CHATASSERT: LLM-Based Test Oracle Generation With External Tools Assistance

Ishrak Hayet, Adam Scott, and Marcelo d'Amorim

***Abstract*—Test oracle generation is an important and challenging problem. Neural-based solutions have been recently proposed for oracle generation but they are still inaccurate. For example, the accuracy of the state-of-the-art technique TECO is only 27.5% on its dataset including 3,540 test cases. We propose CHATASSERT, a prompt engineering framework designed for oracle generation that uses dynamic and static information to iteratively refine prompts for querying large language models (LLMs). CHATASSERT uses code summaries and examples to assist an LLM in generating candidate test oracles, uses a lightweight static analysis to assist the LLM in repairing generated oracles that fail to compile, and uses dynamic information obtained from test runs to help the LLM in repairing oracles that compile but do not pass. Experimental results using an independent publicly-available dataset show that CHATASSERT improves the state-of-the-art technique, TECO, on key evaluation metrics. For example, it improves *Acc@1* by 15%. Overall, results provide initial yet strong evidence that using external tools in the formulation of prompts is an important aid in LLM-based oracle generation.**

***Index Terms*—Test oracle generation, large language models (LLMs), tool-augmented LLMs, prompt engineering framework.**

I. INTRODUCTION

SOFTWARE testing is a widely adopted technique for software quality assurance, but it is very time-consuming. Automated test case generation promises to reduce this cost, but it is challenging. Test oracle generation is particularly challenging to automate as the oracle needs to capture the intent of the program to be tested. Neural techniques have been recently proposed to generate oracles based on contextual data [1], [2], [3], [4], [5] but they are still inaccurate. For example, the state-of-the-art (SoTA) approach for test oracle generation, TECO [5], produces the ground truth oracle among the ten oracles it reports in only 42% of the cases (Section V-D). It is therefore imperative to improve the accuracy of these techniques if we want them to work in realistic settings.

Recent prior work applied large language models (LLMs) in a variety of code-related tasks, e.g., program repair [6], [7], [8], [9], code synthesis [10], test generation [5], [11], [12], and filtering static analysis warnings [13]. LLMs, such as CHATGPT [14], are convenient to developers because they do not need to gather large amounts of data to train a model for a specific task. LLMs are pre-trained on simple general-purpose tasks using huge amounts of data. They can be useful for solving more specific downstream tasks provided users carefully design prompts describing the task to be solved (Section II-A).

To assess how promising the use of LLMs for oracle generation can be, we evaluate how a CHATGPT [16] configured with a simplistic prompt compares against the SoTA technique TECO [5] (Section V-B). Very poor results would indicate a low potential for using LLMs for oracle generation. Results obtained by running TECO and the simplistic CHATGPT-based tool on the TECO dataset show that the LLM-based tool achieves an $Acc@1$ score 1% higher compared to that of TECO. We interpret this result as encouraging to motivate the use of LLMs for oracle generation.

Yet, we observe that without appropriate prompt engineering, LLMs face an important challenge of reasoning about the code being analyzed [17], [18], [19], [20]. LLMs are oblivious to the functionality to be tested in the program. Fig. 1(a) shows an example test case from project `NanoHttp` [15] asserting that the size of the set that the expression `cookies.getCookies()` denotes should be one. When provided with the prefix of the test case, excluding the assertion, the SoTA technique TECO [5] often generates inaccurate and invalid assertions as Fig. 1(b) highlights. TECO is limited to using a restrictive set of code features for training, including information about local types, absent types, unset fields, setup methods, last called methods, and similar statements. Additionally, TECO lacks the ability to repair the failing or non-compiling test oracles it generates.

We propose CHATASSERT to mitigate those limitations. CHATASSERT leverages static and dynamic information to iteratively refine the prompts of a conversational LLM, such as CHATGPT. Through iterative refinement of prompts, CHATASSERT avoids incoherent outputs and eventually generates exact matches. For example, CHATASSERT produces the exact match assertion for the test from Fig. 1. Notably, CHATASSERT can handle complex assertions by understanding the relationships between methods and objects within the code. For example, CHATASSERT is capable of recognizing that the `getCookies` method is declared in the type

---

Received 5 January 2024; revised 25 November 2024; accepted 6 December 2024. Date of publication 16 December 2024; date of current version 10 January 2025. This work was supported by the National Science Foundation under Grant 2319472 and Grant 2349961. Recommended for acceptance by L. Ma. *(Corresponding author: Ishrak Hayet.)*

The authors are with the North Carolina State University, Raleigh, NC 27695 USA (e-mail: ihayet@ncsu.edu; amscott9@ncsu.edu; mdamori@ncsu.edu).

Digital Object Identifier 10.1109/TSE.2024.3519159

0098-5589 © 2024 IEEE. All rights reserved, including rights for text and data mining, and training of artificial intelligence and similar technologies.
Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

```
@Test
public void testCookieSentBackToClient() throws ... {
  this.testServer.cookiesToSend.add(new Cookie(...));
  HttpGet httpget=new HttpGet("STR");
  ResponseHandler<String> rHandler=new Basic...();
  this.httpclient.execute(httpget, rHandler);
  CookieStore cookies=this.httpclient.getCookieStore();
  assertEquals(1, cookies.getCookies().size());
}
//Focal
public Cookie(String name, String value, int numDays)
     {...}
```

(a) Test sequence and associated focal method from project NanoHttpd [15]

```
assertThat(cookies, hasItem("STR"));
Cookie cookie = cookies.
     get("STR");(C,...,=,c,...,.,g,e,t,(,",S,T,R,",),;);
assertThat(cookies).hasSize(1);
assertEquals(1, cookies.size());
assertNotNull(cookies);
```

(b) Assertions generated by the TECO. Highlighted assertions are invalid.

```
assertEquals(1, cookies.getCookies().size());
assertEquals("STR",cookies.getCookies().get(0).getName());
assertTrue(cookies.getCookies().size() > 0);
assertEquals("STR",cookies.getCookies().get(0).getValue());
assertNotEquals("STR",cookies.get...().get(0).getDomain());
```

(c) Assertions generated by ChatAssert. Highlighted assertion is an exact match with the ground truth.

Fig. 1. Comparison of assertions generated by TECO and ChatAssert for the test case `testCookieSentBackToClient`.

associated with the variable `cookies`. ChatAssert observes that the method `getCookies` returns a `List` and appropriately calls the `size` method on the returned `List` object. As a result, ChatAssert produces the correct assertion in this case. Fig. 1(b) shows the invalid assertion that TECO generates for this case. We conjecture that TECO fails to identify that the method `getCookies` could be called on the object `cookies`. Also, TECO does not prevent the generation of syntactical nonsense as reflected in the fragment starting with `(C,...`.

ChatAssert mitigates the semantic gap problem of LLMs by *iteratively refining prompts with sensible information obtained from external tools*, e.g., the output of code summarization tools, the compiler output, the output of test runs, etc. ChatAssert has two modes of execution: generation and repair. In generation mode, ChatAssert uses code summaries (CS) and examples (EX) to produce candidate test oracles. In repair mode, ChatAssert uses a lightweight static analysis to assist the LLM in repairing oracles that fail to compile (SR) and it uses dynamic information obtained from test runs to assist the LLM in repairing oracles that compile but do not pass (DR). ChatAssert alternates between these two modes of execution. ChatAssert queries the LLM iteratively, using prompts augmented with information collected from previous iterations.

To understand the impact of ChatAssert's prompt engineering method, we rigorously evaluated its oracle generation capabilities when configured with various open and close-sourced LLMs. This study involved testing LLMs with all features of ChatAssert enabled and then with no features enabled across four popular LLMs, namely Mistral [21], Codestral [22], and Magicoder [23], and ChatGPT [14]. The results consistently show that ChatAssert's prompt engineering significantly enhances the performance of all tested LLMs. Notably, ChatGPT with ChatAssert's prompt engineering exhibits a **30%** performance improvement compared to ChatGPT with a simplistic prompt. This improvement is well above the improvement that the other LLMs obtain with ChatAssert's prompt engineering; 9% on average. These findings highlight the crucial role of prompt engineering in unleashing the full potential of LLMs for oracle generation. Table III presents detailed results of this comparison.

We compared ChatAssert against prior work, including ATLAS [1], TOGA [4], and TECO [5]. To sum up, results show that ChatAssert performs better overall. For example, compared to TECO, ChatAssert improves *Acc@10* by **12%** and improves three of the four standard NLP metrics we considered: BLEU, CodeBLEU, and Rouge. Table IV shows results. Mutation scores of the oracles that ChatAssert generate are also significantly higher compared to that of TECO. Fig. 6 summarizes the results.

We conducted an ablation study to assess the contribution of each of ChatAssert's features. We found that all features had an impact on ChatAssert's performance. The contribution of examples (i.e., feature EX) was slightly less than that of the other features. Table VI shows the results.

Finally, we compared the performance of the variant of ChatAssert that excludes the feature to dynamically repair oracles based on the test outputs (ChatAssert-DR). The rationale is that TECO does not support that feature. Results also show that ChatAssert-DR is superior to TECO. For example, it outperforms TECO on *Acc@10* by **5%**.

We make the following contributions:

- A novel oracle generation technique that employs prompt engineering with static and dynamic analysis to reduce the semantic gap between the LLMs and the objects under analysis (i.e., the program and the test sequence);
- A tool implementing the prompt engineering framework for oracle generation;
- A comprehensive evaluation showing the positive impact of the proposed prompt engineering method.

