverages of STARS **approach that of developer-written ground-truth assertions** (Labeled as "Gold" in Table 3. The ground-truth is low due to the limited number of tests per project (see "# of test prefix")), demonstrating the practicality of STARS. We further conclude that STARS produces higher-quality assertions than the baseline, as evidenced by its significantly higher mutation score (p-value = 0.0284 < 0.05) with a small effect size.

Listing 5 shows an example from *orientdb-server* project. STARS successfully generates the desired assertion, which contains a complex method call sequence, while the baseline generates a simple one with abstracted string literals. Essentially, with the assertion components (`secondDatabase`.

Table 3. **RQ2: Comparison with Baseline**. Larger values (darker colors) indicate better performance. We compare our approach (**Ours**) against the baseline **ChatAssert** [30] (**Base**). We also report ground-truth results (**Gold**). "/" denotes time out (15 minutes per test prefix). "-" denotes no package names. Project and package names are abbreviated.

Project	Pkg	# of Test Prefix	Exact Match						Mutation Score					Test Strength					Line Coverage					Branch Coverage				
			Baseline			Ours			Gold	Baseline		Ours		Gold	Baseline		Ours		Gold	Baseline		Ours		Gold	Baseline		Ours	
			@1	@5	@10	@1	@5	@10		@1	@10	@1	@10		@1	@10	1	@10		@1	@10	@1	@10		@1	@10	@1	@10
Single-Package Projects																												
a-found	-	18	2	4	5	9	15	15	13.39	9.07	9.60	10.60	11.10	81.34	27.78	84.60	79.90	82.28	20.12	17.26	17.97	17.05	17.13	16.19	12.47	13.56	12.76	12.81
beeju	-	1	0	0	0	0	0	0	4.38	2.50	2.50	1.88	6.88	17.50	10.26	10.26	7.69	25.58	23.45	23.20	23.20	22.94	24.23	19.23	19.23	19.23	19.23	19.23
ci	-	11	5	7	7	3	5	5	1.38	0.62	0.85	0.71	1.11	72.09	58.33	61.29	47.06	73.53	3.57	3.09	3.09	3.22	3.22	1.48	1.05	1.05	1.05	1.05
commons	-	5	0	0	0	0	1	1	6.60	0.69	0.69	1.38	2.75	96.00	50.00	50.00	52.62	86.96	3.86	1.51	1.51	1.89	2.42	4.91	1.52	1.52	2.03	2.88
coveralls	-	23	2	2	2	10	12	12	10.51	5.60	6.83	9.05	11.48	90.91	66.67	67.24	59.42	71.23	14.36	12.01	12.97	16.46	17.68	9.07	8.19	9.07	12.87	13.74
cubesen	-	1	1	1	1	1	1	1	1.85	0.00	0.93	1.84	1.85	100.00	0.00	50.00	100.00	100.00	1.08	1.08	1.08	1.08	1.08	0.00	0.00	0.00	0.00	0.00
cucumber	-	8	1	1	1	2	3	3	29.29	8.59	9.60	20.20	26.26	85.29	77.27	73.08	74.07	82.54	32.05	23.08	23.08	26.71	27.99	30.41	18.92	18.92	21.62	22.97
collect	-	1	0	0	0	0	0	0	0.80	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00	0.00	68.75	0.00	0.00	1.25	1.25	88.24	0.00	0.00	0.00	0.00
dropw	-	1	1	1	1	1	1	1	4.65	4.65	4.65	4.65	4.65	100.00	100.00	100.00	100.00	100.00	3.23	3.23	3.23	3.23	3.23	25.00	25.00	25.00	25.00	25.00
embed	-	1	0	0	0	0	0	0	2.36	1.89	1.89	1.89	1.89	16.39	13.11	13.11	13.11	13.11	18.26	18.26	18.26	18.26	18.26	7.82	7.82	7.82	7.82	7.82
fluent		1	1	1	1	0	0	0	6.96	5.69	6.96	5.69	8.86	36.67	30.00	35.48	30.00	41.18	19.51	7.32	7.32	19.51	21.95	11.70	5.32	5.32	11.70	15.96
InMem	-	1	0	0	0	0	0	1	57.14	0.00	0.00	57.14	57.14	69.57	0.00	0.00	69.57	69.57	83.72	65.12	65.12	83.72	83.72	63.64	40.91	40.91	63.64	63.64
jactive	-	9	0	0	0	0	0	0	6.50	2.35	3.59	2.48	3.60	72.31	39.53	54.17	40.00	54.17	7.25	6.05	6.13	5.65	6.29	17.78	12.00	12.44	11.56	13.78
Socket	-	16	3	4	5	1	4	4	5.04	2.78	3.94	1.97	4.69	62.59	53.33	53.12	35.79	55.10	8.86	6.87	7.86	6.80	8.29	5.10	2.87	4.31	2.87	4.24
Sockify	-	1	0	0	0	0	0	0	2.79	0.00	0.00	2.39	2.39	77.78	0.00	0.00	66.67	66.67	2.72	2.17	2.17	2.72	2.72	2.22	1.67	1.67	2.22	2.22
jcodec	-	18	4	7	7	3	7	7	0.88	0.32	0.52	0.46	0.76	57.86	39.72	45.50	40.64	47.78	2.60	1.93	2.08	1.96	2.60	1.22	0.85	1.06	0.95	1.12
jxapi	-	38	20	32	33	31	35	36	5.84	4.10	6.35	4.61	7.89	72.15	64.52	72.94	67.16	66.38	17.43	16.65	17.62	16.34	19.67	4.17	4.34	4.84	3.67	6.01
mark	-	6	0	0	0	0	0	0	23.64	9.09	17.58	16.43	19.39	50.64	20.54	39.73	47.36	43.84	46.14	43.82	44.21	23.75	44.21	35.09	28.07	32.46	15.79	31.58
money	-	7	0	0	0	0	0	0	11.13	4.00	5.22	4.87	11.48	84.21	40.35	44.12	45.90	89.19	68.6	58.91	64.34	56.20	70.54	52.99	32.84	44.78	29.85	50.75
morph	-	17	13	15	16	0	15	15	16.51	/	/	/	/	44.94	/	/	/	/	41.77	27.05	30.18	39.73	43.67	38.50	18.36	20.63	34.65	38.74
open	-	3	0	0	0	0	0	0	14.04	4.21	4.21	7.41	11.19	50.53	34.30	34.30	28.67	43.35	51.55	32.75	32.75	46.84	46.84	21.69	11.79	11.79	23.68	23.68
ParCE	-	9	5	5	5	5	5	6	48.72	11.97	14.53	31.61	45.30	72.15	60.87	62.96	55.22	64.63	73.93	23.5	24.36	64.1	72.65	53.73	12.69	13.43	41.04	50.75
