Retrieval-Augmented Fine-Tuning for Improving Retrieve-and-Edit Based Assertion Generation

Hongyan Li*, Weifeng Sun*, Meng Yan†, Ling Xu, Qiang Li, Xiaohong Zhang, Hongyu Zhang

Abstract—Unit Testing is crucial in software development and maintenance, aiming to verify that the implemented functionality is consistent with the expected functionality. A unit test is composed of two parts: a test prefix, which drives the unit under test to a particular state, and a test assertion, which determines what the expected behavior is under that state. To reduce the effort of conducting unit tests manually, Yu et al. proposed an integrated approach (*integration* for short), combining information retrieval with a deep learning-based approach to generate assertions for test prefixes, and obtained promising results. In our previous work, we found that the overall performance of *integration* is mainly due to its success in retrieving assertions. Moreover, *integration* is limited to specific types of edit operations and struggles to understand the semantic differences between the retrieved focal-test (*focal-test* includes a test prefix and a unit under test) and the input focal-test. Based on these insights, we then proposed a retrieve-and-edit approach named EditAS to learn the assertion edit patterns to improve the effectiveness of assertion generation in our prior study. Despite being promising, we find that the effectiveness of EditAS can be further improved. Our analysis shows that: ① The editing ability of EditAS still has ample room for improvement. Its performance degrades as the edit distance between the retrieval assertion and ground truth increases. Specifically, the average accuracy of EditAS is 12.38% when the edit distance is greater than 5. ② EditAS lacks a fine-grained semantic understanding of both the retrieved focal-test and the input focal-test themselves, which leads to many inaccurate token modifications. In particular, an average of 25.57% of the incorrectly generated assertions that need to be modified are not modified, and an average of 6.45% of the assertions that match the ground truth are still modified. Thanks to pre-trained models employing pre-training paradigms on large-scale data, they tend to have good semantic comprehension and code generation abilities. In light of this, we propose $EditAS^2$, which improves retrieval-and-edit based assertion generation through retrieval-augmented fine-tuning. Specifically, $EditAS^2$ first retrieves a similar focal-test from a predefined corpus and treats its assertion as a prototype. Then, $EditAS^2$ uses a pre-trained model, CodeT5, to learn the semantics of the input and similar focal-tests as well as assertion editing patterns to automatically edit the prototype. We first evaluate the $EditAS^2$ for its inference performance on two large-scale datasets, and the experimental results show that $EditAS^2$ outperforms state-of-the-art assertion generation methods and pre-trained models, with average performance improvements of 15.93%-129.19% and 11.01%-68.88% in accuracy and CodeBLEU, respectively. We also evaluate the performance of $EditAS^2$ in detecting real-world bugs from Defects4J. The experimental results indicate that $EditAS^2$ achieves the best bug detection performance among all the methods.

Index Terms—Unit Testing, Assertion Generation, Test Assertion, Pre-trained Model, Fine-Tuning.

1 Introduction

Throughout the software lifecycle, unit testing has become an indispensable part. Unit tests for a component (a method, class, or module), serve to document the expected usage and verify that the implemented functionality aligns with the expected functionality. This practice contributes to the early detection and diagnosis of failures, preventing them from spreading throughout the system and avoiding system regressions. Effective unit tests can improve software quality, reduce the frequency and expense of software failures [1], [2], as well as enhance the entire software development process.

```
1 @Test
2 public void test01() throws Throwable  {
3     CategoryAxis3D categoryAxis3D0 = new CategoryAxis3D();
4     categoryAxis3D0.setVisible(false);
5     CategoryAxis3D categoryAxis3D1 = new CategoryAxis3D();
6     boolean boolean0 = categoryAxis3D1.equals(categoryAxis3D0);
7     assertFalse(categoryAxis3D0.isVisible());
8     assertFalse(boolean0);
9 }
```

Fig. 1. Example of a unit test case.

A unit test consists of two parts: a test prefix, which drives the unit under test to a particular state, and a test oracle (e.g., assertion), which determines what the expected behavior is under that state [3]. For example, in the unit test shown in Figure 1, lines 3-6 are test prefixes that create two `CategoryAxis3D` objects, `categoryAxis3D0` and `categoryAxis3D1`, with `categoryAxis3D0` set to invisible. A Boolean variable `boolean0` is obtained by checking if `categoryAxis3D0` and `categoryAxis3D1` are equal. Lines 7-8 provide test assertions that specify the oracle. Specifically, the assertions state that the visibility property of `categoryAxis3D0` should be `False` and the two `categoryAxis3D` objects should not equal once the test prefix has been executed.

While the advantages of unit testing are evident, creating effective unit tests is a non-trivial and time-consuming task. Prior research suggests that developers allocate over 15% of their time to generate tests [4]. To reduce manual efforts,

* *Both authors contributed equally to this research*
† *Meng Yan is the corresponding author*

- *Hongyan Li, Weifeng Sun, Meng Yan, Ling Xu, Qiang Li, Xiaohong Zhang, and Hongyu Zhang are with the Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education, Chongqing University, Chongqing 400044, China, and with the School of Big Data and Software Engineering, Chongqing University, Chongqing 400044, China.*
  E-mail: hongyan.li@cqu.edu.cn, weifeng.sun@cqu.edu.cn, mengy@cqu.edu.cn, xuling@cqu.edu.cn, liqiang@stu.cqu.edu.cn, xhongz@cqu.edu.cn, hyzhang@cqu.edu.cn

various automated testing tools, including Randoop [5] and EvoSuite [6], have been proposed to automate unit test generation. However, these testing tools prioritize high-coverage test generation but struggle to understand the expected behavior of generated test prefixes. This limitation leads to meaningless assertions that fail to reveal functional bugs in the unit under test. As an illustration, an industrial evaluation [7] of assertions generated by EvoSuite revealed that "assertions are meaningful and useful in hand-written tests, unlike generated assertions". Therefore, these automated unit test generation tools cannot replace manually written unit tests.

To solve the problem mentioned above, a number of assertion generation techniques have been proposed. These methods take a pair consisting of the test prefix and its *focal method* (i.e., the method under test), called focal-test as input, and output the appropriate assertion. They are primarily categorized into two main groups: deep learning-based (DL-based) and information retrieval-based (IR-based). DL-based approaches treat assertion generation as a translation task, such as ATLAS [8], which uses neural machine translation (NMT) for "translation". Recent studies [9], [10] have investigated the use of pre-trained models for assertion generation. For instance, Yuan et al. [9] fine-tuned pre-trained models like CodeGPT [11] and CodeT5 [12] specifically for this task. Additionally, Nashid et al. [10] employed prompt engineering with few-shot learning, querying the black-box large language model (LLM) Codex [13] to generate assertions. However, Deep learning-based approaches generally prefer high-frequency words in the corpus and may have trouble with low-frequency words, such as project-specific identifiers. Unlike deep learning-based approaches, retrieval-based approaches retrieve assertions from the corpus that are most similar to a given focal-test and then further adjust the tokens in the retrieved assertions. Yu et al. [14] introduces IR-based assertion retrieval (IR_{ar}) and retrieval assertion adaptation (RA_{adapt}) techniques, and proposes an integrated approach (*Integration* for short) combining an IR-based method with a DL-based method. Integration uses a threshold to decide whether to use the IR-based or DL-based method.

To improve assertion generation capabilities, we proposed a novel retrieve-and-edit approach named EDITAS in our previous work [15]. It treats the retrieved assertion as a prototype and utilizes a neural sequence-to-sequence model to capture fine-grained semantic differences between focal-tests to modify the prototype. Despite being promising, after a systematic inspection, we find that the effectiveness of EDITAS can be further improved. To understand the application of this mechanism, we perform an in-depth analysis of EDITAS on two publicly available datasets adopted by Yu et al [14], namely $Data_{atlas}$ and $Data_{integration}$. Specifically, both $Data_{atlas}$ and $Data_{integration}$ are derived from the original dataset proposed by ATLAS [8]. $Data_{atlas}$ is created by removing certain samples, such as those that could not be tokenized. In contrast, $Data_{integration}$ is expanded to include previously excluded cases with unknown tokens. The main findings are: (1) EDITAS *could be improved in editing ability.* While EDITAS exhibits a significant performance boost for longer edit distances, its effectiveness diminishes as the edit distance (E) increases. For instance, when $E = 1$, EDITAS correctly generates an average of 44.21% of assertions for $Data_{integration}$ and $Data_{atlas}$, when $E = 3$, the average accuracy drops to 29.39%. In addition, a significant proportion of the incorrect assertions generated by EDITAS are very close to ground truth. For instance, an average of 23.14% of the incorrect assertions generated by EDITAS have only one token different from the ground truth. (2) EDITAS *could be improved in semantic understanding ability.* Even when it finds assertions that exactly match the ground truth, it may incorrectly modify them, while not modifying those that need to be modified. In particular, an average of 25.57% of the incorrectly generated assertions that need to be modified are not modified, and an average of 6.45% of the assertions that match the ground truth are still modified.

To alleviate the limitations mentioned above and enhance performance, we utilize pre-trained models to take full advantage of the benefits they gain from pre-training on large-scale data. In this study, we specifically choose CodeT5 [12] because its encoder-decoder architecture is better suited for comprehension and generation tasks, which corresponds to the semantic understanding and editing performance required for our task. Additionally, CodeT5 has been widely and effectively used for a variety of code-related tasks [16], [17]. Therefore, we propose $EditAS^2$, an innovative approach that improves retrieval-and-edit based assertion generation through retrieval-augmented fine-tuning. Specifically, $EditAS^2$ consists of two key components: a retrieval component and an editing component. In the retrieval component, $EditAS^2$ retrieves a similar focal-test from a predefined corpus and treats its assertion as a prototype. In the editing component, $EditAS^2$ utilizes CodeT5, a pre-trained model, to learn the semantic differences between input and similar focal-tests, and automatically edit the prototype. To enhance CodeT5's understanding of the semantic differences between input and similar focal-tests, we feed CodeT5 with the edit sequence of both focal-tests and the focal-tests themselves. Overall, $EditAS^2$ is a significant improvement over EDITAS in the following ways: **(1) Enhanced Editing Component:** We replace the neural sequence-to-sequence editing component of EDITAS with a pre-trained model (i.e. CodeT5) trained on a large corpus. This upgrade aims to improve the model's semantic understanding and editing capabilities. **(2) Dual-Input Strategy:** Taking advantage of the pre-trained model's strengths in text processing, we provide both the editing sequence and the focal-tests themselves to the editing component, while EDITAS provides only the editing sequence. This dual-input strategy is designed to maximize the semantic comprehension of the pre-trained model, thereby improving the overall performance.

We evaluate the inference performance and bug detection ability of $EditAS^2$. The experimental results show that $EditAS^2$ outperforms all state-of-the-art assertion generation methods and pre-trained methods in terms of inference performance, with an average improvement of 15.93%-129.19% and 11.01%-68.88% in accuracy and CodeBLEU score, respectively. In terms of bug detection ability, $EditAS^2$ also outperforms all the baselines.

In summary, our contributions are as follows:

- **New Insight.** We perform a thorough investigation of EDITAS, the state-of-the-art assertion generation method. The results of our investigation provide insight for further studies in this direction.

- **Novel Approach.** We propose $EditAS^2$, a novel approach that improves retrieval-and-edit based assertion generation through retrieval-augmented fine-tuning. $EditAS^2$ utilizes assertions from similar focal-tests as prototypes and uses a pre-trained model to learn the semantics of the focal-tests and assertion editing patterns.
- **Effectiveness.** We conduct extensive experiments to evaluate our approach on two datasets for its inference and bug-finding performance. The experimental results show that $EditAS^2$ outperforms all baselines, demonstrating its robustness and effectiveness.
- **New Dataset and Open Science** [1]**.** We construct a new cross-project test set that has no overlap with the projects in the training set. Our replication package includes the cross-project test set, the source code of $EditAS^2$, the trained model, and assertion generation result for follow-up studies.

As an extended version of our previous work [15], which proposed and evaluated EDITAS, this paper has been significantly extended in the following ways:

- **Gaining Insights Through Empirical Study.** We conduct a new empirical study to provide insight into EDITAS. The results further illustrate the effectiveness of the retrieve-and-edit model structure employed by EDITAS, but there is room for improvement in terms of the model's semantic understanding and editing abilities.
- **Introducing a Novel Solution.** To enhance the semantic understanding and editing abilities of EDITAS, we propose a novel approach that improves retrieval-and-edit based assertion generation through retrieval-augmented fine-tuning, namely $EditAS^2$.
- **Constructing a New Dataset.** We construct a new cross-project test set, ensuring no overlap with the projects used in the training set, to evaluate the robustness of assertion generation models in cross-project scenarios.
- **Incorporating Diverse Baselines.** In addition to the comparison with state-of-the-art assertion generation approaches, we have included comparisons with fine-tuned pre-trained models.
- **Adopting Code-Specific Evaluation Metrics.** Instead of BLEU, we use a new evaluation metric called CodeBLEU, a widely adopted metric for code generation tasks that goes beyond the limitations of BLEU. Recent studies [18] have revealed the inadequacy of BLEU in considering grammatical and logical correctness, which has led to a preference for candidates exhibiting high n-gram accuracy but substantial logical errors.
- **Extending Evaluation to Bug Detection.** We evaluate the proposed approach against comprehensive baselines in two different performance aspects. In our previous work, only the inference performance of the model was evaluated, i.e. the correctness and similarity between the generated assertions and the manually written ones. In this work, we add the evaluation of the approach in terms of bug-finding performance. The evaluation results demonstrate its effectiveness.

2 Background and Related Work

2.1 Automated Assertion Generation

A number of approaches have been proposed to automatically generate assertions, which can be categorized as deep learning-based (DL-based) and information retrieval-based (IR-based).

2.1.1 DL-based Assertion Generation

DL-based approaches design neural networks to generate assertions from scratch. For example, the first learning-based assertion generation method, ATLAS [8], was recently proposed by Watson et al. ATLAS employs Neural Machine Translation (NMT) to generate assertions for a given *focal-test*, which consists of a test method without any assertion (i.e., test prefix) and its focal method (i.e., the method under test), including both method names and method bodies. To develop ATLAS, Watson et al. extracted Test-Assert Pairs (TAPs) from GitHub projects that use the JUnit testing framework. Each pair consists of a focal-test and its corresponding assertion. Dinella [3] proposed TOGA, which solves the assertion generation problem by using a ranking architecture on a set of candidate test assertions. Unlike TOGA, this paper focuses only on the generation of assertions using focal-tests. Recent research [19]–[21] has explored the potential of pre-trained models (e.g., T5 [22] and BART [23]) to support the task of assertion generation through pre-training and fine-tuning. Specifically, these approaches involve pre-training a transformer model using a large source code or English corpus and then fine-tuning it for assertion generation. In contrast, our work focuses on using a pre-trained model as an assertion editor to edit the retrieved assertion into a correct assertion based on the semantic differences between the input focal-test and the similar focal-test. Nie et al. [24] proposed TECO, a deep learning model for test completion. It predicts the next statement in a test method by using six types of code semantics, including execution results and context. The model outperforms previous approaches in generating syntactically correct and runnable test code. By incorporating execution reasoning, TECO addresses issues that purely syntax-based methods face, making it more effective for practical test generation tasks. Wang et al. [25] introduced DeepAssert, a deep learning model that leverages the pre-trained GraphCodeBERT to generate multiple assertions for test methods. DeepAssert predicts a sequence of assertions, effectively addressing the need for test cases to include multiple assertions.

2.1.2 IR-based Assertion Generation

Inspired by the application of information retrieval in SE, Yu et al. [14] introduced *Integration* for assertion generation based on information retrieval (IR), including IR-based assertion retrieval (IR_{ar}) and retrieved-assertion adaptation (RA_{adapt}) techniques. IR_{ar} retrieve the assertion whose corresponding focal-test has the highest similarity (e.g., Jaccard [26] similarity) with the given focal-test. IR_{ar} retrieve

[1] https://github.com/Lhy-apple/EditAs2

the assertion whose corresponding focal-test has the highest similarity (e.g., Jaccard [26] similarity) with the given focal-test. RA_{adapt} then further adjusts the tokens in the retrieved assertion, taking into account the contextual information. RA_{adapt} has two adaptation strategies: one based on heuristics (RA^{H}_{adapt}) and the other on neural networks (RA^{NN}_{adapt}). RA^{H}_{adapt} utilizes lexical similarity for code replacement. In contrast to RA^{H}_{adapt}, RA^{NN}_{adapt} further enhances lexical similarity by incorporating semantic information using a neural network architecture and computes replacement values for code adaptation. Drawing on the idea of "combining IR and DL", *Integration* first retrieves assertions using Jaccard similarity and adjusts the retrieved assertions if necessary. Then, *Integration* uses a semantic compatibility inference model to compute the "compatibility" of the adjusted assertions with the current focal-test. If the compatibility is below a specified threshold, *Integration* switches to ATLAS to generate an assertion from scratch. Given that RA^{NN}_{adapt} outperforms other IR-based approaches, including IR_{ar} and RA^{H}_{adapt}, we adopt the combination of RA^{NN}_{adapt} and ATLAS to explore the optimal performance of *Integration*. Based on Integration, Sun et al. [15] proposed a retrieve-and-edit approach name EDITAS. EDITAS treats the retrieved assertion as a prototype and utilizes a neural sequence-to-sequence model to capture fine-grained semantic differences between focal-tests to modify the prototype. Zhang et al. [27] investigated the use of large language models (LLMs) for automated assertion generation. Their results show that LLMs, particularly CodeT5, outperform traditional methods in terms of accuracy and bug detection. They also introduced a retrieval-and-repair approach that combines assertion retrieval with LLMs to enhance performance.