Overall, results provide initial yet strong evidence of the effectiveness of ChatAssert. Our artifacts are publicly available: https://github.com/ncsu-swat/chatassert.

II. Background

This section provides background information for this paper.

A. Terminology

A *large language model* (LLM), such as InCoder [24], Polycoder [25], and ChatGPT [14], is trained on massive amounts of data using general-purpose tasks such as predicting a masked token and predicting the following sentence. Users interact with an LLM through a *prompt*, which describes a specific task. For Software Engineering tasks, a prompt typically includes code and text [26]. Users of LLMs often can control the *temperature*, a variable that sets the level of (un)predictability of answers.

The *test setup method* is responsible for creating or initializing the data that will be accessed by the test. A *test prefix*

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

is the set of statements preceding the assert statement that one wishes to generate [1]. A *focal method* is a method, reachable from the test prefix, that is more likely to be the target of the generated assertion. (Prior work has shown that providing a focal method improves the performance of oracle generation techniques [1].) Different heuristics to compute focal methods exist [27]. ChatAssert uses the heuristic provided by Watson and others [1]. These three elements –the setup method, test prefix, and the focal method– are commonly used as input for test oracle generation techniques.

A *plausible* assertion is an assertion that seems reasonable, according to an objective definition (e.g., confidence scores of prediction models), but it is not guaranteed to compile or execute. An *executable* assertion is a plausible assertion that compiles and runs successfully [5]. Furthermore, it is also important to measure how often techniques can predict the oracle originally present in a test case, i.e., the ground truth. The generated oracle is an *exact match* in that case. To sum up, assertions are related as follows $Exact\ Match \subseteq Executable \subseteq Plausible$.

B. Evaluation Metrics for Oracle Generation

Oracle generation techniques, including TECO, produce multiple assertions on output. For example, TECO ranks the assertions it produces based on their compile, pass, and beam search scores associated with the model output. These techniques use a ranking score to evaluate their performance. ***Acc@k*** is a popular metric for that. It evaluates how often the ground truth appears among the top k elements in a ranked list. To compute *Acc@10*, for example, we check the presence of the ground truth oracle on each one of the first 10 reported oracles and consider that the generation task was successful if one of them is an exact match. The accuracy value is obtained by computing the fraction of cases an exact match is found.

Prior work also considers ***NLP metrics*** that measure the distance of solutions to the ground truth. These metrics complement the evaluation based on exact matches (*Acc@k*). More precisely, they account for the cases where exact matches are not produced but solutions are not "too far" [2], [5]. We consider standard NLP metrics used in prior work, namely, BLEU [28], CodeBLEU [29], ROUGE [30], and EditSim [31]. BLEU score computes the n-gram similarity between a candidate and a reference. CodeBLEU computes similarity from corresponding abstract syntax trees and data-flow structures of candidate and reference. ROUGE also uses n-gram similarity; similar to Nie et al. [5], we use the F1-score of the ROUGE-L metric for that. EditSim uses the Levenshtein distance [32].

III. ILLUSTRATIVE EXAMPLE

Fig. 2 shows one example to illustrate ChatAssert. In the interest of space, we selected a short example. Fig. 2(a) shows an example test prefix from the OrientDB [33] project, which implements a NoSQL database. Fig. 2(b) shows the dialogue between ChatAssert and ChatGPT to generate an oracle for the given test prefix. In this example, ChatAssert produces an oracle matching the one from the original code. The test prefix from Fig. 2(a) creates a document object, `doc`, and calls a method to reset a flag on that object indicating that it is in a "dirty" state.

```
@Test
public void testRemoveNotificationOne() {
  final ODocument doc = new ODocument();
  ORecordInternal.unsetDirty(doc);
  <AssertPlaceholder>;}
```

(a) Example test sequence from project OrientDB [33].

```
P1.1 I want you to generate a JUnit assertion for a test.
      For that task, consider the following related classes
      and methods:
P1.2 <include A1 and A2 from Figure 4>
P1.3 Now, given test prefix <TEST> and focal method <FOCAL
      >, generate a JUnit assertion:
     <TEST>:
     ```
     @Test
     public void testRemoveNotificationOne ( ) {
         final ODocument doc = new ODocument ( ) ;
         ORecordInternal . unsetDirty (doc) ;
         <AssertPlaceHolder>;
     }
     ```
     <FOCAL>: --ommited for space
A1. Based on the provided test prefix and the focal method,
      one junit assertion can be as follows:
     ```
     assertTrue(doc.isDirty());
     ```
P2. I am getting the following assertion error: java.lang.
     AssertionError at com.orientechnologies.orient.core.db.
     record.TrackedSetTest.testRemoveNotificationOne(
     TrackedSetTest.java:110). Can you fix the assertion?
A2. Since there is an assertion error, you can try the
     following assertion which checks if doc.isDirty() is
     false:
     ```
     assertFalse(doc.isDirty());
     ```
```

(b) Sequence of interactions between ChatAssert and ChatGPT.

Fig. 2. Illustrative example of ChatAssert's oracle generation for the input from Fig. 2(a). ChatAssert (1) summarizes the code (Fig. 4 shows the conversation); (2) formulates a task to the LLM (P1.1-1.3); and (3) runs the test with the generated assertion, observes a test failure and requests a repair to the generated assertion (P2).

Fig. 2(b) shows the conversation between ChatAssert and ChatGPT. The symbol P.$x.y$ identifies a prompt with the number x and a position within that prompt with the number y. ChatAssert starts the conversation by requesting ChatGPT to summarize a fragment of the application code related to the test (Fig. 2(a)). For this example, it provides the code for the class `ODocument` and the code for the method `unsetDirty` and asks ChatGPT to summarize them. Fig. 4 shows the dialog to obtain these summaries. (Section IV-C details the summarization task.) Code summaries add contextual and explanatory data to the prompts, which has shown to benefit LLM's performance [18], [34], [35], [36]. ChatAssert conjoins these summaries (see P1.2) and elaborates the first prompt asking ChatGPT to produce an assertion for the test method in P1.3 from Fig. 2(a). It is worth noting that the ChatGPT API does not provide sessions to save data across queries. Relevant information needs to be communicated across queries.

In response to the first prompt, ChatGPT replies with the assertion `assertTrue(doc.isDirty())`. Note that the

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

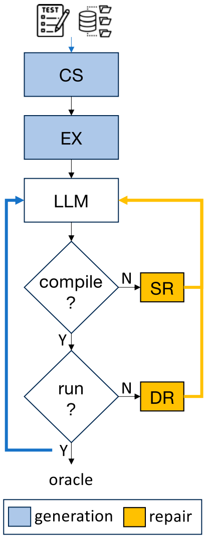


Fig. 3. CHATASSERT.

method `isDirty` was made visible to CHATGPT from the summary. CHATASSERT replaces `<AssertPlaceHolder>` in the test method with the generated assert statement and runs the test. The execution results in a test failure. CHATASSERT extracts the error message from the test output to refine the prompt and requests CHATGPT to repair that assertion (P2). CHATASSERT uses the runtime information from the output of the test run in the formulation of another prompt. This time, CHATASSERT obtains an assertion that matches with the ground truth, namely `assertFalse(doc.isDirty())`.

This example shows some important features of CHATASSERT. It shows the use of code summarization to improve oracle generation (P1.2), the use of dynamic information – associated with test runs– to repair assertions that make tests fail (P.2), and the use of feedback to improve the quality of subsequent interactions (P.2).

IV. APPROACH

CHATASSERT is an automated test oracle generation technique that uses external tools to obtain sensible information to build prompts for querying an LLM.

A. Overview

CHATASSERT takes as input the test metadata (e.g., test prefix), the program under test, and a set of configuration options, and produces a set of candidate oracles on output.

Fig. 3 lays out the organization of CHATASSERT. An execution of CHATASSERT has two stages: (1) generation and (2) repair. The generation mode is responsible for querying the LLM for new oracles whereas the repair mode is responsible for fixing an oracle that either does not compile or compiles but fails. CHATASSERT queries the LLM incrementally, refining the prompt in every round. Intuitively, CHATASSERT provides feedback to the LLM about the observations it makes during the execution. For example, CHATASSERT maintains a list of oracles that do not compile along with corresponding compiler error messages to assist the LLM in producing better answers. Querying the model in batch prevents specialized intervention. It is worth noting that CHATASSERT does *not* sort the oracles it generates. The problem of ranking oracles is orthogonal and out of scope for this paper.

1) Features: Table I shows CHATASSERT's features. Code Summarization (CS) is the feature that extracts natural language summaries describing methods and classes involved in the test prefix and adds them into the prompt as part of the context. Examples (EX) is the feature that mines similar examples from the project under test to enable few-shot learning. Dynamic Repair (DR) is the feature responsible for using the output of failing test runs (e.g., the exception raised, error message when available, etc.) to guide the LLM towards oracles that result in passing runs. Static Repair (SR) is the feature that attempts to statically repair oracles that fail to compile. We observed that the LLM often generates assertions with method calls missing the identifier of the target object. This feature uses the error message of the failing run and a simple type analysis to locate type-consistent identifiers for fixing the broken assertion.