sofa	-	15	1	5	7	4	7	8	25.28	20.12	21.28	21.33	23.26	51.89	49.38	51.98	44.14	47.62	45.24	40.27	40.36	43.63	44.02	33.05	26.99	27.19	31.47	32.06
boot	-	1	0	0	0	0	0	0	9.52	0.00	0.00	0.00	9.52	32.00	0.00	0.00	0.00	32.00	23.08	20.36	20.36	22.62	23.08	3.23	3.23	3.23	3.23	3.23
spring	-	1	0	0	0	0	0	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	23.08	0.00	0.00	0.00	0.00	3.23	0.00	0.00	0.00	0.00
stat	-	21	4	7	7	4	7	7	33.94	16.5	19.27	22.02	33.94	92.5	78.25	80.77	66.67	88.10	40.27	28.73	29.41	36.65	38.46	26.74	12.79	12.79	26.74	26.74
tcl	-	6	5	6	6	1	5	5	22.84	15.45	16.94	16.43	22.27	61.57	52.22	56.38	47.36	60.31	43.27	34.04	34.50	41.61	43.06	28.47	23.50	23.63	26.77	28.47
Teaching	-	16	8	10	11	10	14	14	0.88	0.52	0.97	0.80	1.25	87.22	58.54	76.27	80.43	84.06	0.61	0.58	0.87	0.63	0.97	1.36	0.99	2.04	1.11	2.35
wymi	-	1	0	0	0	1	1	1	0.83	0.25	0.95	0.83	0.83	76.47	23.52	88.24	76.47	76.47	1.38	1.34	1.34	1.38	1.38	1.37	1.19	1.19	1.37	1.37
XsdParser	-	2	0	1	1	0	0	0	40.31	16.45	16.45	41.89	43.15	90.78	66.77	66.77	96.73	98.56	54.79	36.93	36.93	54.29	54.71	36.08	20.06	20.06	35.48	35.48
raml	-	2	0	0	0	0	0	0	0.34	0.00	0.00	0.00	0.00	50	0.00	0.00	0.00	0.00	1.97	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
yubico	-	7	4	5	5	6	7	7	46.46	2.36	2.36	37.80	43.30	65.56	27.27	27.27	63.16	66.27	58.71	13.23	13.23	58.71	60.32	55.88	7.35	7.35	55.88	56.62
Ave (Single-Pkgs)		268	80	114	121	**92**	**145**	**149**	14.21	4.83	5.76	**10.59**	**13.49**	66.22	36.86	45.15	**49.54**	**59.05**	28.29	17.82	18.30	**23.09**	**25.18**	**21.86**	11.31	12.10	**16.44**	**18.57**
Multi-package Projects																												
fabric8	core	13	4	7	7	8	9	9	3.29	1.27	1.27	1.88	3.05	74.42	44.58	44.58	44.35	71.20	5.13	3.33	3.33	4.47	4.49	3.78	2.22	2.22	3.46	3.62
graphviz	enric	3	0	0	0	0	0	0	2.62	1.75	1.75	1.31	1.31	100.00	80.00	80.00	100.00	100.00	13.15	2.11	2.11	10.33	10.33	13.64	2.02	2.02	11.62	11.62
	graph	2	0	0	0	0	0	0	10.71	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00	0.00	15.13	0.00	0.00	8.40	8.40	0.00	0.00	0.00	0.00	0.00
	viz	13	0	0	0	0	0	0	6.20	0.32	0.96	1.12	2.03	82.27	25.00	48.65	45.65	71.69	15.85	3.46	3.71	13.81	13.81	6.52	1.02	1.32	5.78	5.78
hazelcast	simul	33	5	9	11	12	19	19	9.44	3.92	5.77	6.90	9.88	61.68	41.25	53.96	49.44	62.38	14.25	10.87	11.15	14.02	14.98	12.49	8.41	8.78	11.15	12.94
	hazel	1	0	0	0	0	0	0	0.08	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.00	0.00
nano	core	5	1	2	2	2	3	3	20.04	0.00	0.00	14.48	23.83	46.88	0.00	0.00	31.71	52.20	44.00	16.16	16.16	46.44	46.88	28.00	5.46	5.46	29.89	30.46
	nanol	7	0	0	0	2	2	2	17.12	/	/	/	/	38.78	/	/	/	/	36.00	38.14	38.14	42.80	42.80	23.00	24.66	23.29	27.40	28.77
	webso	1	1	1	1	1	1	1	2.42	5.15	5.15	0.00	2.42	72.73	32.69	32.69	0.00	72.73	4.00	14.38	14.38	3.70	3.70	3.00	7.04	7.04	3.29	3.29
	server	11	3	3	3	1	7	7	23.07	0.76	0.85	16.85	19.20	100.00	57.33	60.00	79.95	90.05	23.55	14.00	22.74	23.14	23.14	15.55	9.09	15.03	15.15	15.18
orientdb	core	83	18	34	37	19	40	45	/	/	/	/	/	/	/	/	/	/	23.46	16.66	16.80	22.53	22.73	16.20	10.71	10.80	15.50	15.71
	graph	17	5	13	13	0	10	11	2.15	1.05	2.04	1.05	1.35	42.64	34.17	38.83	34.17	40.77	5.27	4.60	5.17	4.69	4.84	2.41	2.13	2.75	2.13	2.34
	object	3	0	1	2	1	1	1	11.08	5.87	6.93	12.31	14.39	52.24	40.44	47.13	81.02	92.12	19.15	14.76	14.85	18.22	18.42	16.58	12.69	12.75	15.40	16.02
	lucene	8	0	4	4	0	1	1	21.12	/	/	/	/	55.4	/	/	/	/	40.65	23.43	23.43	39.93	39.93	26.02	12.65	12.65	25.20	25.20
	etl	8	0	3	5	0	2	2	18.04	/	/	/	/	52.84	/	/	/	/	33.10	7.82	7.82	32.13	32.29	19.81	0.00	0.00	18.55	18.55
	client	3	0	1	1	2	3	3	1.35	0.95	1.05	1.09	1.70	51.95	36.84	39.74	40.75	62.50	3.33	3.31	3.34	3.31	3.40	2.61	2.61	2.61	2.61	2.61
	distri	1	1	1	1	0	1	1	0.02	0.02	0.11	0.02	0.02	50.00	50.00	100.00	100.00	14.29	0.13	0.13	0.15	0.12	0.17	0.00	0.00	0.00	0.00	0.00
storm	core	18	1	1	1	6	6	6	9.64	1.53	1.53	5.20	6.97	57.22	42.67	42.67	30.11	39.57	17.03	4.07	4.07	17.48	17.61	15.70	4.60	4.60	16.15	16.29
	exter	2	0	0	0	0	0	0	1.15	0.00	0.00	0.38	0.38	100.00	0.00	0.00	100.00	100.00	1.70	0.00	0.00	1.27	1.27	0.00	0.00	0.00	0.00	0.00 v
Ave (Multi-Pkgs)		232	39	80	88	**54**	**105**	**111**	8.86	1.51	1.83	4.17	5.77	68.84	32.33	39.22	49.14	57.97	16.57	9.33	9.86	16.15	16.27	10.81	5.54	5.86	**10.70**	**10.97**
Ave (All Pkgs)		500	119	194	209	**146**	**250**	**260**	12.29	3.75	4.48	**8.50**	**10.97**	67.16	35.38	43.21	**49.41**	**58.70**	23.92	14.66	15.15	**20.50**	**21.86**	17.74	9.16	9.78	**14.30**	**15.74**