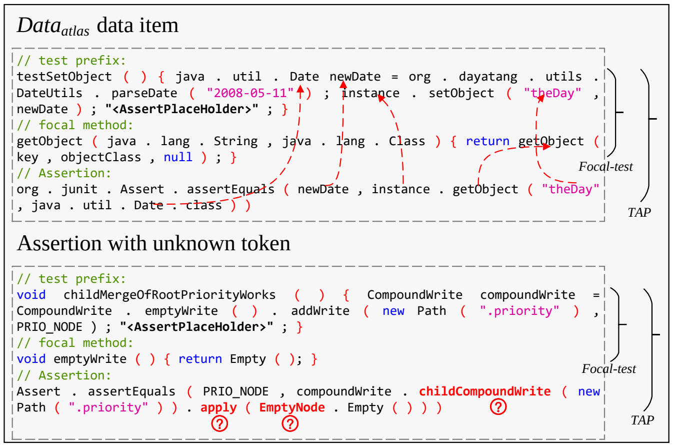
2.2 Pre-trained Models

Pre-trained models have demonstrated remarkable capabilities in software engineering across various domains [28]–[31]. They are trained on extensive unlabeled corpora to acquire deep linguistic knowledge and semantic representations. This training process gives them a strong ability to understand natural language and solve a variety of tasks [32]. Pre-trained models typically use the Transformer architecture [33], which can be broadly categorized into encoder-decoder models (e.g., UnixCoder [34] and CodeT5 [12]), encoder-only models (e.g., CodeBERT [35] and GraphCodeBERT [36]), and decoder-only (e.g., CodeGPT [11]) models. Among them, encoder-only models require an additional decoder for generation tasks, and this decoder needs to be initialized from scratch and cannot benefit from pre-training. In addition, decoder-only models perform well in autoregressive tasks (such as code completion), but their unidirectional framework may not be the best choice for understanding tasks. On the contrary, encoder-decoder models (e.g. CodeT5 [12]) are very suitable for both comprehension tasks and generation tasks. Therefore, we choose CodeT5 in this work to enhance the model's semantic understanding and editing abilities, as CodeT5 has been widely used for various code-related tasks [16], [17]. Specifically, we employ CodeT5 to help the model understand the semantic differences between input focal-tests and similar focal-tests and edit the prototypes. In addition, we choose pre-trained models based on different architectural classes to compare their inference performance with $EditAS^{2}$.

3 A SYSTEMATIC EVALUATION OF EDITAS

Currently, EDITAS [15] is the state-of-the-art approach for test assertion generation. Therefore, we start with understanding the application of this mechanism and then analyze its effect, reveal its problems and causes, and ultimately look for ways to improve it. In particular, this section discusses the following research questions:

- **RQ1**: *Where and why does* EDITAS *work?* We explore the advantages of EDITAS for assertion generation. We analyze the assertions that are generated correctly.
- **RQ2**: *Where and why does* EDITAS *fail?* We further explore the weaknesses of EDITAS that are critical to how it can be improved. We analyze the assertions that are generated incorrectly.


Data$_{atlas}$ data item

```
// test prefix:
testSetObject ( ) { java . util . Date newDate = org . dayatang . utils .
DateUtils . parseDate ( "2008-05-11" ) ; instance . setObject ( "theDay" ,
newDate ) ; "<AssertPlaceHolder>" ; }
// focal method:
getObject ( java . lang . String , java . lang . Class ) { return getObject (
key , objectClass , null ) ; }
// Assertion:
org . junit . Assert . assertEquals ( newDate , instance . getObject ( "theDay"
, java . util . Date . class ) )
```

Assertion with unknown token

```
// test prefix:
void  childMergeOfRootPriorityWorks  (  )  {  CompoundWrite  compoundWrite  =
CompoundWrite  .  emptyWrite  (  )  .  addWrite  (  new  Path  (  ".priority"  )  ,
PRIO_NODE ) ; "<AssertPlaceHolder>" ; }
// focal method:
void emptyWrite ( ) { return Empty ( ); }
// Assertion:
Assert  .  assertEquals  (  PRIO_NODE  ,  compoundWrite  .  childCompoundWrite  (  new
Path ( ".priority" ) ) . apply ( EmptyNode . Empty ( ) ) )
```

Fig. 2. Assertion with known *vs.* unknown tokens.

3.1 Dataset

Two publicly available datasets from Yu et al, namely $Data_{atlas}$ and $Data_{integration}$, are used in this paper. To make the content of this paper self-contained, we briefly introduce these two datasets in the following.

1) Data$_{atlas}$. The dataset is derived from the original dataset proposed by ATLAS [8] and consists of test prefixes, corresponding focal methods, and assertions. Initially, $Data_{atlas}$ is extracted from the 2.5 million test methods available on GitHub. It is then preprocessed according to established natural language processing practices [37], [38], resulting in a collection of Test-Assert Pairs (TAPs). Each TAP consists of a focal test and its corresponding assertion, and has been tokenized as shown in the top part of Figure 2. In Addition, test methods with token lengths longer than 1K are excluded, and assertions containing *unknown* tokens that do not exist in the focal-test and vocabulary are filtered out. For example, the bottom assertion of Figure 2 contains the unknown marker words `childCompoundWrite`, `apply`, and `EmptyNode`. After removing duplicates, $Data_{atlas}$ contains a total of 156,760 data items (An example is shown in Figure 2) and is divided into training, validation, and test sets in a ratio of 8:1:1.

2) Data$_{integration}$. The exclusion of assertions with unknown tokens from $Data_{atlas}$ simplifies the assertion

TABLE 1
Detailed statistics of each type in $Data_{integration}$ and $Data_{atlas}$

AssertType	Total	Equals	True	That	NotNull	False	Null	ArrayEquals	Same	Other
$Data_{atlas}$	15,676	7,866 (50%)	2,783 (18%)	1,441 (9%)	1,162 (7%)	1,006 (6%)	798 (5%)	307 (2%)	311 (2%)	2 (0%)
$Data_{integration}$	26,542	12,557 (47%)	3,652 (14%)	3,532 (13%)	1,284 (5%)	1,071 (4%)	735 (3%)	362 (1%)	319 (1%)	3,030 (11%)

TABLE 2
EDIT DISTANCE (E) BETWEEN RETRIEVED ASSERTIONS AND GROUND TRUTH FOR TEST SETS (RA_{adapt}^{NN} VS EDITAS)

Dataset	Prediction	Approach	$E = 0$	$E = 1$	$E = 2$	$E = 3$	$E = 4$	$E = 5$	$E > 5$	Total
$Data_{atlas}$	correct	RA_{adapt}^{NN}	5612 (82.06%)	538 (7.87%)	385 (5.63%)	127 (1.86%)	92 (1.35%)	40 (0.58%)	45 (0.66%)	6839
		EDITAS	5093 (60.78%)	1225 (14.62%)	474 (5.66%)	241 (2.88%)	188 (2.24%)	151 (1.80%)	1008 (12.03%)	8380
	incorrect	RA_{adapt}^{NN}	75 (0.85%)	1841 (20.83%)	544 (6.16%)	472 (5.34%)	434 (4.91%)	457 (5.17%)	5014 (56.74%)	8837
		EDITAS	594 (8.14%)	1154 (15.82%)	455 (6.24%)	358 (4.91%)	338 (4.63%)	346 (4.74%)	4051 (55.52%)	7296
$Data_{integration}$	correct	RA_{adapt}^{NN}	9956 (92.55%)	438 (4.07%)	232 (2.16%)	68 (0.63%)	32 (0.30%)	7 (0.07%)	25 (0.23%)	10758
		EDITAS	9410 (79.93%)	1130 (9.60%)	398 (3.38%)	190 (1.61%)	127 (1.08%)	73 (0.62%)	445 (3.78%)	11773
	incorrect	RA_{adapt}^{NN}	157 (0.99%)	2623 (16.62%)	1339 (8.48%)	956 (6.06%)	840 (5.32%)	677 (4.29%)	9192 (58.24%)	15784
		EDITAS	703 (4.76%)	1931 (13.07%)	1173 (7.94%)	834 (5.65%)	745 (5.04%)	611 (4.14%)	8772 (59.39%)	14769

generation problem, making $Data_{atlas}$ unsuitable for reflecting the distribution of real-world data. Recognizing this limitation, Yu et al. [14] propose an extended dataset $Data_{integration}$ based on $Data_{atlas}$. $Data_{integration}$ introduces 108,660 previously excluded cases with unknown tokens. The final expanded dataset $Data_{integration}$ consists of 265,420 data items and is divided into training, validation, and test sets in a ratio of 8:1:1.

It is important to note that the data items in both datasets are not strictly partitioned by project, meaning that items from the same project, including focal methods and test cases, may appear in both the training and test sets. This setup closely reflects real-world software development practices, where focal methods and test cases within the same project are often related due to shared functionality, dependencies, or component reuse. To prevent data leakage, we have ensured that no identical data items appear in both the training and test sets. By reflecting this realistic scenarios, our approach is well-suited to generate meaningful assertions for interrelated and evolving test suites, providing a practical solution for incremental test development.

Table 1 presents detailed statistics for the test sets of both datasets, illustrating the distribution of assertion types.

3.2 RQ1: Where and why does EDITAS work?

Previous work [15] has demonstrated that the IR-based assertion generation method (RA_{adapt}^{NN}) contributes the most to *Integration*. Therefore, in RQ1, we analyze the correct assertions generated by EDITAS and RA_{adapt}^{NN} in comparison. Both methods first use the same retrieval mechanism to find similar assertions, and then modify retrieved assertion to generate the target assertion. To explore the progress of EDITAS in terms of modification performance, we count the number of modifications required for the method to generate the correct assertion, i.e., the edit distance between the retrieved assertion and ground truth. The statistics are shown in Table 2, which categorizes assertions into correct and incorrect results (see *column: "Prediction"*) and further classifies them according to the edit distance (see *column:"E"*) between the retrieved assertions and the ground truth.

First of all, the overall performance of EDITAS is higher than RA_{adapt}^{NN}. The effectiveness of EDITAS for $Data_{atlas}$ is $8,380/(8,380 + 7,296) = 53.46\%$, while the effectiveness of RA_{adapt}^{NN} is $6,839/(6,839 + 8,837) = 43.63\%$.

For the retrieved assertions that are exactly equal to the ground truth ($E = 0$), RA_{adapt}^{NN} outperforms EDITAS. For instance, in $Data_{integration}$, when $E = 0$, RA_{adapt}^{NN} correctly generated assertions accounted for 92.55% of assertions, and achieves an accuracy rate of $98.45\% = 9,956/(9,956 + 157)$. In contrast, EDITAS correctly generated assertions accounted for 79.93% of assertions, with an accuracy rate of $93.05\% = 9,410/(9,410 + 703)$. This is because RA_{adapt}^{NN} tends to retain retrieved assertions without modification, while EDITAS prefers to edit them. Despite this, EDITAS generates more correct assertions overall, indicating that its editing capability surpasses that of RA_{adapt}^{NN}. However, EDITAS still needs to be improved in terms of semantic understanding as it unnecessarily edits some of the correct assertions due to misinterpreting the semantics of input focal-tests versus retrieved focal-tests.

> ***Finding-1.1***: RA_{adapt}^{NN} tends to keep retrieved assertions without modification, while EDITAS tends to edit them. However, EDITAS performs better overall performance due to its stronger editing ability.

For retrieval assertions that are very similar to the ground truth, EDITAS outperforms the editing ability of RA_{adapt}^{NN} for both single and multiple edit distances. For example, when $E = 1$, the accuracy of RA_{adapt}^{NN} in $Data_{integration}$ is $14.31\% = (438/438 + 2,623)$, while EDITAS achieves $36.92\% = 1,130/(1,130 + 1,931)$. When $E = 3$, the accuracy of RA_{adapt}^{NN} in $Data_{integration}$ is $6.64\% = 68/(68 + 956)$, while the accuracy of EDITAS reaches $18.55\% = 190/(190 + 834)$.

> ***Finging-1.2***: In terms of adaptation operations, EDITAS outperforms the editing ability of RA_{adapt}^{NN} for both single and multiple edit distances.

EDITAS alleviates the limitations of edit distance length

TABLE 3
NUMBER OF EDITS (N) MADE BY DIFFERENT APPROACHES FOR INCORRECT ASSERTIONS (RA_{adapt}^{NN} VS EDITAS)

Dataset	Approach	$N=0$	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$	$N>5$	Total
$Data_{atlas}$	RA_{adapt}^{NN}	7601 (86.01%)	420 (4.75%)	348 (3.94%)	187 (2.12%)	104 (1.18%)	59 (0.67%)	118 (1.34%)	8837
	EDITAS	1424 (19.52%)	1030 (14.12%)	554 (7.59%)	344 (4.71%)	336 (4.61%)	311 (4.26%)	3297 (45.19%)	7296
$Data_{integration}$	RA_{adapt}^{NN}	14487 (91.78%)	529 (3.35%)	360 (2.28%)	183 (1.16%)	89 (0.56%)	44 (0.28%)	92 (0.58%)	15784
	EDITAS	4670 (31.62%)	1783 (12.07%)	1001 (6.78%)	839 (5.68%)	576 (3.90%)	470 (3.18%)	5430 (36.77%)	14769

TABLE 4
NUMBER OF EDITS (N) BETWEEN GROUND TRUTH AND INCORRECT ASSERTIONS GENERATED BY EDITAS

Dataset	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$
$Data_{atlas}$	2111 (28.93%)	479 (6.57%)	413 (5.66%)	453 (6.21%)	438 (6.00%)
$Data_{integration}$	2563 (17.35%)	1291 (8.74%)	1030 (6.97%)	859 (5.82%)	834 (5.65%)

and exhibits better performance. This improvement addresses the drawback associated with RA_{adapt}^{NN}, which is effective only for shorter edit distances. In $Data_{atlas}$, RA_{adapt}^{NN} correctly generated assertions with $E > 5$ accounted for 0.66% of assertions, and only achieves an accuracy rate of $0.89\% = 45/(45 + 5,014)$ for assertions with $E > 5$. In contrast, EDITAS achieves a higher proportion and higher accuracy, with assertions having $E > 5$ accounting for 12.03% of correctly generated assertions, and achieves an accuracy rate of $19.92\% = 1,008/(1,008 + 4,051)$ for assertions with $E > 5$.

> ***Finding-1.3***: In particular, EDITAS alleviates the limitation in terms of edit distance length and achieves better performance even at $E > 5$. On the whole, the retrieve-and-edit model structure employed by EDITAS is effective for the assertion generation task.

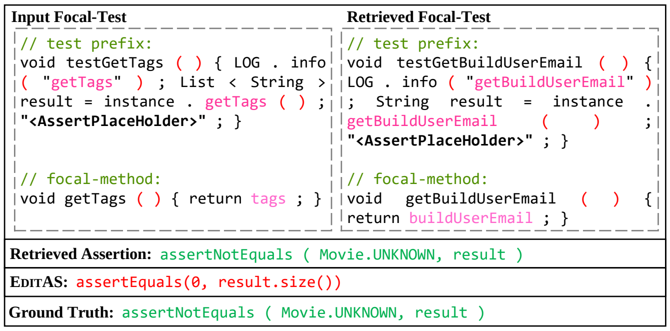
3.3 RQ2: Where and why does EDITAS fail?

Previous work [15] has shown that RA_{adapt}^{NN} has difficulty understanding semantic differences between focal-tests and has relatively low performance. Therefore, in RQ2, we analyze incorrect prediction results of EDITAS to gain insight into why EDITAS fails.

The editing ability of EDITAS *needs to be improved.* As stated above, Table 2 categorizes assertions according to whether they are generated correctly (see *column: "Prediction"*) and then classifies them according to the edit distance between the retrieved assertions and the ground truth (see *column:"E"*). we can observe that the performance of EDITAS is limited and decreases as the edit distance increases. For instance, when $E = 1$, EDITAS correctly generates $36.92\% = 1,130/(1,130 + 1,931)$ of assertions for $Data_{integration}$ and $51.49\% = 1,225/(1,225 + 1,154)$ for $Data_{atlas}$. When $E = 3$, the average accuracy decreases to 29.39%. When $E > 5$, EDITAS performs even worse, with an average accuracy of 12.38%. This emphasizes the importance of enhancing EDITAS's ability to handle longer edit distances.

> ***Finding-2.1***: The editing ability of EDITAS still has ample room for improvement. Its performance decreases as the edit distance increases. Specifically, the average accuracy of EDITAS is 12.38% when the $E > 5$.