B. The CHATASSERT Pseudocode

Algorithm 1 shows the pseudocode of CHATASSERT. It takes as input the test metadata md and a set of configuration options, including the number of candidate oracles to generate, NO. CHATASSERT produces on output a set of candidate oracles, OS. The test metadata md includes the elements considered in prior work to characterize a test object [1], [4], [5], [37]: the test setup method, if exists, and the test prefix, including the test name.

Line 11 summarizes test-related code and stores the results in the variable $summaries$. Section IV-C details function SUMMARIZE. Line 12 creates an initial prompt based on the obtained summary, the test metadata, and the focal method of the test [1], and stores the prompt on variable $prompt$.

CHATASSERT uses a prompt encoding the following data items: t(1) the task definition, (2) code summaries, (3) global history, and (4) local history. The function INITIAL_PROMPT creates a prompt object and defines the first two fields, task definition, and code summaries, which are final. As a reference, the text under P1.1–P1.3 on Fig. 2(b) shows values for these two fields. The global history field is a list that records observations that CHATASSERT makes about oracles already generated and tested and helps to guide CHATGPT towards solutions (e.g., "avoid oracle $oracle" as it has already been generated). Finally, the local history is a list storing temporary information related to a specific oracle.

The outer loop defined on lines 13–35 iteratively generates candidate oracles until reaching the target number, NO, or the global limit of queries to CHATGPT (NT). Before starting a new iteration of that loop, CHATASSERT calls a method in the prompt object to clear the local history (line 14). Information in the global history is retained across iterations of the outer loop whereas the local history is always cleared at the beginning of a local search (lines 16–35). Intuitively, the *generation mode*

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

TABLE I
DESCRIPTION OF CHATASSERT'S FEATURES

ID	Name	Description
CS	Code Summ.	ChatAssert incorporates summaries of the methods used in the test prefix into the prompt.
EX	EXamples	ChatAssert incorporates similar examples (e.g., another test from the same test file) into the prompt.
DR	Dynamic Repair	ChatAssert uses dynamic information from test runs to assist the LLM repair failing assertions. It mines the error messages from test outputs and updates the prompt accordingly (e.g., "expected x but observed y. Can you fix the assertion?").
SR	Static Repair	ChatAssert uses compilation error messages and type information to assist the LLM repair assertions that do not compile. For example, it looks for class definition that declare a method with an expected signature.

Algorithm 1. The ChatAssert algorithm.

Input: The test metadata md (e.g., test prefix), the number of candidate oracles to generate NO, the local and global maximum number of LLM trials, respectively, LT and NT, and the maximum number of repair trials RT.

Output: A set of candidate oracles OS.

```
1.  let
2.    def summarize(): ... # asks llm to summ. test-related code
3.    def initial_prompt(): ... # creates initial prompt
4.    def focal(): ... # mines focal method
5.    def extract_ora(): ... # extracts oracle from model output
6.    def rewrite_test(): ... # rewrites test case with new oracle
7.    def compile(): ... # compiles the test case
8.    def run(): ... # runs the test case
9.  in
10.   $OS$ = set()
11.   $summaries$ = summarize($md$)
12.   $prompt$ = initial_prompt($summaries$, $md$, focal($md$))
13.   while $NO > 0$ and $NT$-- $> 0$:
14.     $prompt$.clear_local_history()
15.     $lt$ = $LT$ # reset counter $lt$, focusing on one oracle
16.     while $lt$-- $> 0$:
17.       $out$ = $prompt$.query() # query the model
18.       $ora$ = extract_ora($out$) # extract oracle from text
19.       $t$ = rewrite_test($md$, $ora$)
20.       if not compile($t$):
21.         $rt$ = $RT$
22.         while $rt$-- $> 0$:
23.           $ora$ = fuzz($md$, $ora$) # e.g., add missing ids
24.           if compile($t$ = rewrite_test($md$, $ora$)) break
25.         if $rt == -1$: # $ora$ still fails to compile
26.           $prompt$.notify_compile_error("local", $ora$)
27.           continue
28.       $tresult$ = run($t$)
29.       if $tresult.success$:
30.         $NO$-- # found one! it compiles and runs
31.         $OS$.add($ora$)
32.         $prompt$.notify_successful_gen("global", $ora$)
33.         break
34.       else:
35.         $prompt$.notify_failing_ora("local", $ora$, $tresult$)
36.   return $OS$
```

of ChatAssert corresponds to the first iteration of this loop (lines 16–35). The *repair mode* corresponds to the remaining iterations, which attempt to repair the oracle produced in the first iteration. The inner loop starts by querying the model (line 17), extracting the oracle from the output (line 18), and rewriting the original test case with that oracle (line 19). The block on lines 20-27 is responsible for repairing the code that fails to compile. We empirically observed that the most common reason for that problem is missing identifiers. For example, ChatGPT generates a method call expression without indicating the target object of that call. To deal with that problem, ChatAssert (1) uses the compilation error message to find what identifier is missing, (2) uses a lightweight static analysis to find what classes, from those instantiated in the test body, declare the identifier, and (3) checks if pre-pending the expression with a matching id will make the test to compile. The loop in lines 22-24 indicates that multiple identifiers can be found in this process; the function call FUZZ abstracts the steps above. As the repair process above focuses on one case of compilation issue, we use the LLM itself as a fallback. After RT unsuccessful attempts, if ChatAssert still cannot find an oracle that makes the test compile, it adds the compilation error message in the local history (e.g., "the oracle $oracle fails to compile") and continues to another iteration of the inner loop. The local history is part of the prompt; conceptually, it makes ChatGPT avoid generating the oracle that variable ora stores again.

The test referred to by variable t at line 28 must compile if execution reaches that location. In that case, variable $tresult$ stores the results of the test run. If the test passes (line 29), ChatAssert decrements variable NO denoting the total number of assertions that remain to be generated, adds the assertion to the results set, and updates the global history to indicate that this is an executable oracle that should not be generated again and break the inner loop. If the test fails (line 35), we update the local history to indicate that the current oracle (ora) fails to run and should be avoided. The method NOTIFY_FAILING_ORA extracts the error message (e.g., "expected x found y") from the test results variable ($tresult$) and adds it to local history, which is subsequently used in the prompt of the following iteration of the inner loop.

Mapping of features. The algorithm focuses on the feedback loop of ChatAssert, as reflected through the `notify*` method calls. Feature CS maps to line 11, feature EX is encapsulated in the `initial_prompt` called at line 12, feature DR maps to line 35, and feature SR maps to lines 21–24.

C. Code Summarization (CS)

We detail the code summarization feature in the following. An LLM has no prior knowledge about the semantics of the program under analysis. Conceptually, the CS feature is responsible for obtaining an approximation of the intent of the test prefix to facilitate the oracle generation task. To that end, ChatAssert proceeds as follows for each method call or instantiation expression it encounters in the input test prefix.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

```
P1 I will ask you to explain a few methods and classes. I
   will also walk you through the steps of a Java test
   method prefix and ask you about updates to each
   variable.

A1. Yes, I will try to understand and describe the method,
    classes and variable assignments that you provide.
P2 Line 3 instantiates an object of class ODocument. Can
   you explain what that class does? Here is the code:
```
class ODocument implements ... {
  public static final byte RECORD_TYPE = 'd';...
  public ODocument() {
    setup(ODatabaseRecordThreadLocal.instance().
     getIfDefined());
  } ...
  @Override
  public Optional<OVertex> asVertex() {
    if (this instanceof OVertex)
      return Optional.of((OVertex) this);
    OClass type = this.getImmutableSchemaClass();
    if (type == null)
      return Optional.empty();
    if (type.isVertexType())
      return Optional.of(new OVertexDelegate(this));
    return Optional.empty();
  } ... <other methods>
}
```
A2. The class ODocument is some form of an associative
    database. It contains multiple flag variables like
    trackChanges, ordered, lazyLoad etc. to control how a
    document will be handled. ODocument has multiple
    overloaded constructors which enable various ways of
    instantiating a documents container...
P3 In line 4, method unsetDirty from class ORecordInternal
   is invoked. Can you explain that method? Here is the
   code:
```
public static void unsetDirty(final ORecord record) {
  final ORecordAbstract rec = (ORecordAbstract) record;
  rec.unsetDirty();
}
```
A3. The purpose of method 'unsetDirty()' is to clear or
    unset the dirty status of a record. In databases, a
    record is considered dirty if it has been modified or
    updated since it was last saved or synchronized with
    the database. By calling 'unsetDirty()', the record is
    marked as not dirty anymore. ...
```

Fig. 4. Example of code summarization.

First, ChatAssert locates the class c that declares a given function f —associated with the corresponding method or constructor— using a static type analysis based on the JavaSymbolSolver module [38] of the JavaParser [39] toolset. Second, ChatAssert considers the following cases: (1) if c is not an application class, ChatAssert indicates so in the prompt. it does not query the LLM; (2) if f is a method, ChatAssert builds a prompt with the method signature and body and asks the LLM to summarize the method; (3) if f is a constructor, ChatAssert builds a prompt with the entire class body and asks the LLM to summarize the class.