`getMetadata().getSequenceLibrary()`)) provided, LLMs can then focus on completing the assertion instead. Note that with a retrieval, we found that this method call sequence does not exist in the project, which again consolidates the necessity of synthesis rather than retrieval.

Interestingly, in some projects (e.g., projects *XsdParser*, *yubico-java*, and project *nanohttpd*), the assertions generated by STARS even outperformed the developer-written assertions in mutation

```java
@Test
public void testSequencesUpdate( ) throws InterruptedException{
    database.activateOnCurrentThread( ) ;
    ...
    database.command("test") ;
    secondDatabase.activateOnCurrentThread( ) ;
    // ✔ Generated by STARS , the same as ground truth
    assertNotNull(secondDatabase.getMetadata().getSequenceLibrary().getSequence("test"))
    // ✗ Generated by baseline
    assertEquals(1,database.countClass("STR"));
}
```

Listing 5. Example of assertions generated by STARS and ChatAssert [30]

Table 4. RQ3. (Ablation Study): Impact of each step in STARS under three LLMs. Higher values are better.

Step	Total	STARS (GPT-3.5-turbo)			STARS (Deepseek-V3)			STARS (GPT-5-mini)		
		Cand.	Exact Match		Cand.	Exact Match		Cand.	Exact Match	
			Num	Ratio		Num	Ratio		Num	Ratio
Step 1. Gathering*	500	/	383*	0.77*	/	383*	0.77*	/	383*	0.77*
Step 2. Generation	500	11,604	254	0.508	12,831	299	0.598	11,831	298	0.596
Step 3.1. Selection	500	7,453	250	0.500	8,645	298	0.596	8,555	293	0.586
Step 3.2. Repair	500	9,480	265	0.530	8,145	333	0.666	8,371	302	0.604
Step 4. Prioritization (top 1)	500	500	**146**	0.292	500	**230**	0.460	500	**139**	0.278
Step 4. Prioritization (top 10)	500	3,503	**260**	0.520	4,319	**329**	0.658	3,907	**293**	0.586

*: See Setup in Section 4.5. **Cand.**: Number of candidate test assertions/targets in the corresponding step.

score or line/branch coverages. For instance, in the *XsdParser* project, STARS achieves a mutation score@1 of 41.89%, demonstrating that even a single Top-1 STARS-generated assertion per test prefix outperforms developer-written assertions (40.31%). When allowing up to Top-10 assertions, STARS further boosts the mutation score to 43.15%. Another example in the project *nanohttpd* (package: *core*), STARS achieves 46.44% line coverage@1 and 29.89% branch coverage@1, better than the developer-written assertions (44% and 28%, respectively), and these two coverages can be further enhanced to 46.88% and 30.46% when allowing Top-10 assertions. These results suggest that STARS not only generates more effective assertions than the baseline, but can also **complement, and in some cases, enhance the developer-written assertions.**

4.5 RQ3: Ablation Study

Setup — The ablation study is in a zero-shot setting, with a maximum of 5 repair iterations. For Step 1 (Gathering), where no assertion is generated, we report whether the gathered assertion components appear in the ground-truth assertions. For Step 4 (Prioritization), results are shown with top-1 and top-10 cut-offs (i.e., one or ten assertions per test prefix). In addition, we also compare our STARS with direct use of GPT-5.4, showing the benefit our approach brings.