The semantic understanding ability of EDITAS *needs to be improved.* ① As reported in Table 2, we can see that in $Data_{integration}$, there are 703 test samples (when $E = 0$) whose retrieved assertions match the ground truth but are still modified by EDITAS, and a similar phenomenon occurs in $Data_{atlas}$. For example, as shown in Figure 3, while the input focal-test and the retrieved focal-test differ in the methods invoked and their return types (*getTags()* returns a list of tags, and *getBuildUserEmail()* returns a string representing an email), their structural design and intent remain aligned. Both scenarios involve invoking a method to retrieve a value and validating its correctness. In this case, the retrieved assertion, *assertNotEquals(Movie. UNKNOWN, result)*, is appropriate, ensuring that the result is neither undefined nor a default value. However, EDITAS fails to recognize the semantic equivalence between the input and retrieved focal-tests. Instead of retaining the correct retrieved assertion, it modifies it unnecessarily to *assertEquals(0, result. size())*, resulting in an incorrect assertion. This example shows that EDITAS struggles to capture the core semantic similarities between the input and the retrieved focal-tests, leading to incorrect modifications to otherwise correct assertions, highlighting the need for better semantic understanding capabilities of EDITAS.



```
// test prefix:
void testGetTags ( ) { LOG . info
( "getTags" ) ; List < String >
result = instance . getTags ( ) ;
"<AssertPlaceHolder>" ; }


// focal-method:
void getTags ( ) { return tags ; }
```

```
// test prefix:
void  testGetBuildUserEmail  (  )  {
LOG . info ( "getBuildUserEmail" )
;  String  result  =  instance  .
getBuildUserEmail     (     )     ;
"<AssertPlaceHolder>" ; }

// focal-method:
void   getBuildUserEmail   (   )   {
return buildUserEmail ; }
```

Retrieved Assertion: `assertNotEquals ( Movie.UNKNOWN, result )`

EDITAS: `assertEquals(0, result.size())`

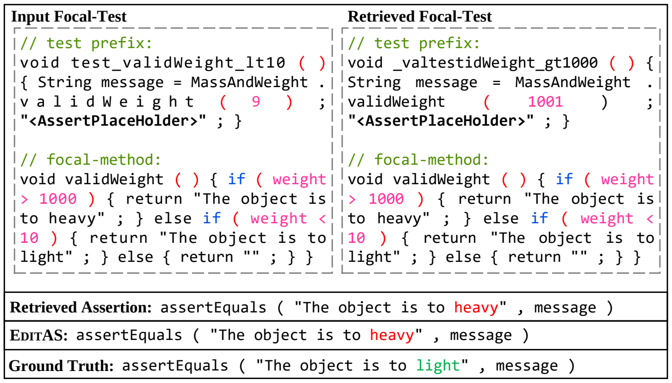
Ground Truth: `assertNotEquals ( Movie.UNKNOWN, result )`

Fig. 3. An example of an incorrect assertion generated by EDITAS: the retrieved assertion is unnecessarily modified despite semantic consistency between the input and retrieved focal-tests. The left example is from sample #249 in the test set of $Data_{integration}$, and the right example is from sample #9141 in the training set of $Data_{integration}$

② Furthermore, we count the number of edits made by different approaches for those incorrect assertions. As shown in Table 3, we can see that a large proportion of the incorrect assertions are not modified at all, but rather the retrieved assertions are output directly. In $Data_{integration}$, RA_{adapt}^{NN} has 14,487 such samples (when $N = 0$), accounting for 91.78% of the incorrect assertions. This may be due to the fact that RA_{adapt}^{NN} does not need to modify the retrieved assertions when all the tokens in the retrieved assertions are present in the input focal-test. To fill this shortcoming, EDITAS learns the edit patterns through the edit sequences between

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3558403

focal-tests and then automatically adjust retrieved assertions. However, the results show that, despite a reduction to 31.62% in $Data_{integration}$, this percentage remains relatively high. For example, as shown in Figure 4, the input focal-test checks whether a weight of 9 returns the message *"The object is too light"*, while the retrieved focal-test checks if a weight of 1001 returns *"The object is too heavy"*. Although these focal-tests share a similar structure, EDITAS does not adequately understand the semantic differences between them. Its failure to comprehend the distinct meanings behind the weight values in the tests results in an incorrect modification. Instead of adjusting the retrieved assertion to align with the input context, EDITAS outputs *assertEquals("The object is too heavy", message)* directly, without modification. The correct assertion, given the input test, should be *assertEquals("The object is too light", message)*. EDITAS demonstrates some ability to reduce the rate of unmodified assertions, but its limited semantic understanding still results in a high percentage of errors. Further refinement is necessary to improve its ability to handle subtle differences in focal-tests, ensuring more accurate assertion modifications. This example highlights another limitation: EDITAS struggles to modify assertions that require adjustment due to semantic differences, reflecting the need for a better understanding of the fine-grained semantics of focal-tests.


Input Focal-Test

```
// test prefix:
void test_validWeight_lt10 ( )
{ String message = MassAndWeight .
v a l i d W e i g h t  (  9  )  ;
"<AssertPlaceHolder>" ; }

// focal-method:
void validWeight ( ) { if ( weight
> 1000 ) { return "The object is
to heavy" ; } else if ( weight <
10 ) { return "The object is to
light" ; } else { return "" ; } }
```

Retrieved Focal-Test

```
// test prefix:
void _valtestidWeight_gt1000 ( ) {
String  message  =  MassAndWeight  .
validWeight   (   1001   )   ;
"<AssertPlaceHolder>" ; }

// focal-method:
void validWeight ( ) { if ( weight
> 1000 ) { return "The object is
to heavy" ; } else if ( weight <
10 ) { return "The object is to
light" ; } else { return "" ; } }
```

Retrieved Assertion: `assertEquals ( "The object is to heavy" , message )`

EDITAS: `assertEquals ( "The object is to heavy" , message )`

Ground Truth: `assertEquals ( "The object is to light" , message )`

Fig. 4. An example of incorrect assertion generated by EDITAS: the retrieved assertion is not modified as necessary based on the semantic differences between the input and the retrieved focal-tests. The left example is from sample #226 in the test set of $Data_{integration}$, and the right example is from sample #80591 in the training set of $Data_{integration}$.

In summary, EDITAS sometimes makes unnecessary changes to assertions or struggles to make the necessary changes, suggesting gaps in its semantic understanding. This may be due to its focus on the surface features of focal-tests' editing sequence and lack of a fine-grained semantic understanding of both the retrieved focal-test and the input focal-test. This emphasizes the need for further improvements in EDITAS' semantic understanding to ensure more accurate assertion generation and modification.

> ***Finding-2.2***: EDITAS could be improved in understanding semantic differences in focal-tests. In particular, an average of 25.57% of the incorrectly generated assertions that need to be modified are not modified, and an average of 6.45% of the assertions that match the ground truth are still modified.

A significant proportion of the incorrect assertions generated by EDITAS are very close to the ground truth. We count the number of edits between incorrect assertions and ground truth to explore how close the incorrect assertions generated by EDITAS are to the correct ones. As can be seen in Table 4, a significant proportion of the incorrectly generated assertions are very similar to ground truth, i.e., their edit distance is rather small. For example, among the $7,296$ incorrect assertions generated by EDITAS in $Data_{atlas}$, $2,111$ (i.e., 28.93%) assertions have only one different token from the ground truth, while $3,003$ (i.e., 41.16%) assertions have no more than three different tokens from the ground truth. Similarly, among the $14,769$ incorrect assertions generated by EDITAS in $Data_{integration}$, $2,563$ (i.e., 17.35%) assertions have only one different token from the ground truth, while $4,884$ (i.e., 33.07%) assertions have no more than three different tokens from the ground truth. To summarize, a significant proportion of the incorrect assertions generated by EDITAS are very similar to ground truth. In particular, many incorrect assertions may become correct after modifying only one token.

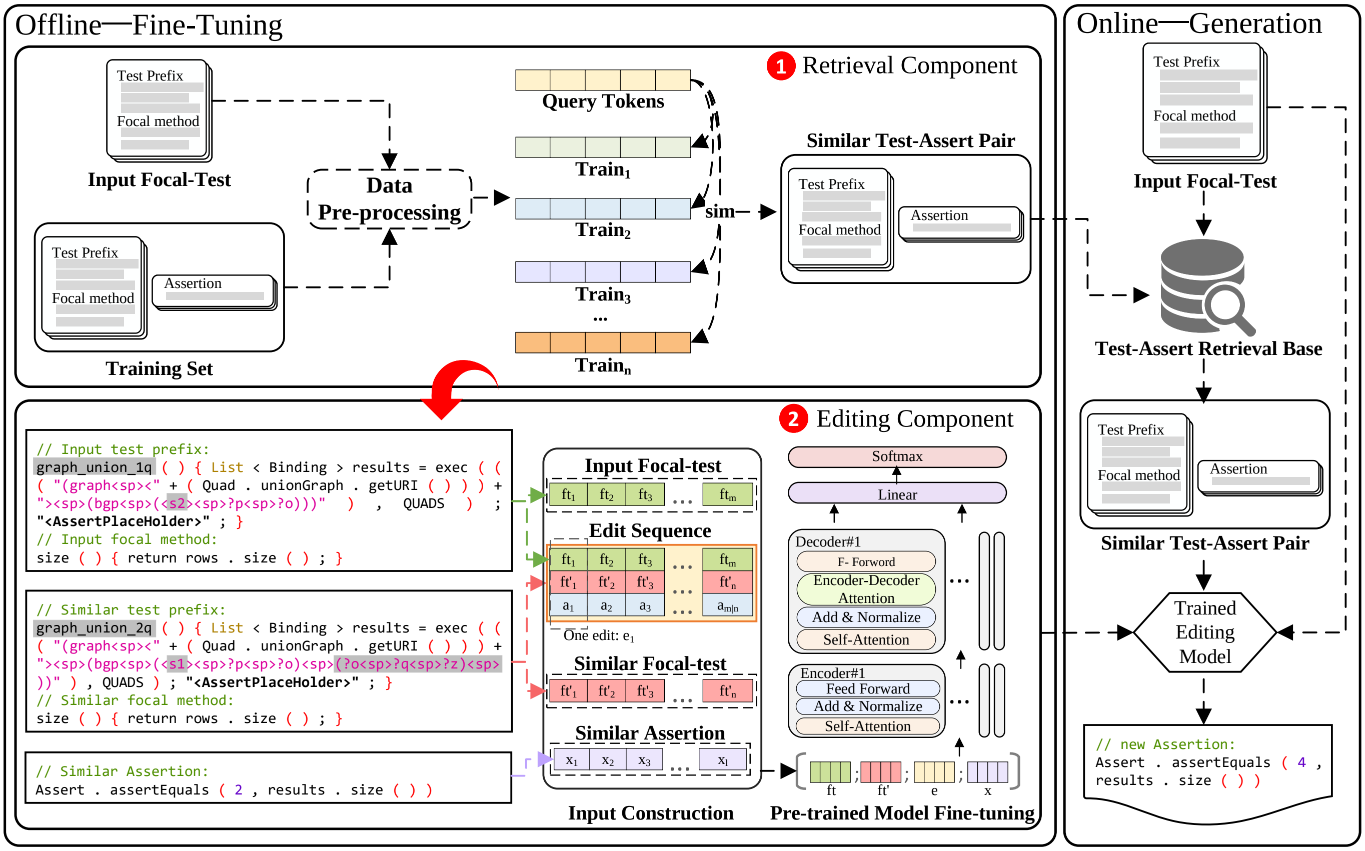
> ***Finding-2.3***: A significant proportion of the incorrect assertions generated by EDITAS are very similar to ground truth. On average, 23.14% of the incorrect assertions can be made correct by modifying only one token, while on average 37.12% assertions require modifying no more than three tokens.

4 APPROACH

Drawing from our empirical study, it is evident that the retrieve-and-edit model structure employed by EDITAS is effective for the assertion generation task. However, the semantic understanding and editing abilities of EDITAS could be improved. To address this issue, we propose $EditAS^2$. Figure 5 shows the overall framework of $EditAS^2$, which is divided into two phases, the offline fine-tuning phase and the online generation phase. The offline fine-tuning phase consists of a retrieval component and an editing component. In the retrieval component, $EditAS^2$ uses the large-scale training dataset as the retrieval corpus to extract the most similar focal-test and corresponding assertion to the input focal-test. In the editing component, $EditAS^2$ first calculates the edit sequence between the input focal-test and the retrieved focal-test. Then the edit sequence along with the input focal-test and retrieved Test-Assert Pair (TAP) is converted into an embedding and fed into the pre-trained model. Finally, the pre-trained model is fine-tuned such that it can adjust the retrieved assertions based on the edit sequence and focal-test pairs. In the online generation phase, given an input focal-test, $EditAS^2$ first retrieves the similar TAP from the Test-Assert Retrieval Base, and then uses the trained editing model to adjust the retrieved assertion to generate the corresponding assertion. We will go over the details of each component below.

4.1 Retrieval component

In our approach, the Retrieve component retrieves a similar TAP from a corpus given the input focal-test. Specifically, to facilitate efficient retrieval, $EditAS^2$ first tokenizes each



```
// Input test prefix:
graph_union_1q ( ) { List < Binding > results = exec ( (
( "(graph<sp><" + ( Quad . unionGraph . getURI ( ) ) ) +
"><sp>(bgp<sp>(<s2><sp>?p<sp>?o)))"  )  ,  QUADS  )  ;
"<AssertPlaceHolder>" ; }
// Input focal method:
size ( ) { return rows . size ( ) ; }
```

```
// Similar test prefix:
graph_union_2q ( ) { List < Binding > results = exec ( (
( "(graph<sp><" + ( Quad . unionGraph . getURI ( ) ) ) +
"><sp>(bgp<sp>(<s1><sp>?p<sp>?o)<sp>(?o<sp>?q<sp>?z)<sp>
))" ) , QUADS ) ; "<AssertPlaceHolder>" ; }
// Similar focal method:
size ( ) { return rows . size ( ) ; }
```

```
// Similar Assertion:
Assert . assertEquals ( 2 , results . size ( ) )
```

```
// new Assertion:
Assert . assertEquals ( 4 ,
results . size ( ) )
```

Fig. 5. The overall framework of our approach.

focal-test in the training and test sets using javalang [39] and removes duplicate tokens. Then, the Retrieve component retrieves the TAP whose focal-test has the highest Jaccard similarity coefficient with the input focal-test. The Jaccard similarity is a text similarity measurement that considers the overlap of words between two texts, calculated using the following formula:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

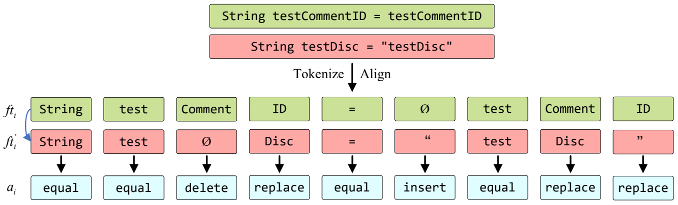
Where A and B are two bags of words, $|\cdot|$ denotes the number of elements in a collection.

4.2 Editing component

The editing component aims to learn different assertion patterns from similar TAPs and modify one assertion to another based on semantic differences between focal-tests. Specifically, for a given focal-test ft and its similar focal-test instance ft', along with their corresponding assertions x and y, the neural edit model aims to find a function f such that $f(ft, ft', x) = y$. In this section, we describe the focal-test semantic difference representation, the input construction and the retrieval-augmented fine-tuning for editing model training in detail.

4.2.1 *Focal-test semantic difference representation*

We extract and compare semantic information and modification details between focal-tests using edit sequences, according to the previous work's finding [40]: different words between the two methods can reflect their semantic differences. We follow a similar approach as in [41], [42], aligning the two tokenized focal-test sequences using a diff tool and creating an edit sequence based on the resulting alignment. As shown in Figure 6, each element (named as an *edit*) in an edit sequence is represented as a triple $\langle ft_i, ft_i', a_i \rangle$, where ft_i is a token in one focal-test and ft_i' is a token in the similar focal-test, and a_i is the edit action that transforms ft_i to ft_i'. There are four types of edit actions: *insert*, *delete*, *equal*, or *replace*. when a_i is an *insert* (*delete*) operation, it means that ft_i (ft_i') will be an empty token $\emptyset$. Constructing such an edit sequence can not only preserve the information of the focal-test (i.e., ft_i and ft_i') but also highlight their fine-grained differences through a_i. It is worth noting that in order to refine the length of the edit sequence, we remove edits where a_i is equal.



```
String testCommentID = testCommentID
String testDisc = "testDisc"
```

Tokenize
Align
fti
fti'
ai
String test Comment ID = Ø test Comment ID
String test Ø Disc = " test Disc "
equal equal delete replace equal insert equal replace replace

Fig. 6. Converting a difference between focal-tests to an edit sequence.

4.2.2 *Input Construction*

The model's input comprises an input focal-test ft, a similar focal-test ft', an edit sequence e and a similar assertion x. Among them, the input focal-test ft, the similar focal-test ft' and the similar assertion x help the model to understand their own semantic information, and the edit sequence e helps the model to understand the semantic differences between the focal-tests. As shown in Figure 5, We first tokenize the input focal-test, the similar focal-test and the similar assertion into token sequences, i.e., $ft = (ft_1, ft_2, ..., ft_m)$, $ft' = (ft_1', ft_2'..., ft_n')$, and $x = (x_1, x_2, ..., x_l)$. Here, m, n, and l correspond to the lengths of the input focal-test, the similar focal-test, and the similar assertion, respectively. We utilize the Roberta [43] tokenizer to acquire tokens for these inputs. In contrast to token generation through whitespace-based separation, Byte-Pair Encoding (BPE) [44] is utilized to tackle Out-of-Vocabulary (OoV) issues. Next, we generate edit sequences denoted as e for focal-tests. Each edit in the edit sequence e is represented as a triple $\langle ft_i, ft_i', a_i \rangle$. Finally, We concatenate the inputs of ft, ft', e, and x into one input I, which is then employed to generate the target assertion. In this study, we opt for concatenating multiple inputs into a single unified input rather than employing multiple encoders to process each input individually. The self-attention mechanism of CodeT5 is more advantageous with a single input compared to multiple encoders, allowing it to capture relationships between different inputs.