Fig. 4 illustrates the code summarization feature for the test input sequence from Fig. 2(a). ChatAssert identifies an object instantiation at line 3 (`ODocument` instantiation) and requests ChatGPT to provide a summary of the entire class. In the case of a method call expression at line 4 (`ORecordInternal.unsetDirty()`), it requests ChatGPT to summarize the method. We empirically observed that the choice between requesting a summary of class versus a summary of methods provides a good balance between space efficiency as per the number of tokens required to express method or class— and quality of results. Fig. 4 shows the conversation between ChatAssert, represented as "P", and the LLM, represented as "A". Note that each summary request is done separately, in a different query.

After all method call expressions are processed using the method above, the corresponding code summaries are conjoined and added to the prompt as part of the context. It is worth noting that such a task pipeline (i.e., using the answer of code summarization to help oracle generation) is akin to the "chain of thought" prompt design [17], [18], which is highly useful for a variety of tasks. In both cases the LLM is queried in two stages: the first stage is responsible for building background knowledge to help answer the task; the second stage is responsible for answering the task.

D. Examples (EX)

The EX feature is responsible for selecting examples and incorporating them into the prompt as context. For that, ChatAssert uses the pre-trained UniXCoder, which has shown good performance over its predecessors for code search and clone detection [40]. More precisely, we compare the cosine similarity score between the vector embeddings of the test method that we want to generate the oracle to –i.e., the target test method– and the other methods declared in the same test file. We select only test methods as examples for which the cosine similarity score was high. After checking with multiple thresholds, we find a cosine similarity threshold of > 0.6 to perform well. According to Reynolds and McDonell [34], both zero-shot and few-shot prompting outperformed one-shot prompting since one-shot prompting tends to contaminate the prompt semantics. Therefore, we only include examples if we can find more than one reference test method with cosine similarity scores above the threshold.

E. Static Repair (SR) and Dynamic Repair (DR)

The SR feature is responsible for repairing an assertion that fails to compile. ChatAssert focuses on compilation errors caused by undeclared identifiers, which we find to be the most prevalent error in oracles generated with LLMs. If the symbol that provoked the compilation error is a variable, we replace all instances of the variable with the placeholder `<insert>`. Then we carry out a fuzzing-based infilling by replacing `<insert>` with different identifiers from the test prefix. If the symbol is a method name, we add a prefix hole `<insert>`. $m(...)$ and carry out a fuzzing-based infilling with identifiers of non-primitive type that are consistent with m. ChatAssert checks whether the assert statement compiles after each infilling trial and it halts the repair process if the code compiles or after RT trials.

As recent LLM-based repair techniques [36], [41], [42], ChatAssert leverages dynamic information for assertion repair. More precisely, the DR feature is responsible for

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

repairing an assertion that successfully compiles but fails during execution. For that, CHATASSERT processes the error message obtained from the failure, i.e., from the run of the test with the invalid assertion. Then, it revises the prompt by incorporating the error message in the prompt. CHATASSERT asks CHATGPT to revise the assert statement by circumventing the failure. CHATASSERT attempts to repair an oracle for LT times (Lines 15–35 from Algorithm 1.)

It is worth noting that CHATASSERT, when configured with the DR feature, assumes that the program is bug-free, i.e., that the test has failed because of a buggy assertion and not because of a buggy code. However, note that CHATASSERT makes no observations on the state as techniques for regression oracle generation do [43], [44], so the possibility of over-constraining behavior is diminished. The rationale for CHATASSERT providing the DR feature is twofold: (1) when the developer is writing test code she certainly expects the assertion to pass and (2) the error messages from test failures are a rich source of information to guide repair.

V. EVALUATION

We pose the following research questions:

- **RQ1**: How effective is an LLM for test oracle generation?
- **RQ2**: Does using CHATASSERT's prompt engineering improve the oracle generation capabilities of LLMs?
- **RQ3**: How does CHATASSERT compare with prior work on oracle generation?
- **RQ4**: What is the impact of each of CHATASSERT's features on its performance?

The first question evaluates how promising CHATGPT—which has shown effectiveness in solving various Software Engineering tasks [12], [26], [45]— can be for the task of generating test oracles. The second question measures the impact and generality of the prompt engineering mechanism that CHATASSERT proposes. The third question compares CHATASSERT against recent prior work. The fourth question evaluates the influence of each of CHATASSERT's features on its performance. Finally, the fifth question compares CHATASSERT-DR against the SoTA technique TECO in detail.

A. Experimental Setup

We have used JavaParser [39] and JavaSymbolSolver [38] to statically analyze code when building static context for queries to the LLM. For the few-shot examples feature (EX), we have used the UniXCoder [40] model with a cosine similarity threshold of 0.6. We elaborate further on the dataset, experimental setup, and comparison baselines in the following.

1) Normalization: To facilitate the identification of exact matches, we make semantic-preserving transformations on some assertion commands under the assumption that their arguments are side-effect-free. For example, we consider `assertEquals(a,b)` and `assertEquals(b, a)` to be equivalent. We also remove the optional `message` and `delta` arguments from assert statements for similar reasons.

2) Metrics: As in prior work, we use *Acc@k* and NLP metrics for comparison (Section II-B). In addition, we also use mutation scores to evaluate the bug-finding ability of the generated oracles. Considering ranking metrics (*Acc@k*), note that we configure CHATGPT to generate k oracles. Recall that CHATASSERT does not rank oracles (Section IV-A).

3) Dataset: We evaluate CHATASSERT on a part of the evaluation set of the TECO dataset [5]. To reduce runtime and financial costs associated with running CHATGPT on various CHATASSERT configurations and projects, we run our experiments on a sample of the TECO dataset. We randomly sample 500 distinct samples from a total of 3,540 examples in the TECO dataset for a 98% confidence level and 5% error margin. There are a total of 51 projects in the test set. So, the first step is to select 10 random samples per project ($\approx$500 examples over 51 projects). For projects with less than 10 examples, we select all the examples from that project. For projects with more than 10 examples, we randomly sample 10 samples. After executing this procedure, if we cannot reach the count of 500 samples, we randomly sample from the set of remaining examples again to reach the 500 sample count. To mitigate confounding effects, throughout the sampling process, we eliminate examples that include Hamcrest's [46] `assertThat` construct and assertions that use helper methods.

4) Comparison Baselines: We group the comparison baselines according to their purpose:

LLM variants (used in RQ2). To evaluate the generality and impact of CHATASSERT's prompt engineering mechanism, we configure the oracle generation procedure with popular LLMs, namely CHATGPT [14], Mistral [21], Codestral [22], and Magicoder [23]. We chose CHATGPT because it is one of the leading large language models, particularly noted for its advanced capabilities achieved through techniques like Reinforcement Learning from Human Feedback (RLHF) which improves the model's performance in chat contexts by learning from interactions with humans [47]. We chose Mistral because it is a general-purpose LLM that has proven superior to Llama 2 13B on various benchmarks [21]. We selected Codestral because it is a larger model specifically optimized for coding tasks, demonstrating better performance on benchmarks such as HumanEval compared to Llama 3 70B, and supporting a wide range of programming languages with high accuracy [22]. Magicoder was chosen for its robust capabilities in generating high-quality instruction data which benefits from its foundation on another LLM, DeepSeek-Coder, which has proven to perform well against larger models such as CodeLlama-34B [48]. Additionally, Magicoder uses an innovative approach called OSS-Instruct, which uses open-source code references to enhance performance [23].

Prior work (used in RQ3). We compare CHATASSERT with three baseline models from prior work, namely ATLAS [1], TOGA [4], and TECO [5]. ATLAS trains a Seq2seq model [49] for assertion oracle generation. TOGA uses a grammar to generate candidate test oracles which are later ranked using a fine-tuned CodeBERT model [4], [50]. The TECO model is obtained by fine-tuning a CODE-T5 [37] model on both syntax level data and runtime semantics.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

```
Given the setup code <SETUP>, test prefix <TEST>,
and focal method <FOCAL>, generate one org.junit.
Assert statement:
<SETUP>:...\n<TEST>:...\n<FOCAL>:...
```

Fig. 5. ChatGPT-one prompt template.

TABLE II
Comparison of TECO and ChatGPT-one. Higher Values Indicate Better Performance

Model	**Acc@1**	**BLEU**	**CodeBLEU**	**ROUGE**	**EditSim**
TECO	0.09	0.85	0.40	0.72	0.55
ChatGPT-one	0.10	0.77	0.24	0.36	0.46

B. Answering RQ1: How Effective Is an LLM for Test Oracle Generation?

To answer this question, we select ChatGPT platform's popular GPT3.5 model as the representative LLM because it offers a convenient API to quickly and easily set up feasibility experimentation. We evaluate whether ChatGPT, configured with a simplistic prompt, can generate oracles with accuracy that is comparable to that of the SoTA.

This research question compares the state-of-the-art technique TECO with ChatGPT-one, a technique obtained by using ChatGPT with a simplistic prompt, joining four elements: (1) a natural language description of the task, (2) the test setup code, (3) the test prefix code, and (4) the name of the focal method. Fig. 5 shows the prompt template that ChatGPT-one uses. ChatGPT-one is powered by the GPT3.5 LLM and queries the model using the default temperature. The "one" in the name ChatGPT-one signifies that ChatGPT-one asks ChatGPT for only one assert statement for each test method.

We run TECO and ChatGPT-one on the TECO dataset, including 3,540 tests, and evaluate their performance using the metrics listed in Section V-A1. We use $k{=}1$ for $Acc@k$ as ChatGPT-one is asked to generate only one assert statement for a given test method. Table II shows results comparing TECO and ChatGPT-one.