Analysis of improvements brought by each step in STARS — Table 4 shows STARS's ablation study. From Step 1 (Gathering) in Table 4, we observe that STARS's synthesized assertion components covers approximately 77% of developer-written assertions (this score is identical across three LLMs because Step 1 does not involve LLMs). This match rate indicates that the majority of the assertion components used by developers can effectively be captured in our synthesis step. In other words, if the LLMs generate correct assertions based on the synthesized assertion components, at least 77% of developer-written assertions could be replicated. This finding provides strong empirical evidence for the statistical usefulness of assertion components synthesis.

We further investigated all the assertions involving method call sequences and found that 33.91% (=235/693) of sequences cannot be retrieved directly from the project source code, which further consolidated the necessity of synthesis rather than retrieval in Step 1. The remaining unmatched

Table 5. RQ3 (Ablation Study) Comparison with Direct Use of GPT-5.4. Best results are highlighted in **bold**.

Assertion Per Prefix	Total	Direct Use (GPT-5.4)					STARS (GPT-5.4)				
		Cand.	W/o Repair	W/ Repair			Cand.	W/o Repair	W/ Repair		
			Match	Match	Strength	Mutation		Match	Match	Strength	Mutation
Top 1	500	500	103	103	61.21	9.35	500	**183**	**293**	**65.8**	**11.41**
Top 10	500	5,000	202	208	65.73	12.42	2,833	**285**	**415**	**69.53**	**14.64**
All	500	9,973	210	216	67.16	12.82	4,319	**306**	**417**	**71.43**	**14.87**

*__Total__: Total number of test prefixes for which assertions are to be generated. **Cand.**: Number of test assertions.
*__Match__: Exact Match. **Strength**: Test Strength. *__Mutation__: Mutation Score.

assertions are primarily due to cases such as invocations of static methods, for example, `assertTrue (ResourceUtil.jsonEquals(first, same))`, which leave no observable clues in the test prefix; calls to third-party libraries; or methods from other test methods. Also, this step is LLM-independent, since no LLMs are involved yet; thus, the results for Step 1 are identical across LLMs.

The advantage of more powerful LLMs becomes apparent from Step 2 onward. Given the same set of assertion components, Deepseek-V3 and GPT-5-mini generate higher numbers of valid assertions compared to GPT-3.5-turbo (exact match: 299 and 298 *vs.* 254). This disparity marks the point where the performance gap between the two LLMs begins to widen. In Step 3.1 (Selection), although the exact-match rate drops slightly, the number of candidate assertions is dramatically reduced, from roughly 12k to 8k, demonstrating a significant pruning benefit. Then, Step 3.2 (Repair) contributes to an overall improvement for three models. Finally, Step 4 prioritizes a fixed number of assertions for each test prefix; as more assertions are considered, the exact-match rate increases. Deepseek-V3 achieves the best performance, with 329 exact matches when the top 10 assertions are considered.

Comparison with direct use of the frontier model – Table 5 shows the comparison with the direct use of GPT-5.4. In particular, for each test prefix, we prompt GPT-5.4 using the same prompt template as our STARS (Step 2, Assertion Generation), except for the gathered components collected in our first step. In the prompt, each prefix is allowed to generate at most 20 test assertions, the same as what has been described in our prompt. Since the output assertions are required to be prioritized, we then show the metrics on the Top-1/10/all-generated assertions, and compare with our STARS. For fairness, the comparison is divided into two groups: with/without repair, and the implementation reuses our Step 3.2 (Assertion Repair in Section 3.3).

We can see from Table 5 that our STARS demonstrates the consistent superiority over the direct LLM baseline under all settings. In particular, without repair, the comparison of exact match under different settings is as follows: 103 *vs.* **183** (Top-1), 208 *vs.* **285** (Top-10), and 216 *vs.* **306** (All generated assertions). **The superiority becomes even larger after repair**. For the Exact Match, the comparison shows: 103 *vs.* **293** (Top-1), 208 *vs.* **415** (Top-10), almost double the results. The mutation score and test strength also show observable performance gains, e.g., mutation score improves from 9.35 to 11.41 (Top-1), test strength rises from 61.21 to 65.8 (Top-1). Moreover, when we take all generated assertions into account (the last row in Table 5), the direct prompting generates 9973 assertions, with an approximate average of 20 (19.95 = 9,973 / 500, while only exactly matches 43.2% (=216/500) assertions. In comparison, because the proper assertion components are provided, **the number of candidates halves** ($9,973 \rightarrow 4,319$, with an average of 9 (8.64 = 4,319 / 500) assertions per test), **while the number of exact matches doubles (**$216 \rightarrow$ **417 after repair)**.

Additionally, compare with the best results in Table 4 achieved by STARS (Deepseek-V3), whose Exact Match with Top-10 assertions per prefix is 329 (65.8% = 329/500), STARS (GPT-5.4) further pushes the limits dramatically to 417 (83.4% = 417 / 500). This observation further echoes the observation in Table 4 that the more powerful model gains more performance improvements from our framework, further consolidating the effectiveness of our design.

Table 6. RQ4 (Cost analysis): Number of tokens and queries to generate each executable assertion.