4.2.3 Retrieval-Augmented Fine-Tuning for Editing Model Training

At this stage, we propose an editing model to adjust the retrieved assertions based on the focal-test pairs, the similar assertion, and the edit sequence. The editing model is built using a pre-trained CodeT5 model, fine-tuned on the assertion generation datasets. CodeT5 [12] adopts T5's encoder-decoder architecture and incorporates token-type information specific to code. It utilizes a denoising sequence-to-sequence pre-training task, and specifically captures the semantics conveyed by developer-assigned identifiers through two tasks: identifier tagging and masked identifier prediction. These tasks enable CodeT5 to comprehend the significance of identifiers in code and to learn contextual dependencies involving identifiers.

The decision to employ CodeT5 for this task is not only due to its architectural suitability but also for its outstanding performance in code-related tasks, particularly in assertion generation. As demonstrated by the experimental results in Table 5, CodeT5 consistently surpasses other pre-trained models, such as CodeBERT, GraphCodeBERT, CodeGPT, and UnixCoder, in both accuracy and CodeBLEU scores across the $Data_{atlas}$ and $Data_{integration}$. Even when compared to large language models such as StarCoder, CodeT5 delivers comparable or even superior performance while requiring significantly fewer computational resources and less training time. This combination of high performance and efficiency makes CodeT5 the optimal choice for assertion generation tasks.

The assertion generation datasets are structured as $\{((ft, ft', e, x), y)_1, \ldots, ((ft, ft', e, x), y)_n\}$, where ft signifies the input focal-test, ft' signifies its similar focal-test, e denotes the edit sequence between focal-tests, x denotes the retrieved assertion, and y denotes the expected assertion. More details about assertion datasets can be found in Section 5.2. Overall, the editing model comprises an encoder and a decoder, powered by CodeT5. The encoder accepts ft, ft', e, and x as input, truncates the code tokens, and generates embeddings for the input tokens. The decoder captures the correlation between the input and output sequences to generate new assertions.

❶ **Encoder**. The task of the encoder is to derive the contextually representative embedding of the input sequence I. This sequence comprises four distinct parts, denoted as $I = \{ft, ft', e, x\}$. The various parts of the input play specific roles in generating different aspects of the target assertion. For instance, e aids in comprehending the semantic differences between focal-tests, x acts as an initial template for the target assertion, and both ft and ft' contribute to understanding their individual semantics. Therefore, it is essential to grasp the correlation between every token in the input I and each token in the target assertion y during the fine-tuning process. To effectively establish this correlation, we utilize the pre-trained CodeT5 encoder in this study. CodeT5 has shown significant promise in understanding and generating code, thanks to its remarkable capability to grasp code semantics using identifiers specified by developers. Derived from the Transformer architecture [33], CodeT5's encoder transforms the input, denoted as $I = \{ft, ft', e, x\}$, into the contextual vector representation R.

For each input token I_i, the encoder initially produces three embedding vectors: a query vector q_i, a key vector k_i, and a value vector v_i. Subsequently, the encoder utilizes the query vector q_i and the key vector k_j for each token to calculate the attention score for I_i, as follows:

$$\alpha_{i,j} = \frac{q_i \cdot k_j}{\sqrt{d}} \tag{2}$$

Here, d represents the dimension of q_i and k_j. The attention score $a_{i,j}$ indicates the degree of attention assigned to the j_{th} input while encoding the i_{th} input. To emphasize the relationship between the input tokens, the encoder obtains the normalized scores through the softmax function:

$$\hat{\alpha}_{i,j} = softmax\{\alpha_{i,j}\} = \frac{exp(\alpha_{i,j})}{\sum_t exp(\alpha_{i,t})} \tag{3}$$

To learn the relevant and irrelevant tokens, we used the softmax function $\alpha\hat{_{i,j}}$ to multiply each value vector v_j and then sum the resultant vectors.

$$z_i = \sum_j \hat{\alpha}_{i,j} v_j \tag{4}$$

To grasp the correlation between each input token I_i and the target assertion y, we utilize the attention scores $z = \{z_1, ..., z_{|I|}\}$ as the contextual vector representation R and input for the decoder.

❷ **Decoder**. Decoder aims to generate new assertions using the provided input. The decoder is trained to produce the respective new assertion y incrementally, one token at a time, relying on both the input and all previously generated tokens. Mathematically, the assertion generation task can be articulated as finding $\bar{y}$. Here, $\bar{y}$ is characterized by the expression: $\bar{y} = argmax_y P_\theta(t|I)$ where $P_\theta(t|I)$ is:

$$P_\theta(t|I) = \prod_{i=1}^{w} P_\theta(y_i|y_1, ..., y_{i-1}; ft, ft', e, x) \tag{5}$$

Given the input I, $P_\theta(t|I)$ can be viewed as the conditional log-likelihood of the predicted assertion y. The model can be trained by minimizing the negative log-likelihood concerning both the predicted assertion and ground truth.

Specifically, we employ the pre-trained CodeT5 decoder in our approach. The decoder architecture comprises two main components: the self-attention layer and the encoder-decoder attention layer. The self-attention layer conducts computations similar to the encoder. It takes input Q, K, and V from the output of the previous layer in the decoder but focuses solely on the tokens generated at the current position. On the other hand, the encoder-decoder attention Layer, designed to capture the correlation between the output and input sequence, calculates the attention score between the encoder and the decoder. In this process, the key vector K originates from the encoder output, while the query vector Q is derived from the output of the self-attention layer. Consequently, the attention distribution α^j between the input tokens $w_1, ..., w_m$ and the target token y_j can be expressed as:

$$\alpha^j = softmax(\frac{Q_j K^T}{\sqrt{d_k}}) \tag{6}$$

4.3 Assertion Generation Pipeline in EditAS^2

Given an input focal-test ft, $EditAS^2$ generates an assertion through three steps:

Step 1: Selecting a similar TAP. $EditAS^2$ utilizes a large-scale training dataset as the retrieval corpus. The Retrieval component then finds the TAP in the corpus that best matches ft. Additional details on the retrieval process are described in Section 4.1.

Step 2: Capturing fine-grained semantic differences between focal-tests. Through step 1, $EditAS^2$ obtains the focal-test similar to ft along with the corresponding assertion. Then $EditAS^2$ computes the edit sequence between ft and the similar focal-test to capture the semantic differences between them. Further details are described in Section 4.2.1.

Step 3: Enhancing semantic understanding of focal-tests. Next, $EditAS^2$ co-encodes focal-tests and edit sequences as input to the model, thereby enhancing the model's comprehension of the semantics inherent in the focal-tests. Further details are described in Section 4.2.2.

Step 4: Adjusting retrieved assertions to the focal-tests. Finally, the trained editing model learns the semantics of the focal-tests and adapts the retrieved assertion to the editing sequence, thus generating an assertion corresponding to ft. For more details on the model, see Section 4.2.3.

5 Experimental setup

In this section, we introduce the baselines and dataset, followed by a description of the evaluation metrics employed in the experiment. Then, we will delve into the implementation details. Our experiments are designed to address the following research questions:

- **RQ3:** How effective is the proposed approach in terms of inference performance?
- **RQ4:** How does the proposed approach perform in real-world scenarios?
- **RQ5:** How effective are the different variants?

5.1 Baselines

To answer the above-mentioned research questions, we first compare our approach with the following six state-of-the-art assertion generation approaches [8], [14], [15], including sequence-to-sequence-based, retrieval-based, and retrieval-augmented learning-based approaches.

- **ATLAS** [8] is a classical assertion generation approach that utilizes a sequence-to-sequence model for generating assertions from scratch.
- $\mathbf{IR_{ar}}$ [14] is a retrieval-based assertion generation approach, using the Jaccard similarity coefficient to retrieve the most similar assertion to the given focal-test.
- $\mathbf{RA^{H}_{adapt}}$ [14] is a retrieval-base assertion generation approach, where the retrieval assertion is adapted using heuristics.
- $\mathbf{RA^{NN}_{adapt}}$ [14] is another retrieval-based assertion generation approach, but in this case, the retrieval assertion is adapted using neural networks.
- ***integration*** [14] is an approach that combines retrieval and deep learning techniques for assertion generation. This approach aims to leverage the strengths of both approaches in a unified manner.
- **EditAS** [15] is a state-of-the-art retrieve-and-edit based assertion generation method. It utilizes assertions from similar focal-tests as prototypes and uses a neural sequence-to-sequence model to learn the assertion edit patterns.

While we refer to the experimental design of the most recent study EditAs [15] for comparison with the six state-of-the-art assertion generation methods, we also note that some studies (e.g., [9]) use pre-trained models (e.g., CodeGPT [11] and CodeT5 [12]) to generate assertions. Therefore, we compare our approach with five existing pre-trained code models for further comparative analysis: CodeBERT [35], GraphCodeBERT [36], CodeGPT [11], UnixCoder [34] and CodeT5 [12]. The selection of these models is based on specific criteria: **(1) Open-source:** We select open-source pre-trained models and exclude closed-source ones due to the unavailability of their parameters, making fine-tuning techniques impractical or expensive. For instance, Codex [13] is excluded as its models haven't been publicly released. **(2) Code domain:** We specifically choose pre-trained models designed for the code domain. Given our task of generating test assertions, the selected pre-training model should be trained on a sufficiently large code corpus. Models like T5 [45] and GPT-2 [46], primarily designed for natural language processing tasks, are thus excluded. **(3) Different architectures:** We choose models with different architectures (i.e., encoder-decoder, encoder-only, and decoder-only) and organizations (e.g., Microsoft and Salesforce) for different evaluation scenarios. These models we utilized are trained on a large corpus of code, cover different architectures, represent the state-of-the-art, and are publicly available on the Hugging Face website. To fine-tune these pre-trained models for assertion generation, we use focal-tests as input and corresponding assertions as output. Each model is fine-tuned separately on $Data_{atlas}$ and $Data_{integration}$, enabling them to adapt to the specific characteristics of each dataset and effectively adjust to the task of assertion generation.

In addition, Nashid et al. [10] employ prompt engineering with few-shot learning, querying the black-box LLM Codex [13] to generate assertions. Since Codex is not open-source, we select StarCoder-1B [47] as the foundation model for generating assertions to ensure a proper comparison with LLM-based approaches. To the best of our knowledge, this is the first attempt to fine-tune StarCoder for assertion generation research. Specifically, we fine-tune StarCoder separately on $Data_{atlas}$ and $Data_{integration}$ by using focal-test cases as input and their corresponding assertions as output, with the instruction "Please fill the <AssertPlaceHolder> section with the appropriate assertions" to adapt it to this task. The fine-tuning process relies on a structured prompt system, where the focal-test cases and assertion placeholders guide the model in generating accurate assertions.

- **CodeBERT** [35] is a bi-modal encoder model, incorporating *masked language modeling* [48] and *replaced token detection* [49] in pre-training for semantic links between natural and programming languages.
- **GraphCodeBERT** [36] is an encoder model that incorporates code structure. Beyond traditional *Masked Language Modeling*, it involves *edge prediction* and *node alignment* as structure-aware pre-training tasks.
- **CodeGPT** [11] is a decoder model using multiple transformer decoders. CodeGPT is an *auto-regressive language model*, predicting the next token in a sequence based on the previous token and itself.
- **UnixCoder** [34] is an encoder-decoder model with two novel pre-training tasks: *multi-modal contrastive learning* and *cross-modal generation* to further refine its semantic embedding.
- **CodeT5** [12] is an encoder-decoder model that uses the architecture of T5. It uses code data from CodeSearchNet and C and CSharp code data from BigQuery [50] for pre-training. In addition to the common denoising pre-training tasks, CodeT5 specifically captures the semantics conveyed by developer-assigned identifiers through two tasks: *identifier tagging* and *masked identifier prediction*. Following previous research [28], [30], [51], we use the base model with 223M parameters.
- **StarCoder** [47] is a state-of-the-art code LLM with 1B parameters, built on the GPT-2 architecture with *multi-query attention* and *Fill-in-the-Middle* objective.

5.2 Dataset

Following previous studies [8], [14], [15], we using $Data_{atlas}$ and $Data_{integration}$ to evaluate $EditAS^2$ and baselines. Further details are described in Section 3.1.

In addition, to explore the inference performance of our model in cross-project scenarios, we construct a new cross-project test set. This test set is built upon the widely used Methods2test [52] dataset. The Methods2test [52] dataset is a set of mapped test cases with project information, where each test case is mapped to the corresponding focal method. Importantly, we ensure that the projects in the cross-project test set do not overlap with those in the training set, and all duplicates are excluded. The construction of this cross-project test set follows the methodology outlined in the ATLAS [8] dataset, which includes the following steps: (1) **Project Filtering**: Removing projects already included in the $Data_{atlas}$ and $Data_{integration}$. (2) **Extracting Test Cases and Focal Methods**: Extracting each project's focal methods and corresponding test cases from the "body" field in the "focal_method" and "test_case" fields, respectively. (3) **Filtering Test Cases**: Retaining only test methods containing a single assertion statement, excluding those with multiple assertions. (4) **Constructing Test-Assert Pair (TAP)**: Linking focal methods and test cases to form Test-Assert Pairs by appending the focal method's signature and body to the end of the test method. Additionally, extracting the entire assertion statement from the test case and replacing it with "AssertPlaceHolder". Finally, we randomly select samples from the filtered dataset to match the size of the $Data_{atlas}$ test set, resulting in a cross-project test set with 15,676 samples.

5.3 Bug Benchmark

We use the Defects4J benchmark to evaluate the bug-finding performance of assertion generation approaches in the real world, which includes 17 real Java projects and 835 bugs. Each bug in Defects4J is logged in the issue tracker and fixed in a single commit by modifying the source code. Therefore, each buggy program not only includes a buggy program version but also a bug-fixed program version. In particular, each buggy program fails at least one test case, while the bug-fixed program version successfully passes all test cases. Consequently, if a test case fails on the buggy program version and passes on the bug-fixed version, a specific bug must be triggered. Following the previous studies [3], [53], we also use Defects4J 2.0.0 in this work.

5.4 Evaluation Metrics

In RQ3, we use Accuracy and CodeBLEU [18] as metrics to evaluate the model's inference performance, following the previous studies [8], [14], [15], [18]. In RQ4.1, we utilize BugFound [53] and Unique BugFound metric to evaluate the effectiveness of the generated assertions in finding real-world bugs. In RQ4.2, we utilize Compliation rate and Precision [53] metrics to evaluate the quality of the generated assertions in real scenarios. In RQ5, we focus on evaluating the inference accuracy of different variants of the proposed method, utilizing both Accuracy and CodeBLEU metrics. The following provides a concise description of each metric.

①Accuracy. A generated assertion is considered accurate only when it exactly matches the ground truth. Accuracy refers to the percentage of generated outputs that syntactically match the expected outputs in the samples.

②CodeBLEU. CodeBLEU, an evaluation metric for code synthesis tasks, goes beyond the limitations of BLEU. Recent studies [18] expose BLEU's shortcomings in accounting for grammatical and logical correctness, creating a bias toward candidates with elevated n-gram accuracy despite significant logical errors. In response, researchers propose CodeBLEU as a more effective metric for assessing code generation tasks. Therefore, we use CodeBLEU to measure the similarity between the generated assertion and the ground truth, with detailed information in [18].

③ BugFound. BugFound denotes the total number of bugs that can be found by the generated assertions, including repeated detections of the same bug. Since EvoSuite is random, we run it ten times and count the average number of bugs detected across these ten runs as BugFound.

④ Unique BugFound. Unique BugFound counts the number of distinct bugs detected, ensuring each bug is counted only once, regardless of how many times it is detected. Since EvoSuite is random, we run it ten times and count the distinct bugs detected across these ten runs as Unique BugFound.

⑤ Compilation rate. It measures whether the generated assertions can be successfully compiled into executable test code. A high compilation rate is essential, as assertions that fail to compile cannot be executed, rendering them ineffective in practice.

⑥ Precision. It refers to the ratio of the generated assertions that can indeed find bugs to all the generated assertions that fail to execute [53]. We choose precision instead of simple pass/fail rates because the test cases run on buggy versions of the programs. A passing test case indicates successful execution but no bug detection, while a failing test case may result from either an execution error or the successful identification of a bug. Hence, only failing test cases that find bugs are considered indeed valid.

5.5 Implementation Details

We leverage the Hugging Face implementation [54] for all adopted pre-trained models, with hyperparameters derived from default values. Following earlier research guidance [55], [56], our approach employs the CodeT5-base model, which consists of 12 layers of the Transformer, as its utility and effectiveness are often comparable to its larger counterparts. The Adam optimizer [57] is utilized with a learning rate of 1e-5, and the batch size is set to 8. The length of focal-tests and assertions is set to 256, while the edit sequence length is set to 150. The training process lasts 15 epochs, with our model trained to minimize cross-entropy loss. All experiments are conducted using the PyTorch framework [58] and on an Ubuntu 20.04 server equipped with four NVIDIA GeForce RTX 3090 GPUs.