Summary: *Despite the absolute low marks of both techniques in* $\mathrm{Acc@1}$ *and the higher marks of the SoTA in most NLP metrics, results show that* ChatGPT-one *is competitive with the SoTA considering* $\mathrm{Acc@1}$ *(1ppt difference). We conclude that LLMs show promise for improving neural-based oracle generation techniques.*

C. Answering RQ2: Does Using ChatAssert's Prompt Engineering Improve the Oracle Generation Capabilities of LLMs?

The purpose of this question is to validate whether the prompt engineering mechanism of ChatAssert improves the performance of LLMs for oracle generation.

Table III shows the results of all baselines and metrics. We consider four LLMs as our baselines; one closed-source (ChatGPT) and three open-source (Mistral, Magicoder, and Codestral). These baselines also use a simplistic prompt similar to ChatGPT-one (Section V-B) with the exception of asking for TEN assert statements instead of ONE for each test method. We evaluated these baselines on 500 randomly-selected sample (Section V-A3) from the TECO evaluation dataset. In contrast, ChatGPT-one was evaluated on the entire 3,540 samples from TECO's evaluation dataset. This justifies the 0.05ppt performance difference between ChatGPT-one and ChatGPT at *Acc@1*. It came as a surprise that Codestral, an open-source model, performed significantly better than ChatGPT in *Acc@1*, *Acc@3* and *Acc@5* and achieved a similar score to ChatGPT's in *Acc@10*. It is worth noting that recent work [22], [51] reports similarly impressive results showing that Codestral achieves a HumanEval score of 81% compared to 72% of ChatGPT [52]. The techniques below the dashed line correspond to the variants of ChatAssert, configured with the baseline LLMs that appear above the dashed line.

The results indicate that ChatAssert consistently improves the performance of all four LLMs for test oracle generation. For instance, ChatAssert-ChatGPT achieves an Acc@1 of 0.45 whereas ChatGPT only reaches 0.05. ChatAssert-ChatGPT consistently demonstrates better performance in higher-ranked accuracies (Acc@3, Acc@5, Acc@10), showing that the prompt engineering features of ChatAssert effectively enhance the LLM's ability to generate exact match oracles. When comparing *Acc@10* between ChatGPT and ChatAssert-ChatGPT, the Chi-Square value is 66.76 with a p-value of 3.04×10^{-16}, indicating a statistically significant improvement of ChatAssert-ChatGPT over ChatGPT with a large effect size (Cohen's h 1.3). Likewise, the impact of ChatAssert on the performance of open-source LLMs is also significant. For instance, compared to Mistral, ChatAssert-Mistral shows an average increase in accuracy of 10%.

Despite the impressive performance of Codestral to generate oracles with simplistic prompts compared to ChatGPT, ChatAssert-ChatGPT outperformed ChatAssert-Codestral in more than 10ppt across all accuracy levels. We have observed that ChatGPT responds very well to the summarization (CS) and dynamic repair (DR) features of the prompt compared to Codestral which explains why ChatAssert-ChatGPT outperforms ChatAssert-Codestral.

Summary: *Results indicate that* ChatAssert*'s prompt engineering significantly improves accuracy and NLP metrics for all tested LLMs.*

D. Answering RQ3: How Does ChatAssert Compare With Prior Work on Oracle Generation?

This research question evaluates the performance of ChatAssert in comparison to prior work. For this research question, we consider the version of ChatAssert that performed best in RQ2, i.e., the one configured with ChatGPT. As comparison baselines, we use the techniques listed in Section V-A4. Table IV shows the results of the techniques for various accuracy and NLP metrics. Rows above the dashed line correspond to the baseline techniques. The last row shows the results of ChatAssert (same as in ChatAssert-ChatGPT from Table III).

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

TABLE III
COMPARISON OF CHATGPT AND CHATASSERT. HIGHER VALUES INDICATE BETTER PERFORMANCE

Model	Acc@1	Acc@3	Acc@5	Acc@10	BLEU	CodeBLEU	ROUGE	EditSim
CHATGPT [14]	0.05	0.12	0.18	0.28	0.79	0.31	0.39	0.48
MISTRAL-7B [21]	0.18	0.21	0.22	0.22	0.63	0.29	0.51	0.54
MAGICODER-6.7B [23]	0.06	0.07	0.07	0.07	0.50	0.21	0.44	0.45
CODESTRAL-22B [22]	0.22	0.27	0.28	0.28	0.67	0.30	0.55	0.55
CHATASSERT-CHATGPT	0.45	0.51	0.53	0.54	0.85	0.35	0.52	0.55
CHATASSERT-MISTRAL	0.30	0.32	0.32	0.32	0.70	0.32	0.57	0.57
CHATASSERT-MAGICODER	0.23	0.25	0.25	0.25	0.66	0.32	0.54	0.57
CHATASSERT-CODESTRAL	0.33	0.35	0.35	0.35	0.74	0.38	0.62	0.62

TABLE IV
COMPARISON OF CHATASSERT AGAINST THE BASELINES ATLAS, TOGA, AND TECO. HIGHER VALUES INDICATE BETTER PERFORMANCE

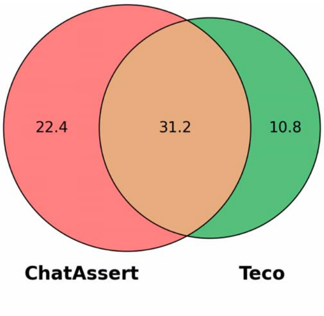
Model	Acc@1	Acc@3	Acc@5	Acc@10	BLEU	CodeBLEU	ROUGE	EditSim
ATLAS [1]	0.00	0.00	0.001	0.001	0.25	0.12	0.34	0.28
TOGA [4]	0.09	0.09	0.14	0.15	0.33	0.27	0.39	0.41
TECO [5]	0.30	0.38	0.41	0.42	0.81	0.31	0.47	0.55
CHATASSERT	0.45	0.51	0.53	0.54	0.85	0.35	0.52	0.55

Overall, results show that the *Acc@k* and NLP scores of CHATASSERT are consistently higher compared to prior work. It is worth noting that ATLAS exhibits a notably low accuracy score. This can be attributed to ATLAS being an earlier model that uses an RNN encoder-decoder, which may not capture the same level of complexity as more recent models [1]. TOGA improves over ATLAS because of its grammar-based oracle generation but still does poorly because of the restrictive nature of its grammar and its requirement of an assertion approximation [4], [53]. Hossain et al. similarly reports that TOGA was able to generate an assertion in only 32% of the cases and among the assertions it generated, more than 50% were inaccurate [54].

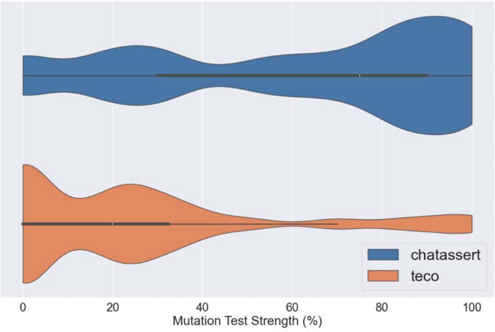
TECO on the other hand improves over TOGA by including six types of code semantics with their neural model to generate assert statements. However, because of this restrictive set of six semantics TECO fails to generate the accurate assert statement in many cases [5]. CHATASSERT outperforms all the prior work by using more inclusive code summaries and examples to generate an oracle and then statically and dynamically repairs the generated oracles with the help of CHATGPT.

Given the relative strength of TECO in comparison to ATLAS and TOGA, we decided to conduct a detailed comparison between CHATASSERT and TECO considering (1) exact matches and (2) non-exact matches. For exact matches, we measure the number of exact matches that each technique generates, analyzing the overlap and exclusivity between the oracles generated by both methods. For non-exact matches, we measure the ability of generated oracles to kill mutants through mutation analysis.

Considering the *exact matches*, the Venn diagram from Fig. 6(a) shows the fraction of the ground truth that each technique generates. The diagram shows that no technique subsumes the other, that CHATASSERT covers 22.4% of the oracles distinctly whereas TECO covers 10.8% of the oracles distinctly, and that the combination of the techniques covers 64.4% of the ground truth. When comparing *Acc@10* of CHATASSERT and TECO, the Chi-Square value is 13.02 with a p-value of 0.0003, indicating a statistically significant difference with a medium effect size (Cohen's h 0.58).



(a) Venn diagram showing differences and commonalities of exact match oracles generated by CHATASSERT and TECO. The numbers inside the partitions represent fractions of the ground truth.



(b) Violin plot showing mutation strength of non-exact match oracles generated by CHATASSERT and TECO.

Fig. 6. Comparison between CHATASSERT and TECO on oracles that are exact matches Fig. 6(a) and oracles that are non-exact matches Fig. 6(b).