	Baseline - ChatAssert						**Ours** - STARS					
	Overall		**Per Assertion**		**Per Test Prefix**		**Overall**		**Per Assertion**		**Per Test Prefix**	
	Tokens	Queries	Tokens	Queries	Tokens	Queries	Tokens	Queries	Tokens	Queries	Tokens	Queries
Step 1. Gathering	\	\	\	\	\	\	494,993	458	120.97	0.11	989.99	0.92
Step 2. Synthesis	12,724,205	5,632	2,924.43	1.29	25,550.61	11.31	3,280,682	1,361	801.73	0.33	6,561.36	2.72
Step 3.1 Selection	\	\	\	\	\	\	181,280	51	44.30	0.01	362.56	0.10
Step 3.2 Repair	996,226	1,139	228.96	0.26	2,000.45	2.28	5,929,876	2,330	1,449.14	0.57	11,859.75	4.66
Step 4 Prioritize	\	\	\	\	\	\	437,169	500	106.84	0.12	874.34	1.00
Overall	13,720,431	6,771	3,153.39	1.55	27,551.06	13.59	**10,324,000**	**4,700**	**2,522.97**	**1.15**	**20,648.00**	**9.40**

Table 7. RQ4 - **Impact of Parameters on Cost Efficiency**: Cost and Exact Match Affected by the Depth of Method Calls (K_m) and the Number of Repair Iterations.

Repair Iteration	Depth of Method Calls (K_m)	Exact Match	Tokens	Queries
1	2	128	5,205,298	2,824
	3	165	10,978,893	4,222
5	2	146	10,324,000	4,700

4.6 RQ4: Cost Analysis

Setup – Table 6 shows the total number of tokens and queries; the average number of tokens and queries to generate each assertion. The experiment used GPT-3.5-turbo in zero-shot mode with temperature 0.7 and five repair iterations. Reported statistics include input and output tokens.

Analysis – Overall, STARS reduces total token consumption by 24.75% (from 13.72M to 10.32M tokens) and query count by 30.6% (from 6,771 to 4,700 queries). On a per-assertion basis, STARS achieves a 20.0% reduction in tokens (3,153.39 vs. 2,522.97). The efficiency gains are even more pronounced when normalized per test prefix: STARS reduces token usage by 25.1% (27,551.06 vs. 20,648.00) and query count by 30.8% (13.59 vs. 9.40). The most significant savings occur during generation, where STARS consumes around a quarter of tokens (12.72M vs. 3.28M) and one-fifth of the queries (5,632 vs. 1,361). Despite incorporating additional stages (gathering, selection, and prioritization), STARS still achieves a lower overall cost. The baseline's high generation cost stems from its reliance on multi-round prompting for code summarization, where outputs are recursively fed back into subsequent inputs, leading to heavy token accumulation. In contrast, the relatively higher cost for our STARS happens in Step 2 (i.e., Synthesis) and Step 3.2 (i.e., Repair). Their costs are justified by the significant quality improvements they bring, as demonstrated in previous RQs.

Cost Efficiency Under Different Parameters – Beyond Table 6, which calculates the costs and queries consumption under the setting of $K_m = 2$ (the depth of method calls) with 5 repair iterations, we further show how these two hyperparameters affect the cost-efficiency in Table 7. Other setups remain the same as in Table 6. We can see from Table 7 that, when increasing K_m from 2 to 3, the Exact Match@1 increases 37 instances, with the tokens doubles ($5,205,298 \rightarrow 10,978,893$), queries doubles ($2,824 \rightarrow 4,222$) under 1 repair iteration. Similarly, when increasing repair iteration from 1 to 5, the tokens and queries also about doubles ($5,205,298 \rightarrow 10,324,000$ and $2,824 \rightarrow 4,700$, respectively) when $K_m = 2$.

5 Threats to Validity

There are four main threats to the validity of this paper. First, *LLM selection bias*. We use four models (GPT-3.5-turbo, GPT-5-mini, DeepSeek-V3, and GPT-5.4). However, these are widely used and serve as representative LLMs, and our goal is not to benchmark models but to demonstrate the general applicability of our method. Second, *dataset selection bias* since we build on the existing benchmark whose real-world representativeness may be limited. Still, the benchmark includes 500

randomly sampled tests, offering reasonable coverage. Third, our method involves several tunable parameters (e.g., number of generated assertions) which may introduce *parameter selection bias*. To address this, we make all parameters configurable and report the number of candidates retained at each stage to promote transparency and reproducibility. Last, the possible *data contamination* might influence the effectiveness of the LLMs used in this evaluation. Yet, since the data leakage would affect both baseline and our approach equally when using the same LLMs, the relative improvements still show the gains from our method.

6 Related Work

Assertion Generation. Assertion generation has received considerable attention in the software testing community [43]. Learning-based methods[20, 47, 62, 64, 66] often struggle with limited accuracy and poor generalizability across diverse codebases. Retrieval-augmented approaches[30, 40, 60, 71] aim to improve assertion relevance by retrieving and adapting similar examples from existing test suites. However, their effectiveness heavily depends on the availability of structurally and semantically similar test cases, and they often suffer from syntactic incoherence or context mismatch. In contrast, our approach, STARS, does not rely on mimicking existing tests within the same test file. Instead, it is guided to synthesize more complex and semantically meaningful assertions, often involving concrete method call sequences or interrelated variable pairs, thereby enhancing both expressiveness and correctness. Moreover, the assertions generated by previous works often struggle with compilation errors [20, 47, 74]. While iterative repair [47] alleviates some of these, performance remains constrained by model capacity. Regarding the number of generated assertions, ALLMAssert [40] points out the limitation of generating a single assertion, and thus proposes to generate multiple assertions for each unit test. Recent LLM-based methods [30, 73] leverage richer context and achieve promising results. Unlike prior work that prompts LLMs to generate assertions end-to-end, our approach introduces an intermediate representation, assertion components, to better guide the generation.