6 Experimental Results

6.1 RQ3: How effective is the proposed approach in terms of inference performance?

6.1.1 Methodology

To provide a comprehensive evaluation of performance in assertion inference, we select six state-of-the-art assertion generation approaches, five pre-trained code models, and the large language model StarCoder to compare with our approach. To ensure a fair comparison, we evaluate all approaches on the same datasets (i.e. $Data_{atlas}$ and $Data_{integration}$). Each approach takes the focal-test as input and generates the corresponding assertion as output. For StarCoder, we fine-tune it separately on both datasets and provide a prompt: "Please fill the <AssertPlaceHolder> section with the appropriate assertions" to guide the generation. Then, we calculate the accuracy and CodeBLEU scores between the assertions generated by different approaches and human-written assertions. Finally, we report the overall results, results for different assertion types, and performance during the training phase.

Additionally, to further evaluate the effectiveness of our approach in cross-project scenarios, we train the models on the training sets of $Data_{atlas}$ and $Data_{integration}$, and test them on the newly constructed cross-project test set (as described in Section 5.2). Since EditAS currently outperforms ATLAS and other retrieval-based methods in assertion generation, we focus our comparison on EditAS, pre-trained models, and StarCoder.

6.1.2 Results

Overall Effectiveness. The experimental results are presented in Table 5. Clearly, ATLAS performs the worst of all the approaches, and this can be primarily attributed to two reasons. Firstly, as a typical sequence-to-sequence DL model, ATLAS encounters challenges such as exposure bias and gradient disappearance, resulting in reduced effectiveness when generating lengthy sequences of tokens for an assertion. Previous research [14] demonstrated that ATLAS achieves an accuracy of 46.3% in generating less than 15 tokens and only 17.9% accuracy in generating tokens exceeding 15. Secondly, ATLAS exhibits a weaker capability to generate statements containing unknown tokens, significantly undermining its overall performance. IR_{ar} is more effective than ATLAS by extracting assertions from the corpus and using them directly as output. This suggests that assertions in the similar focal-tests contain some valuable and reusable information, which is why we use the similar focal-test's assertions as prototypes. Both RA_{adapt}^{H} and RA_{adapt}^{NN} adapt the retrieved assertions based on IR_{ar}. However, their adaptive operational performance is limited, particularly when dealing with the complex dataset, $Data_{integration}$. For instance, RA_{adapt}^{H} improves accuracy by 12.99% compared to IR_{ar}, while the improvement in $Data_{integration}$ is only 4.62%. A similar observation applies to RA_{adapt}^{H}. *Integration* combining RA_{adapt}^{NN} with ATLAS yields higher accuracy and CodeBLEU scores. As can be seen from Table 5, EditAS outperforms ATLAS and IR-based methods. This is because, compared to ATLAS, EditAS utilizes rich semantic information from retrieved assertions instead of generating assertions from scratch. In comparison to the IR-based baselines, EditAS takes into account the semantic differences between the input focal-test and similar focal-test. This demonstrates the effectiveness of using retrieved assertions as prototypes and modifying them by considering the semantic differences between the input and similar focal-tests in assertion generation. Pre-trained models like CodeBERT, GraphCodeBERT, CodeGPT, and UnixCoder demonstrate a similar performance in generating assertions compared to EditAS. This emphasizes that the retrieve-and-edit model structure employed by EditAS is very effective for assertion generation task. On the contrary, CodeT5 achieves some performance improvement, with accuracy increasing by 10.34% in $Data_{atlas}$ and 0.86% in $Data_{integration}$ compared to EditAS. This demonstrates the power of CodeT5 in terms of semantic understanding and code generation, thus supporting our decision to utilize CodeT5 to gain a deeper understanding of the semantic differences in focal-tests and to edit a template to generate the target assertion.

As can be seen from Table 5, $EditAS^2$ outperforms all state-of-the-art assertion generation methods and pre-trained models. Specifically, compared to the DL-based baseline ATLAS and the pre-trained models CodeBERT, GraphCodeBERT, CodeGPT, UnixCoder and CodeT5, the

TABLE 5
Comparisons of our approach with each baseline in terms of inference performance

Approach	$Data_{atlas}$		$Data_{integration}$	
	Accuracy	**CodeBLEU**	**Accuracy**	**CodeBLEU**
ATLAS	31.42 (↑ 104.87%)	60.88 (↑ 34.51%)	21.66 (↑ 153.51%)	35.38 (↑ 103.25%)
IR_{ar}	36.26 (↑ 77.52%)	72.14 (↑ 13.52%)	37.90 (↑ 44.88%)	63.40 (↑ 13.42%)
RA^{H}_{adapt}	40.97 (↑ 57.12%)	71.53 (↑ 14.48%)	39.65 (↑ 38.49%)	62.04 (↑ 15.91%)
RA^{NN}_{adapt}	43.63 (↑ 47.54%)	72.58 (↑ 12.83%)	40.53 (↑ 35.48%)	63.35 (↑ 13.51%)
Integration	46.54 (↑ 38.31%)	72.31 (↑ 13.25%)	42.20 (↑ 30.12%)	62.13 (↑ 15.74%)
EDITAS	53.46 (↑ 20.41%)	79.78 (↑ 2.64%)	44.36 (↑ 23.78%)	65.52 (↑ 9.75%)
CodeBERT	51.83 (↑ 24.19%)	71.36 (↑ 14.76%)	41.59 (↑ 32.03%)	53.69 (↑ 33.94%)
GraphCodeBERT	55.06 (↑ 16.91%)	74.67 (↑ 9.67%)	40.78 (↑ 34.65%)	55.18 (↑ 30.32%)
CodeGPT	52.86 (↑ 21.77%)	72.16 (↑ 13.48%)	38.70 (↑ 41.89%)	51.05 (↑ 40.86%)
UnixCoder	54.03 (↑ 19.14%)	73.31 (↑ 11.70%)	40.86 (↑ 34.39%)	54.62 (↑ 31.66%)
CodeT5	58.99 (↑ 9.12%)	78.24 (↑ 4.67%)	44.74 (↑ 22.73%)	61.28 (↑ 17.35 %)
StarCoder	**64.84**(↓ 0.72%)	80.70(↑ 1.47%)	50.79(↑ 8.11%)	65.51(↑ 9.77%)
$EditAS^2$	64.37	**81.89**	**54.91**	**71.91**

↑ denotes performance improvement of $EditAS^2$ against state-of-the-art baselines

TABLE 6
Detailed statistics of our approach and each baseline for each assert type

Dataset	**Approach**	**Total**	**AssertType**								
			Equals	**True**	**That**	**NotNull**	**False**	**Null**	**ArrayEquals**	**Same**	**Other**
$Data_{atlas}$	ATLAS	4,925 (31%)	2,501 (32%)	966 (35%)	248 (17%)	598 (51%)	229 (23%)	236 (30%)	100 (33%)	47 (15%)	0 (0%)
	IR_{ar}	5,684 (36%)	2,957 (38%)	1,039 (37%)	449 (31%)	439 (38%)	314 (31%)	285 (36%)	111 (36%)	89 (29%)	**1 (50%)**
	RA^{H}_{adapt}	6,423 (41%)	3,300 (42%)	1,151 (41%)	536 (37%)	553 (48%)	335 (33%)	316 (40%)	120 (39%)	111 (36%)	**1 (50%)**
	RA^{NN}_{adapt}	6,839 (44%)	3,509 (45%)	1,225 (44%)	551 (38%)	610 (52%)	342 (34%)	341 (43%)	134 (44%)	126 (41%)	**1 (50%)**
	Integration	7,295 (47%)	3,714 (47%)	1,333 (48%)	546 (38%)	724 (62%)	348 (35%)	352 (44%)	148 (48%)	129 (41%)	**1 (50%)**
	EDITAS	8,380 (53%)	4,131 (53%)	1,581 (57%)	526 (36%)	807 (69%)	577 (57%)	469 (59%)	167 (54%)	122 (39%)	0 (0%)
	CodeBERT	8,125 (52%)	4,000 (51%)	1,442 (52%)	573 (40%)	715 (62%)	561 (56%)	528 (66%)	148 (48%)	158 (51%)	0 (0%)
	GraphCodeBERT	8,631 (55%)	4,309 (55%)	1,474 (53%)	612 (42%)	800 (69%)	600 (60%)	526 (66%)	155 (50%)	155 (50%)	0 (0%)
	CodeGPT	8,287 (53%)	4,179 (53%)	1,246 (45%)	547 (38%)	834 (72%)	656 (65%)	506 (63%)	176 (57%)	143 (46%)	0 (0%)
	UnixCoder	8,469 (54%)	4,147 (53%)	1,508 (54%)	624 (43%)	848 (73%)	544 (54%)	485 (61%)	159 (52%)	154 (50%)	0 (0%)
	CodeT5	9,247 (59%)	4,722 (60%)	1,510 (54%)	677 (47%)	805 (69%)	629 (63%)	560 (70%)	189 (62%)	155 (50%)	0 (0%)
	StarCoder	**10,165 (65%)**	**5,088 (65%)**	1,754 (63%)	802 (56%)	**889 (77%)**	667 (66%)	**562 (70%)**	**212 (69%)**	**190 (61%)**	**1 (50%)**
	$EditAS^2$	10,090 (64%)	4,984 (63%)	**1,818 (65%)**	**808 (56%)**	844 (73%)	**692 (69%)**	**562 (70%)**	191 (62%)	**190 (61%)**	**1 (50%)**
$Data_{integration}$	ATLAS	5,749 (22%)	2,900 (23%)	619 (17%)	537 (15%)	388 (30%)	126 (12%)	85 (12%)	47 (13%)	37 (12%)	1,010 (33%)
	IR_{ar}	10,059 (38%)	4,664 (37%)	1,436 (39%)	1,070 (30%)	600 (47%)	394 (37%)	286 (39%)	147 (41%)	113 (35%)	1,349 (45%)
	RA^{H}_{adapt}	10,525 (40%)	4,882 (39%)	1,487 (41%)	1,142 (32%)	651 (51%)	403 (38%)	297 (40%)	154 (43%)	121 (38%)	1,388 (46%)
	RA^{NN}_{adapt}	10,758 (41%)	4,988 (40%)	1,526 (42%)	1,161 (33%)	691 (54%)	401 (37%)	308 (42%)	162 (45%)	126 (39%)	1,395 (46%)
	Integration	11,201 (42%)	5,248 (42%)	1,566 (43%)	1,196 (34%)	711 (55%)	401 (37%)	313 (43%)	162 (45%)	128 (40%)	1,476 (49%)
	EDITAS	11,773 (44%)	5,339 (42%)	1,702 (47%)	1,304 (37%)	800 (62%)	523 (49%)	376 (51%)	172 (47%)	139 (44%)	1,418 (47%)
	CodeBERT	11,038 (42%)	4,958 (39%)	1,532 (42%)	1252 (35%)	734 (57%)	516 (48%)	375 (51%)	155 (43%)	142 (45%)	1374 (45%)
	GraphCodeBERT	10,825 (41%)	4,899 (39%)	1,469 (40%)	1228 (35%)	747 (58%)	496 (46%)	391 (53%)	138 (38%)	146 (46%)	1311 (43%)
	CodeGPT	10,272 (39%)	4,716 (38%)	1,346 (37%)	1183 (33%)	653 (51%)	466 (44%)	348 (47%)	130 (36%)	125 (39%)	1305 (43%)
	UnixCoder	10,845 (41%)	4,914 (39%)	1,501 (41%)	1209 (34%)	744 (58%)	515 (48%)	345 (47%)	146 (40%)	146 (46%)	1325 (44%)
	CodeT5	11,875 (45%)	5,545 (44%)	1,541 (42%)	1420 (40%)	742 (58%)	532 (50%)	353 (48%)	165 (46%)	147 (46%)	1430 (47%)
	StarCoder	13,483 (51%)	6,261 (50%)	1,810 (50%)	1607 (45%)	809 (63%)	607 (57%)	419 (57%)	188 (52%)	160 (50%)	1622 (54%)
	$EditAS^2$	**14,575 (55%)**	**6,778 (54%)**	**1,969 (54%)**	**1741 (49%)**	**851 (66%)**	**663 (62%)**	**452 (61%)**	**197 (54%)**	**179 (56%)**	**1745 (58%)**

average accuracy of $EditAS^2$ is improved by 129.19%, 28.11%, 25.78%, 31.83%, 26.77% and 15.93%. This is attributed to the fact that these methods only use information from the input focal-test and generate assertions from scratch, not utilizing the semantic information of the retrieved assertions and the semantic differences between the retrieved focal-test and the input focal-test. Compared to the IR-based baselines IR_{ar}, RA^{H}_{adapt}, RA^{NN}_{adapt}, and *Integration*, $EditAS^2$ improves the average accuracy by 61.20%, 47.81%, 41.51% and 34.22%, which proves the effectiveness of our editing component. Compared with EDITAS, $EditAS^2$ has an average accuracy improvement of 22.10% and an average CodeBLEU improvement of 6.20%. This is because $EditAS^2$ utilizes the pre-trained model CodeT5 to enhance its understanding of semantic differences between input and similar focal-tests as well as its ability to edit retrieved assertion. In addition to this, we provide CodeT5 not only with edit sequence between focal-tests, but also with input and similar focal-tests themselves. This dual input ensures a more comprehensive and detailed understanding of the semantic differences between focal-tests. In comparison to large language models such as StarCoder, our approach demonstrates competitive performance on $Data_{atlas}$ and significantly outperforms StarCoder on $Data_{integration}$, achieving improvements in accuracy by 8.11% and CodeBLEU scores by 9.77%, respectively. Notably, this performance is achieved despite StarCoder leveraging a substantially larger pre-training dataset and more parameters. This result demonstrates that even with a significant reduction in parameter size, our approach can still achieve better results than large language models on complex dataset, demonstrating the advantages of the retrieve-and-edit framework of our method.

> ***Summary-3.1***: $EditAS^2$ outperforms state-of-the-art assertion generation methods and pre-trained models in terms of accuracy and CodeBLEU, with average performance improvements of 15.93%-129.19% and 11.01%-68.88% on the two datasets, respectively. When compared to large language models like StarCoder, $EditAS^2$ demonstrates comparable performance on $Data_{atlas}$, while significantly outperforming it on $Data_{integration}$.

TABLE 7
Comparison of our approach and baselines in terms of inference performance on cross-projects

Approach	$Data_{atlas}$		$Data_{integration}$	
	Accuracy	**CodeBLEU**	**Accuracy**	**CodeBLEU**
CodeBert	12.55 (↑ 52.99%)	37.3 (↑ 13.11%)	20.55 (↑ 26.67%)	34.81 (↑ 15.83%)
GraphCodeBert	12.99 (↑ 47.81%)	37.06 (↑ 13.84%)	20.93 (↑ 24.37%)	33.69 (↑ 19.68%)
CodeGPT	13.49 (↑ 42.33%)	37.49 (↑ 12.54%)	18.86 (↑ 38.02%)	33.13 (↑ 21.70%)
UniXcoder	12.98 (↑ 47.92%)	37.82 (↑ 11.55%)	19.69 (↑ 32.20%)	32.78 (↑ 23.00%)
CodeT5	14.03 (↑ 36.85%)	39.8 (↑ 6.01%)	21.27 (↑ 22.38%)	37.36 (↑ 7.92%)
StarCoder	14.14 (↑ 35.79%)	39.07 (↑ 7.99%)	25.67(↑ 1.40%)	39.93 (↑ 0.98%)
EDITAS	16.69 (↑ 15.04%)	38.17 (↑ 10.53%)	25.81 (↑ 0.85%)	36.59 (↑ 10.19%)
$EditAS^2$	**19.20**	**42.19**	**26.03**	**40.32**

↑ denotes performance improvement of $EditAS^2$ against state-of-the-art baselines

Effectiveness on Different Assertion Types. We further evaluate the effectiveness of the $EditAS^2$ and baseline methods for different assertion types on two datasets. In Table 6, each row represents a method, each column represents an assertion type, and each cell represents the number of assertions correctly generated by a particular method for a certain assertion type and their respective ratios. The results show that on $Data_{atlas}$, $EditAS^2$ maintains the best or near-optimal performance across all assertion types, second only to StarCoder. Furthermore, $EditAS^2$ outperforms all baseline methods on $Data_{integration}$ in all assertion types. In particular, $EditAS^2$ also outperforms all baselines in *other* assertion type, indicating that it is proficient in handling non-standard JUnit assertions. This ability makes up for the weaknesses of EDITAS. In conclusion, the experimental results demonstrate the generality of $EditAS^2$ in generating different types of assertions.

> ***Summary-3.2***: $EditAS^2$ outperforms nearly all baselines across all assertion types on both datasets. In particular, compared to EDITAS, $EditAS^2$ has the ability to handle non-standard JUnit assertion types.