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

TABLE V
Comparison of ChatAssert-DR Against TECO (the SoTA). Higher Values Indicate Better Performance

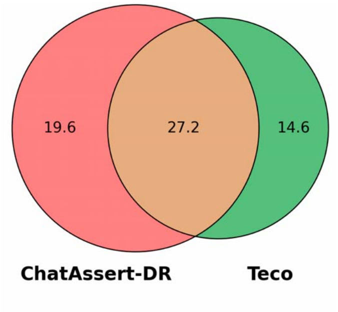
Model	Acc@1	Acc@3	Acc@5	Acc@10	BLEU	CodeBLEU	ROUGE	EditSim
TECO [5]	0.30	0.38	0.41	0.42	0.81	0.31	0.47	0.55
ChatAssert - DR	0.35	0.41	0.44	0.47	0.81	0.28	0.47	0.49

It is worth noting that a generated oracle can be useful even if it is *not an exact match*. Although the NLP metrics provide a rough estimate for those cases, (1) they are inaccurate [55] and (2) TECO and ChatAssert show similar results for those metrics. As a proxy to measure the bug detection capabilities of non-exact match oracles and hence their quality, we carry out a mutation analysis to find out how many mutants are killed by these oracles. We use the PIT Mutation Test tool [56] for that. We used the default mutation operators from the tool. Because mutation testing is expensive, we configured PIT to generate a maximum of 21 mutations per class when evaluating both TECO and ChatAssert. We used a timeout factor of 1 which helps to overcome infinite loops. Other than that, we have not used any time constraints. Fig. 6(b) shows the distribution of mutation strength associated with the oracles that ChatAssert and TECO generate. Results show that the mutation strength of the tests augmented with ChatAssert's oracles is significantly higher (p-value < 0.05) than those augmented with TECO's oracles, with a medium effect size (Cliff's Delta 0.51).

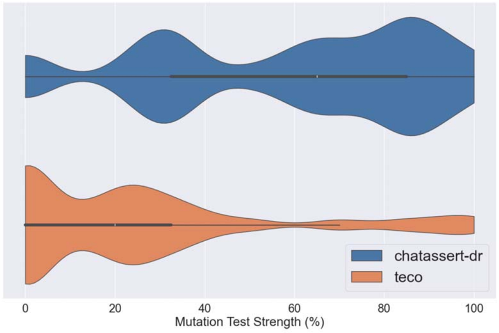
Comparing TECO with ChatAssert-DR. This research question evaluates how TECO compares against ChatAssert without dynamic repair, i.e., ChatAssert-DR. The rationale is that TECO does not dynamically repair test oracles.

Table V compares the various accuracy and NLP metrics between ChatAssert-DR and TECO. Considering *Acc@k*, the numbers show a consistent improvement of ChatAssert-DR for all values of k. For example, the *Acc@1* of ChatAssert-DR is **5%** higher compared to that of TECO. The difference decreases for higher values of k, but the accuracy of ChatAssert-DR is still 5% higher than that of TECO at $k = 10$ (47% and 42%, respectively). Notably, TECO was able to produce slightly more syntactically similar oracles compared to ChatAssert-DR based on the NLP metrics. Curiously, we found that NLP metrics can be unreliable when measuring a technique's ability to generate effective oracles [55]. We also carry out an in-depth comparison between ChatAssert-DR and TECO on exact and non-exact matches.

Considering exact matches, the Venn diagram from Fig. 7(a) shows the fraction of the ground truth that each technique generates. Since TECO generates 10 assertions per test sample, we retrieve 10 assertions with ChatAssert-DR, as for measuring *Acc@10*. The diagram shows that no technique subsumes the other, that ChatAssert-DR discovers 19.6% of the oracles distinctly while TECO discovers 14.6% of the oracles distinctly. Combined, the techniques generate 61.4% of the oracles from the ground truth set. Considering non-exact matches, Fig. 7(b) shows the distribution of mutation strength of the oracles generated by the techniques. Notice that the strength associated with ChatAssert-DR is mostly concentrated between 70-100% compared to 0-20% for the oracles that TECO generates. We conjecture that ChatAssert-DR produces more effective oracles compared to TECO because ChatAssert-DR kills significantly more mutants (p-value < 0.05) compared to TECO with a small effect size (Cliff's Delta 0.38).



(a) Venn diagram showing differences and commonalities of exact match oracles generated by ChatAssert-DR and TECO. The numbers inside the partitions represent fractions of the ground truth.



(b) Violin plot showing mutation strength of non-exact match oracles generated by ChatAssert-DR and TECO.

Fig. 7. Comparison between ChatAssert-DR and TECO on oracles that are exact matches Fig. 7(a) and oracles that are non-exact matches Fig. 7(b).

Summary: ChatAssert *outperforms prior techniques. Compared to the SoTA technique*, TECO, ChatAssert *achieves 15% higher and* ChatAssert-DR *achieves 5% higher Acc@1 scores respectively. Additionally, oracles generated by both* ChatAssert *and* ChatAssert-DR *demonstrate significantly higher mutation strength than* TECO.

E. Answering RQ4: What Is the Impact of Each of ChatAssert's Features on Its Performance?

The purpose of this research question is to evaluate the contribution of each of the features from Table I on ChatAssert's performance.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

TABLE VI
ABLATION STUDY. CHATASSERT - x REPRESENTS THE VERSION OF CHATASSERT WITHOUT FEATURE x. FOR ROWS ABOVE THE DASHED LINE, THE LOWER THE VALUE, THE HIGHER THE IMPACT OF THE CORRESPONDING FEATURE. FOR THE ROW BELOW DASHED LINE, HIGHER VALUES INDICATE BETTER PERFORMANCE

Model	**Acc@1**	**Acc@3**	**Acc@5**	**Acc@10**	**BLEU**	**CodeBLEU**	**ROUGE**	**EditSim**
CHATASSERT - EX	0.37	0.45	0.47	0.48	0.83	0.32	0.50	0.53
CHATASSERT - CS	0.34	0.43	0.45	0.47	0.82	0.30	0.47	0.50
CHATASSERT - SR	0.36	0.42	0.44	0.47	0.82	0.29	0.48	0.50
CHATASSERT - DR	0.35	0.41	0.44	0.47	0.81	0.28	0.47	0.49
CHATASSERT	0.45	0.51	0.53	0.54	0.85	0.35	0.52	0.55

Table VI reports the results of this ablation study, where we run CHATASSERT with each of its four features removed. (CHATASSERT's implementation provides feature flags for that.) Rows CHATASSERT-x show these configurations, where x refers to the abbreviation of the feature as appearing on column "ID" from Table I. For reference, the last row shows the results of CHATASSERT with all features enabled. The table lists these "remove-one" configurations in decreasing order of *Acc@10*. We chose to sort by *Acc@10* because it is more inclusive compared to the other ranking levels.

The results show that the impact of feature EX was slightly less than the other features of CHATASSERT. Notably CHATASSERT found few shot examples for only 45% of the samples. Disabling that feature reduces *Acc@10* by 6%. We observe that the CS, SR, and DR features had similar and relatively higher impacts, with their removal resulting in an average 7% loss in *Acc@10*. A similar drop in accuracy for these different features suggests that multiple features contribute to the generation of a common set of accurate oracles. Considering *Acc@1*, we found the Code Summarization (CS) feature to be the most impactful with a drop of 11% points when the feature CS is not used. Removing any of the features results in a negative impact, indicating that all features contribute positively to CHATASSERT's overall performance.

Summary: *Results indicate that all features of* CHATASSERT *are relevant and that one accurate oracle can be generated with the help of multiple prompting features of* CHATASSERT.

VI. DISCUSSION

This section discusses positive and negative examples for CHATASSERT, lessons and implications, and threats to validity.

A. Examples

The Venn diagram from Fig. 6(a) shows that important sets of assertions are generated exclusively by CHATASSERT or TECO. This section discusses a small selection of cases where we find CHATASSERT to perform particularly well when compared to TECO (positive cases) and cases where the contrary happens (negative cases).

Fig. 8 shows two positive test cases, i.e., cases CHATASSERT was able to generate the oracle matching the ground truth (exact match), but TECO was unable to generate the oracle. In the first example test case, `toLinkedHashSet`, we find that the file containing the test method also contains two other similar test methods which CHATASSERT uses as few shot examples (EX feature). CHATASSERT had initially left out the identifier `Collectors` in the assert statement. With the help of the static repair feature (SR), CHATASSERT was able to prepend `Collectors` to the method call `toLinkedHashSet()`. The combinations CHATASSERT- CS and CHATASSERT- DR, which incorporate both features EX and SR, were able to produce the expected oracle. In the second example test case, `testFlatten`, we observe that the code summarization and the dynamic repair features were essential to producing the oracle. When we analyzed the data from the ablation study, we observed that CHATASSERT could not predict the expression `flattened.size()` when we removed the code summarization feature (CS). Likewise, we observe that CHATASSERT was unable to predict the correct value of the expression `flattened.size()` (i.e., the value 5) when we removed the dynamic repair feature (DR). The combinations CHATASSERT- EX and CHATASSERT- SR, which incorporate both features CS and DR, were able to produce the expected oracle.

```
/* From project: gvlasov/collections-utils  [57] */
@Test public void toLinkedHashSet() {
 final LinkedHashSet<Integer> expected = new
     LinkedHashSet<>();
 expected.add(1); expected.add(2); expected.add(3);
 Assert.assertEquals(expected, Arrays.asList(1, 2, 3).
     stream()
   .collect(Collectors.toLinkedHashSet())); /* <== */ }

/* From project: arnohaase/a-foundation  [58] */
@Test public void testFlatten() {
 final Set<Set<String>> set = new HashSet<>();
 set.add(new HashSet<>(Arrays.asList("a", "b")));
 set.add(new HashSet<>(Arrays.asList("b", "c", "d")));
 final Collection<String> flattened
   = ACollectionHelper.flatten(set);
 assertEquals(5, flattened.size()); /* <== */ }
```

Fig. 8. Positive examples: only CHATASSERT succeeded in generating the ground truth assertion.