Unit Test Generation. Recent learning-based test generation has evolved along two phases. Early approaches [62] treated test generation as a neural machine translation task, fine-tuning models like BART [14] and CodeT5 [65] on paired method-test datasets [62, 66], though such methods are limited by the scope of training data and generalizability. Recently, with the advent of LLMs, recent techniques increasingly adopt prompting-based strategies without fine-tuning, enabling more flexible [26, 27] and scalable [51, 54] test generation. For example, retrieval-augmented few-shot prompting [45, 57, 71], chain-of-thought [67], tree-of-thought [70] or other domain knowledge [38] are adopted to activate and facilitate LLMs' simulation and reasoning capability. Apart from adopting advanced prompting techniques, diverse sources of information, including focal code [6, 12, 16, 20, 30, 31, 37, 46, 50, 50, 63, 66, 72], text context [20, 30, 37, 50, 58], and *documentations* [6, 33, 35, 37, 50, 53], were considered. Unlike these works, the information we provide goes beyond what is readily available and requires program analysis to uncover. Although recent approaches also combine program analysis with LLMs [27, 51], they differ from ours in their objectives, i.e., focusing on generating unit tests rather than generating assertions for given prefixes.

7 Conclusion

In this paper, we present STARS, a static analysis-guided framework for generating test assertions using LLMs. By supplying concrete assertion components, STARS relieves LLMs from synthesizing valid assertions from scratch, allowing them to focus on generating more meaningful and contextually relevant assertions. Experimental results demonstrate that STARS achieves up to 2x improvements in mutation score and test strength over the baseline, sometimes approaching or surpassing developer-written assertions.

Data-Availability Statement

We make our artifact, including the experiment results and the replication code, publicly available at https://doi.org/10.5281/zenodo.21818614 [11] to facilitate follow-up research studies.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work was partially supported by the National Natural Science Foundation of China (No.62372193, No.U2436207, and No. 92582201), the Hong Kong SAR Research Grant Council/General Research Fund (Ref No. 16210725), and the Hong Kong SAR Research Grant Council/Theme-based Research Scheme (Ref No. T41-517/25-N), the Major Technological Innovation Program of HUST (No. 2025ZDKJCX05), and the German Research Foundation (DFG; projects 492507603, 516334526, and 526259073).

References

[1] Juan C. Alonso, Sergio Segura, and Antonio Ruiz-Cortés. 2023. AGORA: Automated Generation of Test Oracles for REST APIs. 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, 1018–1030. doi:10.1145/3597926.3598114

[2] Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 775–787. doi:10.1145/3314221.3314641

[3] Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing contracts correct modulo a test generator. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 104 (Oct. 2021), 27 pages. doi:10.1145/3485481

[4] Jon Ayerdi, Valerio Terragni, Gunel Jahangirova, Aitor Arrieta, and Paolo Tonella. 2024. GenMorph: Automatically Generating Metamorphic Relations via Genetic Programming. *IEEE Transactions on Software Engineering* 50, 7 (2024), 1888–1900. doi:10.1109/TSE.2024.3407840

[5] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[6] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. 2024. Unit Test Generation using Generative AI: A Comparative Performance Analysis of Autogeneration Tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code* (Lisbon, Portugal) *(LLM4Code '24)*. Association for Computing Machinery, New York, NY, USA, 54–61. doi:10.1145/3643795.3648396

[7] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 242–253. doi:10.1145/3213846.3213872

[8] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2023. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 19, 11 pages. doi:10.1145/3551349.3556961

[9] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *J. Syst. Softw.* 181, C (Nov. 2021), 13 pages. doi:10.1016/j.jss.2021.111041

[10] Max Brunsfeld, Amaan Qureshi, Andrew Hlynskyi, Will Lillis, ObserverOfTime, Christian Clason, dundargoc, Phil Turnbull, Timothy Clem, Douglas Creager, Andrew Helwer, Antonin Delpeuch, Riley Bruins, Daumantas Kavolis, Michael Davis, Ika, bfredl, Tuan-Anh Nguyen, Amin Ya, Stafford Brunk, Artem Krinitsyn, Matt Massicotte, Niranjan Hasabnis, Rob Rix, James McCoy, Mingkai Dong, Samuel Moelius, Steven Kalt, and Josh Vera. 2026. *tree-sitter/tree-sitter: nightly*. doi:10.5281/zenodo.21898578

[11] Jialun Cao, Haoyu Wang, Haoran Yan, Ming Wen, and Michael Pradel. 2026. *Replication Package of STARS: Static Analysis-guided Assertion Synthesis Using LLMs*. doi:10.5281/zenodo.21818614

[12] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 572–576. doi:10.1145/3663529.3663801

[13] Lin Chin-Yew. 2004. Rouge: A package for automatic evaluation of summaries. In *Proceedings of the Workshop on Text Summarization Branches Out, 2004*. https://aclanthology.org/W04-1013/

[14] Hugh A. Chipman, Edward I. George, and Robert E. McCulloch. 2010. BART: Bayesian additive regression trees. *The Annals of Applied Statistics* 4, 1 (2010), 266 – 298. doi:10.1214/09-AOAS285

[15] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 449–452. doi:10.1145/2931037.2948707

[16] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C. Desmarais. 2024. Effective test generation using pre-trained Large Language Models and mutation testing. *Inf. Softw. Technol.* 171, C (July 2024), 17 pages. doi:10.1016/j.infsof.2024.107468

[17] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings (Lecture Notes in Computer Science, Vol. 952)*, Walter G. Olthoff (Ed.). Springer, 77–101. doi:10.1007/3-540-49538-X_5

[18] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, and so on. 2025. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] https://arxiv.org/abs/2412.19437

[19] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. doi:10.1109/C-M.1978.218136

[20] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: a neural method for test oracle generation. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2130–2141. doi:10.1145/3510003.3510141

[21] EclEmma Team. 2024. JaCoCo: Java Code Coverage Library. https://www.jacoco.org/jacoco/.