Performance During the Training Phase. $EditAS^2$ achieves an accuracy of 96.37% and a CodeBLEU score of 97.98% on the training set of $Data_{atlas}$, and an accuracy of 88.83% with a CodeBLEU score of 93.06% on the training set of $Data_{integration}$, demonstrating the model's ability to capture edit patterns in the training data. On the validation sets, the model attains an accuracy of 65.58% and a CodeBLEU score of 82.27% for $Data_{atlas}$, and 55.03% with a CodeBLEU score of 72.68% for $Data_{integration}$. The consistency of results between the validation and test sets suggests the model performs reliably on unseen data, reflecting similar feature distributions across both sets. Furthermore, the narrow gap in accuracy between the validation and test sets (65.58% vs. 64.37% on $Data_{atlas}$, and 55.30% vs. 54.91% on $Data_{integration}$) indicates that the model does not exhibit signs of overfitting. To further enhance its generalization capability, data augmentation techniques, such as random cropping, flipping, or perturbations, can be introduced during training to increase data diversity, enabling the model to learn a broader range of edit patterns and improve its performance on previously unseen data.

> ***Summary-3.3***: $EditAS^2$ effectively captures edit patterns during the fine-tuning process. Furthermore, the small gap between validation and test set performance highlights the model's robustness and generalization ability to unseen data.

Performance on Cross-Project Scenarios. The experimental results in Table 7 demonstrate that $EditAS^2$ consistently outperforms all baseline models. Compared to EDITAS, $EditAS^2$ has an average accuracy improvement of 7.95%, and an average CodeBLEU improvement of 10.36%. This demonstrates that even on cross-project datasets, the pre-trained CodeT5 model used by $EditAS^2$ can significantly enhance the performance of the editing module, further improving the overall performance. Compared to pre-trained models such as CodeBERT, GraphCodeBERT, CodeGPT, UnixCoder, and CodeT5, the average accuracy of $EditAS^2$ is improved by 39.79%, 36.09%, 40.18%, 40.06%, and 29.62%, respectively. Furthermore, compared to the large language model StarCoder, $EditAS^2$ demonstrates an 18.60% improvement in average accuracy and a 4.49% improvement in CodeBLEU scores. These results highlight a critical limitation of pre-trained models and even large language models like StarCoder: despite extensive pre-training on a large corpus and fine-tuning with a specific assertion task dataset, they still struggle to handle the significant semantic differences in cross-project data. In contrast, $EditAS^2$ leverages a retrieve-and-edit framework to address these challenges. It retrieves semantically similar Test-Assert Pairs (TAPs) from cross-project corpora and adapts the retrieved assertions by accounting for the semantic differences between the input and retrieved focal-tests. This approach proves to still be effective in cross-project scenarios with significant semantic differences. Overall, the results clearly demonstrate the robustness and generalization capabilities of $EditAS^2$ in cross-project scenarios. Despite significant semantic differences between projects, $EditAS^2$ consistently outperforms all baseline models, highlighting its effectiveness and practicality in real-world software environments.

It is noteworthy that the inference performance on the cross-project test set is significantly lower than the results obtained from the $Data_{atlas}$ and $Data_{integration}$ test sets. For example, $EditAS^2$, trained on $Data_{atlas}$, achieves an accuracy of 19.2% and a CodeBLEU score of 42.1% on the cross-project test set. In contrast, it achieves 64.3% accuracy and 81.8% CodeBLEU on the $Data_{atlas}$ test set. Two primary factors contribute to this observed performance gap. First, cross-project test sets often introduce novel patterns and features that are not present in the training data. This causes a discrepancy between the training and test data distributions, making it harder for the model to generalize to cross-project scenarios [59], [60]. Second, the similarity between the input focal-test and the retrieved focal-test is lower in cross-project settings. For example, in $Data_{atlas}$, 56.61% of samples in the original test set have a Jaccard similarity greater than

0.7, but only 22.15% of cross-project test samples meet this threshold. Similarly, in $Data_{integration}$, 69.46% of the samples in the original test set exhibit a similarity greater than 0.7, while only 23.99% of cross-project samples do. This reduced similarity further impacts performance. In future work, we aim to enhance the model's performance in cross-project settings by addressing these challenges.

> ***Summary-3.4***: In the cross-project scenarios, $EditAS^2$ outperformed all baseline models, achieving an average accuracy improvement of 7.95%-40.18% and an average CodeBLEU improvement of 4.49%-17.28% across both datasets. The results demonstrate the generalization capabilities of $EditAS^2$ in cross-project scenarios.

6.2 RQ4: How does the proposed approach perform in real-world scenarios?

6.2.1 RQ4.1: How effective is the proposed approach in finding real-world bugs?

6.2.1.1 Methodology

Generating assertions serves the purpose of easing the identification of potential issues within software units. To evaluate the effectiveness of the assertions generated by our approach in detecting software bugs, we conduct experiments on the Defects4J benchmark (as described in Section 5.3). This benchmark includes 17 real-world Java projects and a total of 835 bugs. Of these, 588 bugs can be triggered by assertions, and we focus on these. Each bug provides both buggy and bug-fixed versions of the programs for evaluation. The evaluation process consists of the following three main steps:

(1) Getting test prefixes. We use two types of test prefixes in this evaluation, each suited to different testing contexts. **(i) EvoSuite-generated prefixes**: EvoSuite automatically generates tests that cover a wide range of code paths by maximizing branch coverage. We obtain EvoSuite-generated test prefixes by running EvoSuite with default settings (i.e., coverage-guided) on the buggy program versions from Defects4J. By using these test prefixes, we can evaluate the effectiveness of the generated assertions in detecting potential bugs within real-world software projects, where bug locations are not predefined. Furthermore, the wide-ranging code coverage provided by EvoSuite also allows us to evaluate how well the assertions perform across diverse program scenarios, ensuring their applicability and reliability in various real-world conditions. **(ii) Defects4J-provided prefixes**: These prefixes are derived from predefined test cases specifically crafted to trigger bugs in real-world Java projects from Defects4J. By using these prefixes from bug-revealing test cases, we can directly evaluate $EditAS^2$'s ability to generate assertions that effectively detect predefined bugs. Note that our task is to generate assertions, so we exclude test prefixes for test cases that involve exception testing.

(2) Inferring test assertions. We first construct the focal-tests by combining the test prefixes (both EvoSuite-generated and Defects4J-provided) with the corresponding focal methods from the buggy programs. For each focal-test, we apply two versions of the assertion generation models trained in RQ3: one trained on $Data_{atlas}$ and the other on $Data_{integration}$. This setup ensures that assertions are generated using both versions of the model for each test prefix, allowing for a comprehensive evaluation across different training datasets. For IR-based models like $EditAS^2$, the retrieval corpus is aligned with the dataset used for training. The results are reported separately in Table 8 as "$Data_{atlas}$" and "$Data_{integration}$." In the case of pre-trained models and StarCoder, they are fine-tuned on both $Data_{atlas}$ and $Data_{integration}$, using focal-tests as inputs and corresponding assertions as outputs. Additionally, during fine-tuning, StarCoder is provided with a specific prompt: "Please fill the <AssertPlaceHolder> section with the appropriate assertions."

(3) Executing tests to find bugs. Each bug in Defects4J not only includes a buggy program version but also a bug-fixed program version. If a test case triggers an assertion error on the buggy program version but passes on the bug-fixed version, the generated assertion is considered capable of detecting the bug. Therefore, we combine the generated assertions with the test prefixes to execute on the buggy and bug-fixed versions of the program. Additionally, as EvoSuite is random, to avoid introducing bias by running EvoSuite only once per bug, we run it ten times and take the average for the BugFound metric. Due to the significant amount of time and computational resources required for compiling and running all test cases, and considering that EditAS exhibits the highest inference performance among all assertion generation methods in RQ3, we choose to compare the bug-finding performance of our method with that of EditAS, pre-trained models, and StarCoder.

TABLE 8
BugFound performance of assertions generated by different approaches trained on $Data_{atlas}$ and $Data_{integration}$ using EvoSuite and Defects4J prefixes.

Prefix Type	Approach	$Data_{atlas}$	$Data_{integration}$	Total
EvoSuite	EditAS	5.6	2.6	8.2
	CodeBERT	5.9	2.8	8.7
	GraphCodeBERT	1.4	1.9	3.3
	CodeGPT	4.4	4.2	8.6
	UnixCoder	10	3.7	13.7
	CodeT5	11.3	4.4	15.7
	StarCoder	6.2	2.4	8.6
	EditAS^2	**13.5**	**6.6**	**20.1**
Defects4J	EditAS	**19.0**	6.0	25.0
	CodeBERT	13.0	9.0	22.0
	GraphCodeBERT	17.0	9.0	26.0
	CodeGPT	11.0	**12.0**	23.0
	UnixCoder	18.0	6.0	24.0
	CodeT5	17.0	9.0	26.0
	StarCoder	10.0	**12.0**	22.0
	EditAS^2	**19.0**	8.0	**27.0**

Results indicate the average number of bugs detected by the assertions generated from each approach when trained on $Data_{atlas}$ and $Data_{integration}$ using EvoSuite and Defects4J prefixes.

6.2.1.2 Results

The experimental results in Table 8 provide a detailed comparison of the BugFound performance of different approaches trained on $Data_{atlas}$ and $Data_{integration}$ using EvoSuite and Defects4J-provided prefixes. The table reports the number of real-world bugs identified by each method when applied to buggy programs from Defects4J.

Overall performance. As shown, $EditAS^2$ achieves the best performance across both EvoSuite-generated and Defects4J-provided prefixes, outperforming EditAS and all

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2025.3558403

pre-trained models. Specifically, EDITAS^2 detects the highest total number of bugs for both prefix types, with a total of 20.1 bugs (13.5 on $Data_{atlas}$ and 6.6 on $Data_{integration}$) when using EvoSuite-generated prefixes, and 27 bugs (19 on $Data_{atlas}$ and 8 on $Data_{integration}$) when using Defects4J-provided prefixes. These results highlight that our method excels in both detecting potential and predefined bugs, thus proving its robustness in diverse real-world testing scenarios.

Performance with EvoSuite-generated prefixes. EDITAS^2 consistently outperforms other models in detecting potential bugs using EvoSuite-generated prefixes. On $Data_{atlas}$, EDITAS^2 detects 13.5 bugs, achieving an 19.47% improvement over the next best model, CodeT5, which detects 11.3 bugs. A similar trend is observed on $Data_{integration}$, where EDITAS^2 identifies 6.6 bugs, marking a 50.00% increase compared to CodeT5's 4.4 bugs. However, the overall bug detection performance remains relatively low, reflecting the challenges of identifying potential bugs in real-world scenarios. This limitation may be due to the low quality and limited context provided by EvoSuite-generated prefixes. Enhancing prefix generation techniques and incorporating richer contextual information could improve the bug detection capabilities of assertion generation methods in practical scenarios.

Performance with Defects4J-provided prefixes. The bug detection performance is consistently higher with Defects4J-provided prefixes compared to EvoSuite-generated ones across all methods. For example, EDITAS^2 identifies 19 bugs on $Data_{atlas}$ using Defects4J-provided prefixes, compared to just 13.5 bugs with EvoSuite-generated prefixes. This improvement is likely due to the manually crafted nature of Defects4J-provided prefixes, which are specifically designed to expose predefined bugs, offering more precise and targeted context for assertion generation. Furthermore, while EDITAS^2 achieves the best performance on $Data_{atlas}$ with Defects4J-provided prefixes, it finds 4 fewer bugs on $Data_{integration}$ than the highest-performing method, and EDITAS exhibits a similar trend. This difference may be due to noise issues in the $Data_{integration}$ dataset, such as inconsistent assertion formats, a large number of non-standard JUnit assertions, and empty focal methods. The fact that all models perform worse on $Data_{integration}$ compared to $Data_{atlas}$ further emphasizes the detrimental effect of noisy data on model validity.

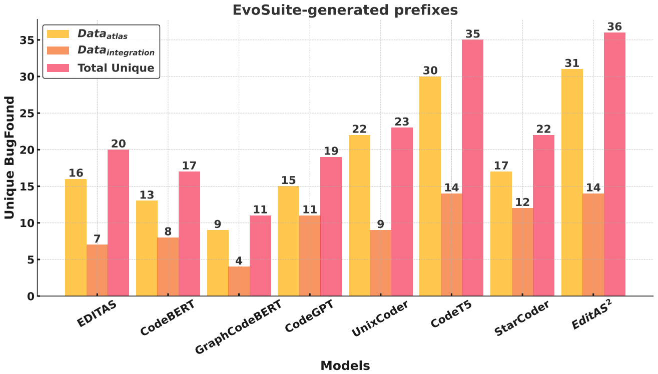


Fig. 7. Unique BugFound performance of assertions generated by different approaches using EvoSuite-generated prefixes.

Figure 7 shows the Unique BugFound performance of different assertion generation approaches using EvoSuite-generated prefixes. The y-axis represents the number of unique bugs detected. The x-axis lists the evaluated models. The bars indicate the unique bugs found across ten runs of EvoSuite. The yellow and orange bars show the unique bugs detected by assertions generated by models trained on $Data_{atlas}$ and $Data_{integration}$, respectively. The red bars represent Total Unique, which combines the distinct bugs identified from both datasets. Among all approaches, EDITAS^2 achieves the highest total unique bug detection, detecting 36 bugs, surpassing the next best model, CodeT5 (35 bugs), by 2.86%. UniXcoder and StarCoder detect 23 and 22 bugs, respectively, while EDITAS detects 20 bugs. Other models, including CodeGPT (19 bugs), CodeBERT (17 bugs), and GraphCodeBERT (11 bugs), exhibit lower detection capabilities. These results highlight the effectiveness of EDITAS^2 in detecting a diverse range of real-world bugs compared to other baselines. Additionally, we observe that for all models, Total Unique BugFound is always higher than the unique BugFound from a single dataset but lower than their sum. This indicates that training models on different datasets lead to the detection of different sets of bugs for the same test samples.

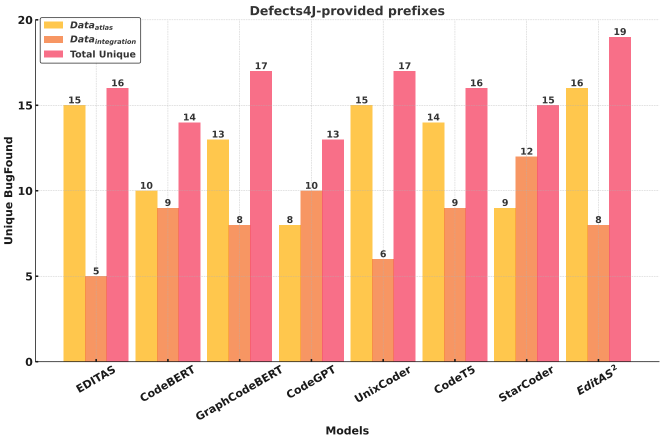


Fig. 8. Unique BugFound performance of assertions generated by different approaches using Defects4j-provided prefixes.

Figure 8 shows the Unique BugFound performance of different assertion generation approaches using Defects4J-provided prefixes. The y-axis represents the number of unique bugs detected. The x-axis lists the evaluated models. The yellow and orange bars show the unique bugs detected by assertions generated by models trained on $Data_{atlas}$ and $Data_{integration}$, respectively. The red bars represent Total Unique, which combines the distinct bugs identified from both datasets. Among all approaches, EDITAS^2 achieves the highest total unique bug detection, with 19 bugs. The next best models, GraphCodeBERT and UniXcoder, each detect 17 bugs. CodeT5 and EDITAS follow closely with 16 bugs each, while StarCoder (15 bugs), CodeBERT (14 bugs), and CodeGPT (13 bugs) show lower performance. These results demonstrate the effectiveness of EDITAS^2 in detecting a diverse set of real-world bugs. Additionally, for all models, the Total Unique BugFound is always greater than the number of unique bugs detected from either dataset alone but does not reach their combined total. This indicates that models trained on different datasets defect overlapping yet distinct sets of bugs for the same test samples.

Example 1: Mockito Project - Bug #26 Test Case

```
1    @Test
2    public void should_return_primitive() {
3        assertEquals(false, values.returnValueFor(Boolean.TYPE));
4        assertEquals((char) 0, values.returnValueFor(Character.TYPE));
5        assertEquals((byte) 0, values.returnValueFor(Byte.TYPE));
6        assertEquals((short) 0, values.returnValueFor(Short.TYPE));
7        assertEquals(0, values.returnValueFor(Integer.TYPE));
8        assertEquals(0L, values.returnValueFor(Long.TYPE));
9        assertEquals(0F, values.returnValueFor(Float.TYPE));
10       assertEquals(0D, values.returnValueFor(Double.TYPE));
11   }
```

Example 2: Math Project - Bug #37 Test Case

```
1    @Test
2    public void testTan() {
3        Complex z = new Complex(3, 4);
4        Complex expected = new Complex(-0.000187346, 0.999356);
5        TestUtils.assertEquals(expected, z.tan(), 1.0e-5);
6        /* Check that no overflow occurs (MATH-722) */
7        Complex actual = new Complex(3.0, 1E10).tan();
8        expected = new Complex(0, 1);
9        TestUtils.assertEquals(expected, actual, 1.0e-5);
10       actual = new Complex(3.0, -1E10).tan();
11       expected = new Complex(0, -1);
12       TestUtils.assertEquals(expected, actual, 1.0e-5);
13   }
```

Fig. 9. Examples of Bug-Revealing Test Cases from Defects4J Projects.