Fig. 9(a) highlights two negative examples, i.e., cases where CHATASSERT failed to generate oracles but, TECO can generate the oracles. For the case of `testGroupByCustomEquality`, when the test prefix contains anonymous inner classes, CHATASSERT is unable to predict the correct assertion. For the test `testIntObjectMap`, CHATASSERT can generate `assertEquals(11,(int)test.get(24));` which is executable and similar to but it is not an exact match of the ground truth oracle. Note this is not a limitation of the technique, it

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

```java
/* From project: arnohaase/a-foundation  [58] */
@Test public void testGroupByCustomEquality() {
 final AEquality equality = new AEquality() {
  @Override public boolean equals(...){...}
  @Override public int hashCode(...) {...} };
 final AFunction1NoThrow<String, Integer> len
  = new AFunction1NoThrow<String, Integer>() {
    @Override public Integer apply(String param) {...} };
 final Map<AEqualsWrapper<Integer>,List<String>> grouped=
   ACollectionHelper.groupBy(
    Arrays.asList("a", "bc", "d", "efg", "hi", "j"),
     len, equality);
  assertEquals(2, grouped.size()); /* <== */ }

/* From project: jcodec/jcodec  [59] */
@Test public void testIntObjectMap() {
 IntObjectMap<Integer> test= new IntObjectMap<Integer>();
 test.put(24, 11);
 assertEquals(Integer.valueOf(11), test.get(24));/*<==*/}
```

(a) ChatAssert failed to generate the ground truth assertion.

```java
/* From project: arnohaase/a-foundation  [58] */
@Test public void testFilter() {
 final APredicateNoThrow<String> len1
   = new APredicateNoThrow<String>() {
     @Override  public boolean apply(String o) {...} };
 assertEquals(Arrays.<String>asList(),
  ACollectionHelper.filter(
   Arrays.<String>asList(), len1)); /*<==*/ }

/* From project: arnohaase/a-foundation  [58] */
@Test public void testMkStringFull() {
 assertEquals("[]", ACollectionHelper.mkString(
   Arrays.asList(),"[","#","]")); /* <== */ }
```

(b) Both ChatAssert and TECO failed to generate the ground truth assertion.

Fig. 9. Negative examples.

is an inherent limitation of the evaluation metrics used in the literature.

Fig. 9(b) shows examples where neither ChatAssert nor TECO can generate the correct assertion. For the test `testFilter`, we observe the presence of the anonymous inner class (a subtype of `APredicateNoThrow<String>`) in the test prefix, suggesting that neural models of TECO or ChatGPT struggle with this kind of programming construct. For the test `testMkStringFull`, we observe that the test prefix does *not* contain any statements that could help in the process of generating the correct oracle.

B. Lessons and Implications

1) Complementary Nature of Approaches: Fig. 6(a) shows that TECO and ChatAssert complement each other. Indeed, the union of both techniques would have an *Acc@10* score of 64%, which is remarkably higher than any previous oracle generator [5]. This observation calls for work that combines multiple oracle generation techniques, ranking their combined results or using their inputs as seeds for fuzzing [7].

2) Iterative Feedback Is Important But Costly: We observe that iterative feedback has a progressively increasing positive impact on ChatAssert's performance (Section V-D) at higher values of k (Table IV). ChatAssert incorporates sensible information throughout its execution to dynamically build context to ChatGPT. This is reflected in the `notify*` methods on the prompt object from Algorithm 1. Making multiple calls to ChatGPT is relatively expensive. For instance, currently, it takes ~10m, on average, for ChatAssert to generate an assertion. One of the reasons for such high costs is the instability of the API service. For example, ChatGPT users found that accessing the service through the API is often slower when compared to accessing the service through the web interface. One of the users recently reported that using a new user identifier as part of the request body can result in significant speedups during API access [60]. Another reason for the high cost relates to the high usage of a closed-source LLM (in our case, ChatGPT). These results call for action to offload subtasks to local servers running open-source LLMs. For example, it is possible to offload the repair task to a local infill LLM [61]. More precisely, the repair task would replace some identifiers in the oracles that ChatGPT produces with masks and request an infill LLM (e.g., Incoder [24]) to replace those masks with alternative identifiers. Considering those aspects, currently, the complete version of ChatAssert is best used asynchronously, to generate a batch of oracles for multiple test methods. Future developments in open-source LLMs may enable synchronous usage. It is worth noting that ChatAssert-DR may be used synchronously. It is significantly faster compared to the complete version of ChatAssert.

3) Future Enhancements: Based on the negative examples of ChatAssert shown in Section VI-A, we note the following weaknesses of ChatAssert: **(a) Anonymous inner classes.** When the context contains anonymous inner classes, ChatAssert is unable to generate an exact match oracle. **(b) Anonymous arrays.** ChatAssert is unable to generate anonymous arrays as the arguments of the oracles. **(c) Concretizing `STR` token.** Similar to TECO, ChatAssert replaces string literals with the `STR` token which prevents us from evaluating the ability of these techniques to generate correct string literals. **(d) Deduplicating equivalent oracles.** ChatAssert does not deduplicate the semantically equivalent oracles which can reduce the diversity among the generated oracles. Future enhancements can address these limitations of ChatAssert and optimize time to generate oracles.

C. Threats to Validity

One threat to construct validity relates to the choice of metrics we used to evaluate the techniques. As in prior works, to deal with that threat, we use two different kinds of metrics to show the different perspectives about the results. One threat to internal validity relates to mistakes that we might have made during the implementation. To mitigate that, we carefully inspected the code and walked through executions. One threat to external validity is the choice of datasets and the sampling criterion we used to reduce the cost to an acceptable level. We based our experiments on the TECO dataset, which is publicly available. Considering sampling, we determined the sampling size to produce results that are within a 95% confidence interval (of representing the actual result) and used stratified sampling to ensure that all projects are uniformly represented in the sample. Another threat to external validity is the possibility of our test cases appearing in the training dataset of ChatGPT. Since ChatGPT is a closed model and the training data are not available to end users of ChatGPT, it is difficult to determine whether a test case belongs to the training dataset of ChatGPT. To

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

mitigate this threat, we check whether ChatGPT can generate an exact match oracle at the first attempt, by reviewing the Acc@1 performance of the model CHATGPT-ONE from Section V-B. Since the Acc@1 score of CHATGPT-ONE is very low (0.10), we say that it is unlikely that ChatGPT has seen most of our test cases in its training dataset.

VII. RELATED WORK

This section elaborates on work most related to ours.

Oracle Generation. Several approaches have been proposed in the literature to address the oracle generation problem. We briefly discuss a subset of these approaches that do *not* rely on machine learning. Randoop generates assert statements based on user-provided contracts and uses execution feedback to guide (test and) regression oracle generation [62], [63]. EvoSuite performs random mutations in the application code and generates assert statements to maximize the number of killed mutants [64]. Approaches that use Natural Language Processing [65], [66], [67], [68], [69], [70] and Grammar-based fuzzing have also been explored [71] to mine likely oracles.

More recently, deep learning-based assert statement generation has been mentioned to generate high-quality assert statements [1], [4]. **ATLAS** [1] trains a Seq2seq model [49] for the assertion generation task and uses a beam search decoder on that model to obtain assertions for a corresponding input, i.e., a pair of test prefix and focal method. Mastropaolo et al. [2] pretrains a **CODE-T5** model [37] on a subset of the CodeSearchNet dataset [72] and fine-tunes on the ATLAS dataset [1] for the assertion generation task. **TOGA** [4] uses a grammar to generate assertions and a fine-tuned CodeBERT model [50] to predict assertion likelihood. **TECO** [5] is an encoder-decoder transformer model that is fine-tuned on different code semantics data, the test prefix, and the method under test. **SAGA** [73] is a deep learning model that is trained with the test prefix, focal method, and a basic focal method summary for the task of assertion generation. However, we perform an extensive program analysis of the test prefix to extract method calls, object instantiations, and variable assignments which we ask ChatGPT to automatically summarize. Our technique utilizes the conversational ability of the pre-trained large language model ChatGPT along with few-shot prompting, and static and dynamic repair to achieve superior performance.

Automated Code Repair. Neural Machine Translation (NMT)-based architectures have recently been successfully adopted for Automated Code Repair tasks [74], [75], [76], [77]. In [74], authors use a perturbation step to generate training samples for program repair and then train a transformer neural network to generate the repaired program. In [75], [76], authors train NMT models with a focus on automatic program repair. More recently, Xia and Zhang proposed ChatRepair to repair code using execution and conversation using ChatGPT [45]. We also use ChatGPT's conversational ability, e.g., to ask for a repair of a failing assertion. It is worth noting that CHATASSERT tries to fix compilation errors due to missing identifiers without the assistance of ChatGPT.