[22] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2023. Out of the BLEU: How should we assess quality of the Code Generation models? *J. Syst. Softw.* 203 (2023), 111741. doi:10.1016/J.JSS.2023.111741

[23] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 599–612. doi:10.1145/3009837.3009851

[24] Harry Foster, Adam Krolnik, and David Lacey. 2004. *Assertion-based design, Second Edition*. Kluwer. doi:10.1007/b117047

[25] Merna D Galassi and John P Galassi. 1978. Assertion: A critical review. *Psychotherapy: Theory, Research & Practice* 15, 1 (1978), 16. doi:10.1037/h0085834

[26] Gwihwan Go, Quan Zhang, Chijin Zhou, Zhao Wei, and Yu Jiang. 2025. LSPRAG: LSP-Guided RAG for Language-Agnostic Real-Time Unit Test Generation. *CoRR* abs/2510.22210. doi:10.48550/ARXIV.2510.22210 arXiv:2510.22210

[27] Sijia Gu, Noor Nashid, and Ali Mesbah. 2026. LLM Test Generation via Iterative Hybrid Program Analysis. In *48th IEEE/ACM International Conference on Software Engineering ICSE 2026, Rio de Janeiro, April 12-18, 2026*. Rio de Janeiro, Brazil. https://doi.org/10.48550/arXiv.2503.13580

[28] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. doi:10.18653/V1/2022.ACL-LONG.499

[29] Richard G. Hamlet. 1977. Testing Programs with the Aid of a Compiler. *IEEE Trans. Software Eng.* 3, 4 (1977), 279–290. doi:10.1109/TSE.1977.231145

[30] Ishrak Hayet, Adam Scott, and Marcelo d'Amorim. 2025. ChatAssert: LLM-Based Test Oracle Generation With External Tools Assistance. *IEEE Transactions on Software Engineering* 51, 1 (2025), 305–319. doi:10.1109/TSE.2024.3519159

[31] Yibo He, Jiaming Huang, Hao Yu, and Tao Xie. 2024. An Empirical Study on Focal Methods in Deep-Learning-Based Approaches for Assertion Generation. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1750–1771. doi:10.1145/3660785

[32] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rygGQyrFvH

[33] Soneya Binta Hossain and Matthew B. Dwyer. 2025. TOGLL: Correct and Strong Test Oracle Generation with LLMS. 1475–1487 pages. doi:10.1109/ICSE55347.2025.00098

[34] Soneya Binta Hossain, Antonio Filieri, Matthew B. Dwyer, Sebastian Elbaum, and Willem Visser. 2023. Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned. 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, 120–132. doi:10.1145/3611643.3616265

[35] Soneya Binta Hossain, Raygan Taylor, and Matthew Dwyer. 2025. Doc2OracLL: Investigating the Impact of Documentation on LLM-Based Test Oracle Generation. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE084 (June 2025), 22 pages. doi:10.1145/3729354

[36] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[37] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. 2024. Do LLMs generate test oracles that capture the actual or the expected program behaviour? arXiv:2410.21136 https://arxiv.org/abs/2410.21136

[38] Anji Li, Mingwei Liu, Zhenxi Chen, Zheng Pei, Zike Li, Dekun Dai, Yanlin Wang, and Zibin Zheng. 2026. Knowledge Matters: Injecting Project and Testing Knowledge into LLM-based Unit Test Generation. In *48th IEEE/ACM International Conference on Software Engineering, ICSE 2026, Rio de Janeiro, April 12-18, 2026*. IEEE/ACM, Rio de Janeiro, Brazil. https://arxiv.org/abs/2511.14224

[39] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Trans. Assoc. Comput. Linguistics* 12 (2024), 157–173. doi:10.1162/TACL_A_00638

[40] Zhuang Liu, Hailong Wang, Tongtong Xu, and Bei Wang. 2025. RAG-Driven multiple assertions generation with large language models. *Empir. Softw. Eng.* 30, 3 (2025), 105. doi:10.1007/S10664-025-10641-1

[41] Quang-Hung Luu, Huai Liu, and Tsong Yueh Chen. 2023. Can ChatGPT advance software testing intelligence? An experience report on metamorphic testing. arXiv:2310.19204 https://arxiv.org/abs/2310.19204

[42] Neri Marsili. 2020. The definition of assertion. *SSRN 3711804* (2020). doi:10.2139/ssrn.3711804

[43] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. 2025. Test Oracle Automation in the Era of LLMs. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 150 (May 2025), 24 pages. doi:10.1145/3715107

[44] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1223–1235. doi:10.1109/ICSE43902.2021.00112

[45] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2450–2462. doi:10.1109/ICSE48619.2023.00205

[46] Chao Ni, Xiaoya Wang, Liushan Chen, Dehai Zhao, Zhengong Cai, Shaohua Wang, and Xiaohu Yang. 2024. CasModaTest: A Cascaded and Model-agnostic Self-directed Framework for Unit Test Generation. arXiv:2406.15743 https://arxiv.org/abs/2406.15743

[47] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2111–2123. doi:10.1109/ICSE48619.2023.00178

[48] OpenAI. 2023. GPT-3.5-turbo Model Availability. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models#gpt-35-turbo-model-availability.