Given the low performance of all methods when using prefixes from Defects4J bug-revealing test cases, we analyze why many faults remain undetected. Our findings highlight two key issues. (1) *Many test cases have no test prefix.* 10.87% of test cases assert return values directly without any prefix steps. This prevents assertion generation models from establishing the necessary context, making their assertions less effective for bug detection. For example, in Figure 9 (Mockito project, Bug #26), the test case directly asserts return values, limiting the model's ability to generate useful assertions. (2) *Misleading prefixes in multi-assertion test cases.* More than half of the test cases (55.63%) contain multiple assertions. A multi-assertion test case has multiple prefixes. To construct the prefix for a specific assertion, we retain all preceding statements while removing the other assertions. For example, in Figure 9 (Math project, Bug #37), the test case includes multiple assertions. To construct the prefix for the second assertion (line 9), we remove the first assertion (line 5) but retain all other preceding statements (lines 1-4 and 6-8). However, removing the first assertion may leave behind the context that triggered it, which may mislead the generation of the intended assertion. In this case, the intended assertion on line 9, "*TestUtils.assertEquals(expected, actual, 1.0e-5)*", could be influenced by the earlier variable assignments on lines 3: "*Complex z = new Complex(3, 4);*". To address this, future work could extend the current method to support multi-assertion generation, better aligning with the complexity of real-world test cases.

> ***Summary-4.1***: The results indicate that the assertions generated by $EditAS^2$ achieve the highest bug detection performance compared to EDITAS and other pre-trained models, using both EvoSuite-generated and Defects4J-provided prefixes.

6.2.2 RQ4.2: What is the quality of assertions generated by the proposed approach in real scenarios?

6.2.2.1 Methodology

To assess the quality of assertions generated by the proposed approach in real-world scenarios, we utilize two key metrics: compilation rate and precision. By incorporating both metrics, we evaluate not only their syntactic correctness through compilation rate but also their practical effectiveness in detecting bugs. This dual evaluation provides deeper insights into the real-world applicability of the assertions. Additionally, we focus on EvoSuite-generated test prefixes, as EvoSuite offers broad code coverage, enabling us to evaluate assertion performance across a wide range of program scenarios. This ensures that the assertions are applicable and reliable under diverse real-world conditions. The experimental setup in this section aligns with RQ4.1, utilizing EvoSuite-generated test prefixes and model-generated test assertions.

TABLE 9
Compilation rate and bug detection precision of generated assertions across different approaches

Approach	Compilation Rate	Precision
EDITAS	33.87%	0.24%
CodeBERT	23.15%	0.41%
GraphCodeBERT	16.68%	0.32%
CodeGPT	19.14%	0.49%
UnixCoder	26.55%	0.61%
CodeT5	33.59%	0.26%
StarCoder	22.77%	0.39%
$EditAS^2$	30.97%	0.42%

6.2.2.2 Results

The results in Table 9 represent the average performance of $EditAS^2$ across the $Data_{atlas}$ and $Data_{integration}$ datasets. $EditAS^2$ shows a balanced performance in both compilation rate and precision.

For compilation rate, $EditAS^2$ achieves 30.97%, outperforming five baseline methods, including and CodeBERT (23.15%), GraphCodeBERT (16.68%), CodeGPT (19.14%), UnixCoder (26.55%), and StarCoder (22.77%), with improvements of 33.80%, 85.72%, 61.78%, 16.67%, and 36.05%, respectively. Notably, StarCoder, a large language model designed for code generation, shows a relatively low compilation rate of 22.77%, highlighting the challenges even powerful models face in generating syntactically correct assertions. Despite its significant model size, StarCoder performs notably worse in this area compared to EDITAS^2. While EDITAS^2 slightly trails CodeT5 (33.59%) and EDITAS (33.87%) by 2.6% and 2.9%, respectively, this may be due to the increased complexity of the input it processes. Unlike CodeT5, which relies solely on the focal-test, and EDITAS, which focuses on the edit sequence, EDITAS^2 integrates the input focal-test, the retrieved focal-test, and the edit sequence. This more comprehensive input structure could increase the amount of data being processed, leading to the risk of token truncation and the omission of important syntactic components needed for generating compilable code. Despite this slight disadvantage in compilation rate, EDITAS^2 demonstrates a clear advantage in precision, achieving 0.42%, which significantly surpasses both EDITAS (0.24%) and CodeT5 (0.26%). This suggests that the richer input context, while slightly impacting the compilation rate, allows EDITAS^2 to generate more accurate assertions that are highly effective in detecting bugs in successfully compiled test cases. To further enhance the compilation rate, future work could focus on extending input length or applying preprocessing techniques that prioritize key tokens, ensuring essential syntactic information is preserved while maintaining the model's high precision in bug detection.

For precision, EDITAS^2 achieves 0.42%, outperforming five baseline methods, including EDITAS (0.24%), CodeT5 (0.26%), StarCoder (0.39%), GraphCodeBERT (0.32%), and CodeBERT (0.41%), with improvements of 75%, 61.5%, 7.7%, 31.25%, and 2.4%, respectively. Although EDITAS^2 performs slightly below CodeGPT (0.49%) and UnixCoder (0.61%) in terms of precision, it notably outperforms both in compilation rate, where CodeGPT and UnixCoder achieve 19.14% and 26.55%, respectively, compared to EDITAS^2's 30.97%. Despite CodeGPT and UnixCoder showing slightly higher precision, their low compilation rates severely limit their practicality. Assertions that fail to compile cannot be executed, rendering them unusable for automated testing, which reduces the overall efficiency of these models in real-world scenarios. Additionally, it is important to note that precision across all models remains relatively low, indicating that achieving high precision in assertion generation remains a significant challenge for all approaches.

> ***Summary-4.2***: EDITAS^2 achieves a balanced performance in both compilation rate and precision, making it a reliable approach for assertion generation. However, the task itself remains inherently challenging, as both metrics have much room for improvement in all models, including large language models.

6.3 RQ5: How effective are the different variants?

6.3.1 Methodology

To enhance the semantic comprehension of the model, we take as inputs the input focal-test and the similar focal-test as well as the edit sequences of both. Each part of the input affects the effectiveness of the $EditAS^2$ differently. To assess the effectiveness of each input, we conduct a comparison between $EditAS^2$ and its two variants, namely $EditAS^2$-w/o focal-tests and $EditAS^2$-w/o edits. $EditAS^2$-w/o focal-tests uses only the edit sequence as input to generate test assertions. $EditAS^2$-w/o edits uses the inputs focal-test and similar focal-test to accomplish this task without the need for edit sequences.

TABLE 10
The effectiveness of each component of our approach

Variants	$Data_{atlas}$		$Data_{integration}$	
	Accuracy	CodeBLEU	Accuracy	CodeBLEU
$EditAS^2$-w/o focal-tests	39.53	68.71	42.95	63.11
$EditAS^2$-w/o edits	64.15	**82.12**	54.17	70.89
$EditAS^2$	**64.37**	81.89	**54.91**	**71.91**

6.3.2 Results

Table 10 shows the results for $EditAS^2$ and its variants. First, we find that $EditAS^2$ outperforms all variants in terms of accuracy on both datasets, suggesting that both the focal-tests themselves and the editing sequence contribute to generating correct test assertions. In particular, when the input focal-test and the similar focal-test are removed, the accuracy of $EditAS^2$-w/o focal-tests decreases by 18.40%, and CodeBLEU by 10.99% on average. This may be due to the fact that compared to EDITAS, we optimize the editing component using a CodeT5 pre-trained model, and the semantics of the focal-tests themselves can further complement CodeT5's understanding of semantic differences in the editing sequences, thus allowing the editing component to correctly modify the similar assertions based on the semantic differences.

> ***Summary***: Different inputs all have different effects on the effectiveness of $EditAS^2$. Focal-tests can further help the model understand the semantic differences in the editing sequence, thus making the model more prepared to edit similar assertions.

7 DISCUSSION

7.1 Threats to validity

The validity of our approach faces four primary challenges. Firstly, following previous works [8], [14], our experiments are conducted solely on two Java datasets. While Java may not be representative of all programming languages, Java is the most attacked language in the field of assertion generation, and the datasets used in our experiments are large enough to draw reliable conclusions. In addition, both our retrieval module and the pre-trained model $EditAS^2$ employed can support multiple programming languages. Secondly, the Retrieve component retrieves a similar focal-test based on lexical similarity. This may result in the retrieved focal-test and input focal-test being similar only in terms of lexicon, but displaying different assertions. To mitigate this threat, we utilize a large Java dataset (247M) to enhance the size and diversity of the retrieved corpus. Additionally, considering the semantic differences between the input and retrieved focal-tests, we have introduced an editing component to address this threat by adapting the prototype. The third significant threat arises from the potential for data leakage from pre-trained models. Given that pre-trained models frequently utilize public repositories like GitHub for their pre-training data [12], [35], there's a chance that the pre-training data of the models we employ could include test methods from the dataset we used. To tackle this concern, we analyze the pre-training datasets (such as CodeSearchNet) utilized by all the pre-trained models studied in this paper. Our investigation reveals that these models lack access to any of the test cases, including assertions, throughout the pre-training phase. As a result, we are assured that our conclusions remain unaffected by any potential data leakage. Additionally, a potential threat to validity comes from the limited number of examples used to demonstrate that "EDITAS's semantic understanding ability needs improvement". In RQ2, we observe that even when EDITAS finds assertions that exactly match the ground truth, it may still modify them incorrectly, while failing to modify assertions that require adjustment. We report the numbers for these cases and provide two key examples to illustrate that these issues arise because the model struggles to recognize the semantic differences between the input and retrieved focal-tests. However, a more systematic analysis would strengthen this claim. Since such analysis requires a lot of manual effort, our future work will extend this analysis

TABLE 11
EDIT DISTANCE (E) BETWEEN RETRIEVED ASSERTIONS AND GROUND TRUTH FOR TEST SETS (EDITAS VS $EditAS^2$)

Dataset	Prediction	Approach	$E=0$	$E=1$	$E=2$	$E=3$	$E=4$	$E=5$	$E>5$	Total
$Data_{atlas}$	correct	EDITAS	5093 (60.78%)	1225 (14.62%)	474 (5.66%)	241 (2.88%)	188 (2.24%)	151 (1.80%)	1008 (12.03%)	8380
		$EditAS^2$	5231 (51.84%)	1673 (16.58%)	631 (6.25%)	327 (3.24%)	279 (2.77%)	240 (2.38%)	1709 (16.94%)	10090
	incorrect	EDITAS	594 (8.14%)	1154 (15.82%)	455 (6.24%)	358 (4.91%)	338 (4.63%)	346 (4.74%)	4051 (55.52%)	7296
		$EditAS^2$	456 (8.16%)	706 (12.64%)	298 (5.33%)	272 (4.87%)	247 (4.42%)	257 (4.60%)	3350 (59.97%)	5586
$Data_{integration}$	correct	EDITAS	9410 (79.93%)	1130 (9.60%)	398 (3.38%)	190 (1.61%)	127 (1.08%)	73 (0.62%)	445 (3.78%)	11773
		$EditAS^2$	9414 (64.59%)	1930 (13.24%)	858 (5.89%)	432 (2.96%)	345 (2.37%)	230 (1.58%)	1366 (9.37%)	14575
	incorrect	EDITAS	703 (4.76%)	1931 (13.07%)	1173 (7.94%)	834 (5.65%)	745 (5.04%)	611 (4.14%)	8772 (59.39%)	14769
		$EditAS^2$	699 (5.84%)	1131 (9.45%)	713 (5.96%)	592 (4.95%)	527 (4.40%)	454 (3.79%)	7851 (65.61%)	11967

TABLE 12
NUMBER OF EDITS (N) MADE BY DIFFERENT APPROACHES FOR INCORRECT ASSERTIONS (EDITAS VS $EditAS^2$)

Dataset	Approach	$N=0$	$N=1$	$N=2$	$N=3$	$N=4$	$N=5$	$N>5$	Total
$Data_{atlas}$	EDITAS	1424 (19.52%)	1030 (14.12%)	554 (7.59%)	344 (4.71%)	336 (4.61%)	311 (4.26%)	3297 (45.19%)	7296
	$EditAS^2$	733 (13.12%)	883 (15.81%)	363 (6.50%)	253 (4.53%)	226 (4.05%)	252 (4.51%)	2876 (51.49%)	5586
$Data_{integration}$	EDITAS	4670 (31.62%)	1783 (12.07%)	1001 (6.78%)	839 (5.68%)	576 (3.90%)	470 (3.18%)	5430 (36.77%)	14769
	$EditAS^2$	1435 (11.99%)	1439 (12.02%)	917 (7.66%)	597 (4.99%)	509 (4.25%)	498 (4.16%)	6572 (54.92%)	11967

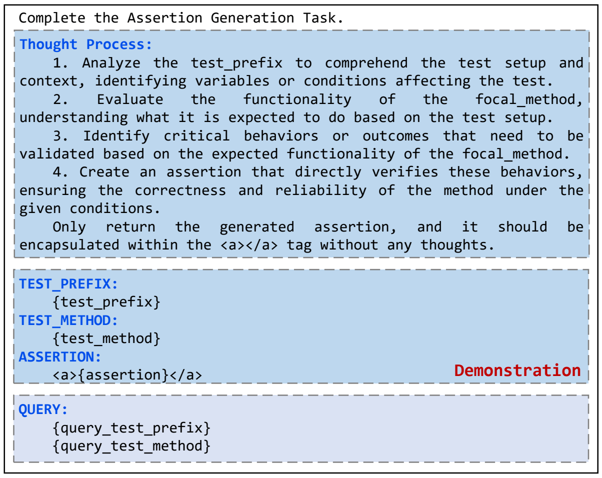
with a wider set of examples to more comprehensively evaluate the semantic understanding ability of EDITAS.

7.2 A systematic evaluation of EditAS^2

To explore the improvement of $EditAS^2$ over EDITAS, we analyze and compare the assertions generated by $EditAS^2$ and EDITAS.

(1) $EditAS^2$ improves the editing ability of EDITAS. Table 11 categorizes assertions according to whether they are generated correctly (see *column: "Prediction"*) and then classifies them according to the edit distance between the retrieved assertions and the ground truth (see *column:"E"*). ① We observe that $EditAS^2$ consistently outperforms EDITAS across all edit distances. For example, when $E=0$, the accuracy of EDITAS in $Data_{atlas}$ is 89.56% = 5093 / (5093 + 594), while $EditAS^2$ achieves 91.98% = 5231 / (5231 + 456). when $E=1$, the accuracy of EDITAS in $Data_{atlas}$ is $51.49\% = (1225/1225+1154)$, while $EditAS^2$ achieves $70.32\% = 1673/(1673+706)$. When $E=3$, the accuracy of EDITAS in $Data_{atla}$ is $40.23\% = 241/(241+358)$, while the accuracy of $EditAS^2$ reaches $54.59\% = 327/(327+272)$. Even when $E>5$, the accuracy of EDITAS in $Data_{atla}$ is $19.92\% = 1008/(1008+4051)$, while the $EditAS^2$ performs significantly better with an accuracy of $33.78\% = 1709/(1709+3350)$. These results demonstrate that $EditAS^2$ improves the editing ability over EDITAS across all edit distances. ② Furthermore, we observe that while the performance of $EditAS^2$ still decreases as the edit distance increases, it mitigates this decline. For example, among all correctly generated assertions, the proportion with $E>3$ increases from 2.88% (in EDITAS) to 3.24% (in $EditAS^2$). For $E>5$, the proportion even rises from 12.03% (in EDITAS) to 16.94% (in $EditAS^2$), demonstrating that $EditAS^2$ mitigates the negative effect of larger edit distances.

(2) $EditAS^2$ improves the semantic understanding ability of EDITAS. ① As reported in Table 11, we can see that in $Data_{atlas}$, EDITAS has 594 test samples (when $E=0$) whose retrieved assertions match the ground truth but are still modified by EDITAS. However, $EditAS^2$ reduces this number to 456. A similar reduction is observed in $Data_{integration}$. ② Furthermore, we count the number of edits made by different approaches for those incorrect assertions. As shown in Table 12, we can see that a large proportion of the incorrect assertions in EDITAS are not modified at all, but rather the retrieved assertions are output directly. In $Data_{atlas}$, EDITAS has 1,424 such cases (when $N=0$), accounting for 19.52% of incorrect assertions. However, $EditAS^2$ reduces this number to 733, cutting it by nearly half. Similarly, in $Data_{integration}$, EDITAS has 4,670 such cases, while $EditAS^2$ reduces this number to 1,435.



```
Complete the Assertion Generation Task.
Thought Process:
    1. Analyze the test_prefix to comprehend the test setup and
context, identifying variables or conditions affecting the test.
    2. Evaluate the functionality of the focal_method,
understanding what it is expected to do based on the test setup.
    3. Identify critical behaviors or outcomes that need to be
validated based on the expected functionality of the focal_method.
    4. Create an assertion that directly verifies these behaviors,
ensuring the correctness and reliability of the method under the
given conditions.
    Only return the generated assertion, and it should be
encapsulated within the <a></a> tag without any thoughts.

TEST_PREFIX:
    {test_prefix}
TEST_METHOD:
    {test_method}
ASSERTION:
    <a>{assertion}</a>                        Demonstration

QUERY:
    {query_test_prefix}
    {query_test_method}
```

Fig. 10. Chain-of-Thought Prompting.