[49] OpenAI. 2026. Introducing GPT-5.4. https://openai.com/index/introducing-gpt-5-4/

[50] Wendkûuni C. Ouédraogo, Kader Kaboré, Yinghua Li, Haoye Tian, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. 2026. Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation. doi:10.48550/arXiv.2407.00225 arXiv:2407.00225

[51] Rangeet Pan, Raju Pavuluri, Ruikai Huang, Rahul Krishna, Tyler Stennett, Alessandro Orso, and Saurabh Sinha. 2026. SAINT: Service-level Integration Test Generation with Program Analysis and LLM-based Agents. In *48th IEEE/ACM International Conference on Software Engineering, ICSE 2026, Rio de Janeiro, April 12-18, 2026*. Rio de Janeiro, Brazil. https://doi.org/10.48550/arXiv.2511.13305

[52] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, Pierre Isabelle, Eugene Charniak, and Dekang Lin (Eds.). Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. doi:10.3115/1073083.1073135

[53] Michael Pradel. 2026. Testora: Using Natural Language Intent to Detect Behavioral Regressions. In *48th IEEE/ACM International Conference on Software Engineering, ICSE 2026, Rio de Janeiro, April 12-18, 2026*. Rio de Janeiro, Brazil. doi:10.48550/arXiv.2503.18597

[54] Ketai Qiu, Luca Di Grazia, Leonardo Mariani, and Mauro Pezzè. 2026. E-Test: E'er-Improving Test Suites. In *48th IEEE/ACM International Conference on Software Engineering ICSE 2026, Rio de Janeiro, April 12-18, 2026*. Rio de Janeiro, Brazil. https://doi.org/10.48550/arXiv.2510.19860

[55] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. doi:10.48550/arXiv.2009.10297 arXiv:2009.10297 [cs].

[56] Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel Briand. 2025. Automated Test Case Repair Using Language Models. *IEEE Trans. Softw. Eng.* 51, 4 (Feb. 2025), 1104–1133. doi:10.1109/TSE.2025.3541166

[57] Jiho Shin, Nima Shiri Harzevili, Reem Aleithan, Hadi Hemmati, and Song Wang. 2026. Retrieval-augmented test generation: How far are we?. In *48th IEEE/ACM International Conference on Software Engineering, ICSE 2026, Rio de Janeiro, April 12-18, 2026*. IEEE/ACM, Rio de Janeiro, Brazil. https://doi.org/10.48550/arXiv.2409.12682

[58] Jiho Shin, Hadi Hemmati, Moshi Wei, and Song Wang. 2024. Assessing Evaluation Metrics for Neural Test Oracle Generation. *IEEE Transactions on Software Engineering* 50, 9 (Sept. 2024), 2337–2349. doi:10.1109/tse.2024.3433463

[59] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, Akshay Nathan, Alan Luo, Alec Helyar, Aleksander Madry, Aleksandr Efremov, Aleksandra Spyra, et al. 2026. OpenAI GPT-5 System Card. (2026). doi:10.48550/arXiv.2601.03267 arXiv:2601.03267 [cs.CL]

[60] Weifeng Sun, Hongyan Li, Meng Yan, Yan Lei, and Hongyu Zhang. 2024. Revisiting and Improving Retrieval-Augmented Deep Assertion Generation. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE '23)*. IEEE Press, Echternach, Luxembourg, 1123–1135. doi:10.1109/ASE56229.2023.00090

[61] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary improvement of assertion oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1178–1189. doi:10.1145/3368089.3409758

[62] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers and Focal Context. doi:10.48550/arXiv.2009.05617 arXiv:2009.05617

[63] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test* (Pittsburgh, Pennsylvania) *(AST '22)*. Association for Computing Machinery, New York, NY, USA, 54–64. doi:10.1145/3524481.3527220

[64] Hailong Wang, Tongtong Xu, and Bei Wang. 2024. Deep Multiple Assertions Generation. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering* (Lisbon, Portugal) *(FORGE '24)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3650105.3652293

[65] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. doi:10.18653/V1/2021.EMNLP-MAIN.685

[66] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. doi:10.1145/3377811.3380429

[67] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). https://doi.org/10.48550/arXiv.2201.11903

[68] Congying Xu, Songqiang Chen, Jiarong Wu, Shing-Chi Cheung, Valerio Terragni, Hengcheng Zhu, and Jialun Cao. 2024. MR-Adopt: Automatic Deduction of Input Transformation Function for Metamorphic Testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 557–569. doi:10.1145/3691620.3696020

[69] Congying Xu, Valerio Terragni, Hengcheng Zhu, Jiarong Wu, and Shing-Chi Cheung. 2024. MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases. *ACM Trans. Softw. Eng. Methodol.* 33, 6, Article 150 (June 2024), 28 pages. doi:10.1145/3656340

[70] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html

[71] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 163–174. doi:10.1145/3510003.3510149

[72] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 76 (July 2024), 24 pages. doi:10.1145/3660783

[73] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2025. Exploring Automated Assertion Generation via Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 34, 3, Article 81 (Feb. 2025), 25 pages. doi:10.1145/3699598

[74] Yuwei Zhang, Zhi Jin, Ze-Jun Wang, Ying Xing, and Ge Li. 2025. SAGA: Summarization-Guided Assert Statement Generation. *J. Comput. Sci. Technol.* 40, 1 (2025), 138–157. doi:10.1007/S11390-023-2878-6

[75] Zhiyuan Zhong, Sinan Wang, Hailong Wang, Shaojin Wen, Hao Guan, Yida Tao, and Yepang Liu. 2024. Advancing Bug Detection in Fastjson2 with Large Language Models Driven Unit Test Generation. arXiv:2410.09414 https://arxiv.org/abs/2410.09414