7.3 Exploring Chain-of-Thought techniques in assertion generation

In this section, we explore the effectiveness of the Chain-of-thought (CoT) technique on the assertion generation task by evaluating two widely used models: StarCoder2-15b-instruct-v0.1 [61] and Qwen2.5-Coder-7B-Instruct [62]. We conduct experiments with both models on the $Data_{atlas}$ and $Data_{integration}$ datasets to assess their ability to generate assertions incorporating CoT reasoning. For both models, we use a CoT-based prompt, as shown in Figure 10. This prompt first instructs the models to reason through the test setup, focal-method functionality, and expected outcomes before generating assertions. A demonstration is provided, where the models are shown an example consisting of a focal-test and its corresponding expected assertion. The results, as shown in Table 13, indicate that $EditAS^2$ outperforms both StarCoder and Qwen2.5-Coder-7B-Instruct in terms of accuracy and CodeBLEU across both datasets. Specifically, $EditAS^2$ achieves 64.37% accuracy and 81.89% CodeBLEU on $Data_{atlas}$, and 54.91% accuracy with 71.91% CodeBLEU on $Data_{integration}$. In contrast, StarCoder shows limited performance, with 0.20% accuracy and 6.07% CodeBLEU on $Data_{atlas}$, and 0.01% accuracy and 0.68% CodeBLEU on $Data_{integration}$. Qwen2.5-Coder-7B-Instruct performs better than StarCoder but still lags behind $EditAS^2$, with 31.68% accuracy and 17.79% CodeBLEU on $Data_{atlas}$, and 12.07% accuracy and 8.48% CodeBLEU on $Data_{integration}$. These results demonstrate that $EditAS^2$ outperforms large models that incorporate CoT reasoning, demonstrating its superior effectiveness in assertion generation tasks.

TABLE 13
Comparisons of our approach with chain-of-thought techniques in terms of inference performance

Approach	$Data_{atlas}$		$Data_{integration}$	
	Accuracy	**CodeBLEU**	**Accuracy**	**CodeBLEU**
starcoder2-15b-instruct-v0.1	0.20%	6.07%	0.01%	0.68%
Qwen2.5-Coder-7B-Instruct	31.68%	17.79%	12.07%	8.48%
$EditAS^2$	**64.37%**	**81.89%**	**54.91%**	**71.91%**

7.4 Distribution of detected bugs across projects

Table 14 presents the distribution of unique bugs detected by $EditAS^2$ using EvoSuite-generated prefixes on the two datasets, $Data_{atlas}$ and $Data_{integration}$. The table lists the projects along with the specific bug IDs detected by $EditAS^2$. The results show that $EditAS^2$ detects the most unique bugs in Jsoup (7 bugs) and Math (6 bugs), followed by Closure and JxPath (5 bugs each). Chart detects 3 bugs, while Compress and JacksonDatabind detect 2 bugs each. Codec, Csv, and Time each detect 1 bug. Additionally, we observe that five projects, Collections, Gson, JacksonCore, JacksonXml, and Lang, fail to detect any bugs. This could be due to two factors. First, these projects have a limited number of bugs that can be triggered by assertions. For instance, JacksonXml, Gson, and Collections have only 5, 4, and 2 bugs, respectively, that can be triggered by assertions. Second, the quality of the prefixes generated by EvoSuite may be suboptimal for some projects, leading to ineffective assertion generation. For example, JacksonCore and Lang detect no bugs using EvoSuite-generated prefixes, but detect 2 and 4 bugs, respectively, when using the Defects4J-provided prefixes.

7.5 Comparison with TOGA

TOGA, introduced by Dinella [3], addresses the assertion generation problem by using a ranking architecture over a set of candidate test assertions. It employs grammar along with type-based constraints to limit the generation space of candidates and uses a transformer-based neural approach to obtain the most probable assertions. We find that the formal implementation of TOGA [63], [64] requires a seed/approximate assertion to determine the variables for constructing and optimizing the candidate assertion set. Also, the seed/approximate assertion needs to be created via EvoSuite tool [6]. Unlike TOGA, $EditAS^2$ focuses exclusively on assertion generation based on the focal-test, without relying on any seed assertion.

TABLE 14
Project distribution of total unique bug detected by EDITAS^2

Project	Unique Bug_ID
Chart	11,18,22
Cli	8,29
Closure	7,28,100,106,164
Codec	6
Compress	42,47
Csv	3
JacksonDatabind	3,12
Jsoup	3,16,32,41,49,58,71
JxPath	6,7,8,9,20
Math	22,29,62,92,94,102
Mockito	35
Time	1

In this section, we compare the bug detection capabilities of these two approaches. Our experiments are conducted using EvoSuite, with the same settings as those in RQ4. For comparison, we use TOGA's open-source pre-trained classification model. This classification model is trained on $Atlas^*$, a variant of $Data_{atlas}$ with added candidate assertions. Our method is trained on $Data_{atlas}$. As shown in Figure 11, $EditAS^2$ detects 31 unique bugs, and TOGA detects 33. Notably, only 10 bugs are detected by both methods, while $EditAS^2$ identifies 21 unique bugs that TOGA fails to detect. This illustrates that the two methods, with their different assertion generation strategies, complement each other in terms of bug detection. Each method uncovers a distinct set of bugs, suggesting that combining these approaches could provide more comprehensive bug detection.

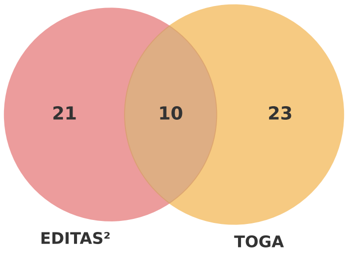


Fig. 11. The overlaps of the unique bugs detected by $EditAS^2$ and TOGA.

7.6 Manual analysis of compilation failures in generated assertions

To explore why some assertions generated by $EditAS^2$ fail to compile, we perform a manual analysis of these failures, identifying common compilation failure patterns. We randomly sample 384 failed assertions, ensuring a 95% confidence level with a 10% confidence interval. Three volunteers with research experience in software evolution and at least five years of Java programming experience—one PhD student and two master's students—review the failures.

They identify the causes and categorize them. For each sample, we provide the test case, compiler error message, and focal method. Each volunteer first assigns tags independently. Then, they compare their tags with a shared tag set. If a similar tag exists, they adopt it; otherwise, they add a new tag. When disagreements occur, the volunteers discuss the differences until they reach a consensus.

As a result of our manual analysis, we classify the compilation failures into five main categories. The number in parentheses represents the number of occurrences of each error type within our sampled dataset. The most frequent errors are *Cannot Find Symbol (227 cases)* and *Syntax Errors (119 cases)*, which together account for the majority of failures. Other notable errors include *Method Signature Mismatch (19 cases)* and *Type Mismatch (18 cases)*. Additionally, we observe a rare case of *Invalid instanceof Check with Generics (1 case)*, caused by Java's type erasure mechanism. The specific classification results are as follows:

(1) **Syntax Errors (119)**: *Syntax errors* occur when the generated assertion does not conform to the grammatical rules of the programming language, leading to compilation failures. These errors often stem from missing ";", mismatched ")" or "}", illegal escape characters, unclosed string literals, or incorrect syntax structures. For instance, a failure occurs due to a mismatched closing ")" in the generated assertion: *"assertEquals(0, list0.size();;"*

(2) **Cannot Find Symbol (227)**: A *Cannot Find Symbol* error occurs when the compiler fails to recognize a referenced symbol, such as a class, method, or variable. This error typically arises due to undefined variables or methods. For example, a failure occurs when referencing an undefined method in the generated assertion: *"assertEquals(-1, namedType0.getIndex());"*. Here, *getIndex()* is not defined for *namedType0*, causing a compilation error.

(3) **Method Signature Mismatch (19)**: A *Method Signature Mismatch* error occurs when the parameters passed in a method call do not match the expected method signature. This error mainly arises due to incompatible parameter types, incorrect argument count, or method overload ambiguity. For example, a failure occurs when the generated assertion invokes a method with an argument that does not conform to the expected signature: *"assertTrue(ongoingInjecter0.thenInject(null)); "*. Here, the method *thenInject* in *OngoingInjecter* does not accept *null* as an argument, leading to a compilation error.

(4) **Type Mismatch (18)**: A *Type Mismatch* error occurs when an operation involves incompatible data types, such as using mismatched primitive types, method return types that do not align with expectations, or numeric values exceeding permissible limits. For example, a failure occurs when the generated assertion includes a numeric literal that exceeds the allowed range: *"Assert.assertEquals(6801737418242, long0);"*. The error occurs because *6801737418242* exceeds the *int* limit, and Java interprets numeric literals as *int* by default.

(5) **Invalid *instanceof* Check with Generics (1)**: This error occurs when *instanceof* is used with a parameterized generic type, which is not allowed due to Java's type erasure. For example, the following generated assertion results in a compilation error: *"Assert.assertTrue(abstractSubHyperplane0 instanceof AbstractSubHyperplane <Euclidean2D, Euclidean1D>);"*. The error occurs because generic type parameters are erased at runtime, making such checks invalid.

8 Conclusion and Future Works

In this paper, we reaffirm the advantages of the retrieve-and-edit approach for the assertion generation task and emphasize the shortcomings of the previous approach in terms of semantic understanding and editing abilities. To alleviate these problems, we propose a novel approach that improves retrieval-and-edit based assertion generation through retrieval-augmented fine- tuning, namely $EditAS^2$. $EditAS^2$ contains two components. A retrieve component is used to retrieve the similar focal-test, which consists of the test method without any assertions and its focal method. An editing component first considers the assertions of the similar focal-test as a prototype. Subsequently, it employs a pre-trained model, CodeT5, to learn the semantics of the focal-tests and assertion editing patterns reflected by the semantic differences to edit the prototype. Experimental results on two widely-adopted Java datasets show that $EditAS^2$ substantially outperforms the state-of-the-art baselines. Furthermore, we demonstrate the effectiveness of $EditAS^2$ in finding real-world bugs. Our source code and experimental data are available at https://github.com/Lhy-apple/EditAs2.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (No. 62372071), the Chongqing Technology Innovation and Application Development Project (No. CSTB2022TIAD-STX0007 and No. CSTB2023TIADSTX0025), the Natural Science Foundation of Chongqing (No. CSTB2023NSCQ-MSX0914) and the Fundamental Research Funds for the Central Universities (No. 2023CDJKYJH013).

References

[1] A. Hartman, "Is ISSTA research relevant to industry?" in Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002. ACM, 2002, pp. 205–206. [Online]. Available: https://doi.org/10.1145/566172.566207

[2] S. Planning, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, vol. 1, 2002.

[3] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A neural method for test oracle generation," in 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022. ACM, 2022, pp. 2130–2141. [Online]. Available: https://doi.org/10.1145/3510003.3510141

[4] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in 25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014. IEEE Computer Society, 2014, pp. 201–211. [Online]. Available: https://doi.org/10.1109/ISSRE.2014.11

[5] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada. ACM, 2007, pp. 815–816. [Online]. Available: https://doi.org/10.1145/1297846.1297902

[6] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011. ACM, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[7] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017. IEEE Computer Society, 2017, pp. 263–272. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.27

[8] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1398–1409. [Online]. Available: https://doi.org/10.1145/3377811.3380429

[9] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," arXiv preprint arXiv:2308.01240, 2023.

[10] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 2450–2462.

[11] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. GONG, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. LIU, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks, J. Vanschoren and S. Yeung, Eds., vol. 1. Curran, 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c16a5320fa475530d9583c34fd356ef5-Paper-round1.pdf

[12] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," CoRR, vol. abs/2109.00859, 2021. [Online]. Available: https://arxiv.org/abs/2109.00859

[13] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[14] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated assertion generation via information retrieval and its integration with deep learning," in Proceedings of the 44th International Conference on Software Engineering, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 163–174. [Online]. Available: https://doi.org/10.1145/3510003.3510149

[15] W. Sun, H. Li, M. Yan, Y. Lei, and H. Zhang, "Revisiting and improving retrieval-augmented deep assertion generation," in 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2023, pp. 1123–1135.

[16] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 146–158. [Online]. Available: https://doi.org/10.1145/3611643.3616256

[17] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 1482–1494.

[18] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," 2020.

[19] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," in IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2022, Pittsburgh, PA, USA, May 21-22, 2022. ACM/IEEE, 2022, pp. 54–64. [Online]. Available: https://doi.org/10.1145/3524481.3527220

[20] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader-Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, 2021, pp. 336–347. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00041

[21] A. Mastropaolo, N. Cooper, D. Nader-Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," IEEE Trans. Software Eng., vol. 49, no. 4, pp. 1580–1598, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3183297

[22] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," J. Mach. Learn. Res., vol. 21, no. 1, jan 2020.

[23] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. rahman Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in Annual Meeting of the Association for Computational Linguistics, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:204960716

[24] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, "Learning deep semantics for test completion," in 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 2111–2123.

[25] H. Wang, T. Xu, and B. Wang, "Deep multiple assertions generation," in 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering (Forge) Conference Acronym:, 2024, pp. 1–11.

[26] T. Tanimoto, An Elementary Mathematical Theory of Classification and Prediction. International Business Machines Corporation, 1958. [Online]. Available: https://books.google.com.hk/books?id=yp34HAAACAAJ

[27] Q. Zhang, W. Sun, C. Fang, B. Yu, H. Li, M. Yan, J. Zhou, and Z. Chen, "Exploring automated assertion generation via large language models," ACM Trans. Softw. Eng. Methodol., Oct. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3699598

[28] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, and N. Sundaresan, "Automating code review activities by large-scale pre-training," in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1035–1047. [Online]. Available: https://doi.org/10.1145/3540250.3549081

[29] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "Cct5: A code-change-oriented pre-trained model," in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1509–1521. [Online]. Available: https://doi.org/10.1145/3611643.3616339

[30] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "Coditt5: Pretraining for source code and natural language editing," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556955

[31] X. Zhou, B. Xu, D. Han, Z. Yang, J. He, and D. Lo, "Ccbert: Self-supervised code change representation learning," in 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2023, pp. 182–193.

[32] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet NL2Code: A survey," in Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 7443–7464. [Online]. Available: https://aclanthology.org/2023.acl-long.411

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in Proceedings of the 31st International Conference on Neural Information Processing Systems, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[34] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "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), S. Muresan, P. Nakov, and A. Villavicencio, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 7212–7225. [Online]. Available: https://aclanthology.org/2022.acl-long.499

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in Findings of the Association for Computational Linguistics: EMNLP 2020, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[36] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcode{bert}: Pre-training code representations with data flow," in International Conference on Learning Representations, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[37] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing LSTM language models," in 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=SyyGPP0TZ

[38] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," CoRR, vol. abs/1409.2329, 2014. [Online]. Available: http://arxiv.org/abs/1409.2329

[39] C. Thunes, "Javalang." https://github.com/c2nes/javalang, 2019.

[40] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization," in 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021. IEEE, 2021, pp. 155–166. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678724

[41] Z. Liu, X. Xia, M. Yan, and S. Li, "Automating just-in-time comment updating," in 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. IEEE, 2020, pp. 585–597. [Online]. Available: https://doi.org/10.1145/3324884.3416581

[42] Z. Liu, X. Xia, D. Lo, M. Yan, and S. Li, "Just-in-time obsolete comment detection and update," IEEE Trans. Software Eng., vol. 49, no. 1, pp. 1–23, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2021.3138909

[43] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," arXiv preprint arXiv:1907.11692, 2019.

[44] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," arXiv preprint arXiv:1508.07909, 2015.

[45] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," J. Mach. Learn. Res., vol. 21, no. 1, jan 2020.

[46] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:160025533

[47] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[48] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018.

[49] K. Clark, M. T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," 2020.

[50] "BigQuery," https://console.cloud.google.com/marketplace/details/github/github-repos.

[51] S. Ayupov and N. Chirkova, "Parameter-efficient finetuning of transformers for source code," ArXiv, vol. abs/2212.05901, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:254564456

[52] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," arXiv preprint arXiv:2009.05617, 2020.

[53] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards more realistic evaluation for neural test oracle generation," 2023.

[54] "Hugging face," site: https://huggingface.co/., 2023.

[55] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," Journal of Machine Learning Research, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[56] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "Circle: continual repair across programming languages," in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 678–690. [Online]. Available: https://doi.org/10.1145/3533767.3534219

[57] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[58] https://pytorch.org/.

[59] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," IEEE Transactions on Software Engineering, vol. 45, no. 2, pp. 111–147, 2017.

[60] C. Zhang, B. Liu, Y. Xin, and L. Yao, "Cpvd: Cross project vulnerability detection based on graph attention network and domain adaptation," IEEE Transactions on Software Engineering, vol. 49, no. 8, pp. 4152–4168, 2023.

[61] "starcoder2-15b-instruct-v0.1," https://huggingface.co/bigcode/starcoder2-15b-instruct-v0.1.

[62] "Qwen2.5-Coder-7B-Instruct," https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct.

[63] "ToGA Artifact." https://github.com/microsoft/toga, 2022.

[64] "A issue bug of ToGA Artifact." https://github.com/microsoft/toga/issues/3, 2022.