Large Language Models. Deep learning has emerged as an alternative way of creating software engineering techniques [20]. Earlier work often trains task-specific models in a supervised manner, e.g., for bug detection [78], [79], [80], type prediction [81], [82], [83], [84], program repair [77], [85], [86], [87], and code completion [88], [89], [90]. More recent work builds on pre-trained models, such as CodeBERT [50], GraphCodeBERT [91], and PLBART [92], by fine-tuning these models for specific tasks, e.g., code completion [93], code editing [94], program repair [95], and pruning call graphs [96]. Even more recent work builds on general-purpose, large language models (LLMs) [97], e.g., Codex [98], InCoder [24], and PolyCoder [99]. Once trained on huge datasets, an LLM can be queried with few-shot prompts, i.e., by providing a small number of task-specific input-output examples to the model. Recent work shows the potential of LLMs to support downstream analyses [5], [6], [7], [8], [9], [10], [11], [12], [13]. The novel aspect of our work is the idea of iteratively querying the LLM for oracle generation.

VIII. CONCLUSION

Oracle generation is an important and challenging problem. Recent neural-based techniques have been proposed to address the problem, but they are inaccurate. Improving the accuracy of oracle generation techniques is imperative to make them more practical. We propose CHATASSERT, a feedback-driven oracle generation technique that iteratively incorporates dynamic and static information in prompts to query a large language model. Results show that CHATASSERT significantly outperforms the baseline techniques. For example, compared to the SoTA technique TECO, CHATASSERT improves *Acc@1* by 15%. Additionally, results confirm that tests augmented with CHATASSERT-generated oracles have much higher mutation test strength and therefore higher bug-finding effectiveness. An ablation study shows that all four components of CHATASSERT are relevant, with examples being slightly less relevant. Our results provide initial evidence of the promising performance of CHATASSERT and call for further work that incorporates external tools to cooperate with LLMs to solve Software Engineering tasks.

ACKNOWLEDGMENT

The authors thank the reviewers and editors for their insightful feedback.

REFERENCES

[1] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proc. Int. Conf. Softw. Eng.*, 2020, pp. 1398–1409.

[2] A. Mastropaolo et al., "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *Proc. Int. Conf. Softw. Eng.*, 2021, pp. 336–347.

[3] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, "Perfect is the enemy of test oracle," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 70–81.

[4] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A neural method for test oracle generation," in *Proc. Int. Conf. Softw. Eng.*, 2022, pp. 2130–2141.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

[5] P. Nie, R. Banerjee, J. J. Li, R. J. Mooney, and M. Gligoric, “Learning deep semantics for test completion,” in *Proc. Int. Conf. Softw. Eng.*, 2023, pp. 2111–2123.

[6] H. Joshi, J. Cambronero Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček, “Repair is nearly generation: Multilingual program repair with llms,” in *Proc. AAAI Conf. on Artif. Intell.*, Jun. 2023, vol. 37, no. 4, pp. 5131–5140.

[7] C. S. Xia and L. Zhang, “Less training, more repairing please: Revisiting automated program repair via zero-shot learning,” in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, A. Roychoudhury, C. Cadar, and M. Kim, Eds., 2022, pp. 959–971.

[8] J. Zhang et al., “PyDex: Repairing bugs in introductory Python assignments using LLMs,” *ACM Program. Lang.*, vol. 8, no. OOPSLA1, pp. 1100–1124, Apr 2024.

[9] N. Jiang, K. Liu, T. Lutellier, and L. Tan, “Impact of code language models on automated program repair,” in *Proc. Int. Conf. Softw. Eng.*, 2023, pp. 1430–1442.

[10] G. Poesia et al., “Synchromesh: Reliable code generation from pre-trained language models,” in *Proc. Int. Conf. Learn. Representations*, 2022. Accessed: Jan. 03, 2024. [Online]. Available: https://openreview.net/forum?id=KmtVD97J43e

[11] S. Kang, J. Yoon, and S. Yoo, “Large language models are few-shot testers: Exploring LLM-based general bug reproduction,” in *Proc. Int. Conf. Softw. Eng.*, 2023, pp. 2312–2323.

[12] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models,” in *Proc. Int. Conf. Softw. Eng.*, 2023, pp. 919–931.

[13] A. Kharkar et al., “Learning to reduce false positives in analytic bug detectors,” in *Proc. Int. Conf. Softw. Eng.*, 2022, pp. 1307–1316.

[14] “ChatGPT,” OpenAI. Accessed: Apr. 15, 2023. [Online]. Available: https://chat.openai.com

[15] GitHub, “NanoHttpd/nanohttpd: Tiny, easily embeddable HTTP server in Java,” github.com. NanoHttpd. Accessed: Jun. 21, 2024. [Online]. Available: https://github.com/NanoHttpd/nanohttpd

[16] “OpenAI,” OpenAI API. Accessed: May 19, 2023. [Online]. Available: https://platform.openai.com/docs/api-reference

[17] J. Wei et al., “Chain-of-thought prompting elicits reasoning in large language models,” in *Proc. Neural Inf. Process. Syst.*, 2022, pp. 24824–24837.

[18] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” in *Proc. Neural Inf. Process. Syst.*, 2022, pp. 22199–22213.

[19] S. Yao et al., “ReAct: Synergizing reasoning and acting in language models,” in *Proc. Int. Conf. Learn. Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X

[20] M. Pradel and S. Chandra, “Neural software analysis,” *Commun. ACM*, vol. 65, no. 1, pp. 86–96, Dec 2021.

[21] A. Q. Jiang et al., “Mistral 7B,” 2023, *arXiv:2310.06825*.

[22] “Codestral: Hello, World!—” mistral.ai. Mistral AI. Accessed: Jun. 24, 2024. [Online]. Available: https://mistral.ai/news/codestral/

[23] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, “Magicoder: Empowering code generation with OSS-instruct,” in *Proc. Int. Conf. Mach. Learn.*, vol. 235, Jul. 2024, pp. 52632–52657.

[24] D. Fried et al., “InCoder: A generative model for code infilling and synthesis,” in *Proc. Int. Conf. Learn. Representations*, 2023. Accessed: Jun. 11, 2023. [Online]. Available: https://openreview.net/forum?id=hQwb-lbM6EL

[25] V. Garousi, A. Mesbah, A. Betin-Can, and S. Mirshokraie, “A systematic mapping study of web application testing,” *Inf. Software Technol.*, vol. 55, no. 8, pp. 1374–1396, 2013.

[26] P. Bareiß, B. Souza, M. d’Amorim, and M. Pradel, “Code generation tools (almost) for free? A study of few-shot, pre-trained language models on code,” 2022, *arXiv:2206.01335*.

[27] Y. He, J. Huang, H. Yu, and T. Xie, “An empirical study on focal methods in deep-learning-based approaches for assertion generation,” in *Proc. Int. Conf. Found. Softw. Eng.*, 2024, pp. 1750–1771.

[28] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “BLEU: A method for automatic evaluation of machine translation,” in *Proc. Assoc. Comput. Linguistics*, 2002, pp. 311–318.

[29] S. Ren et al., “CodeBLEU: A method for automatic evaluation of code synthesis,” 2020, *arXiv:2009.10297*.

[30] C.-Y. Lin, “ROUGE: A package for automatic evaluation of summaries,” in *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, 2004, pp. 74–81.

[31] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “IntelliCode compose: Code generation using transformer,” in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1433–1443.

[32] G. Navarro, “A guided tour to approximate string matching,” *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[33] “Orient technologies,” OrientDB. Accessed: Jul. 7, 2023. [Online]. Available: https://github.com/orienttechnologies/orientdb

[34] L. Reynolds and K. McDonell, “Prompt programming for large language models: Beyond the few-shot paradigm,” in *Proc. Extended Abstr. CHI Conf. Human Factors Comput. Syst.*, 2021, pp. 1–7.

[35] J. Huang and K. C.-C. Chang, “Towards reasoning in large language models: A survey,” in *Proc. Assoc. Comput. Linguistics*, Jul. 2023, pp. 1049–1065.

[36] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, “Examining zero-shot vulnerability repair with large language models,” in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 1–18.

[37] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2021, pp. 8696–8708.

[38] “JavaParser Symbol Solver,” JavaParser. Accessed: Jul. 7, 2023. [Online]. Available: https://javadoc.io/doc/com.github.javaparser/java-symbol-solver-core/

[39] “JavaParser tools,” JavaParser. Accessed: Jul. 7, 2023. [Online]. Available: https://javaparser.org

[40] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “UniXcoder: Unified cross-modal pre-training for code representation,” in *Proc. Assoc. Comput. Linguistics*, 2022, pp. 7212–7225.

[41] E. Pinconschi, R. Abreu, and P. Adão, “A comparative study of automatic program repair techniques for security vulnerabilities,” in *Proc. Int. Symp. Softw. Reliab. Eng.*, 2021, pp. 196–207.

[42] Y. Wu et al., “How effective are neural networks for fixing security vulnerabilities,” in *Proc. Int. Symp. Softw. Testing Anal.*, Jul. 2023, pp. 1282–1294.

[43] G. Fraser and A. Zeller, “Mutation-driven generation of unit tests and oracles,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 278–292, Mar./Apr. 2012.

[44] F. Molina, M. d’Amorim, and N. Aguirre, “Fuzzing class specifications,” in *Proc. Int. Conf. Softw. Eng.*, 2022, pp. 1008–1020.

[45] C. S. Xia and L. Zhang, “Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2024, pp. 819–831.

[46] N. Pryce, J. Walton, and L. Vogel, “Hamcrest,” hamcrest.org. Accessed: May 15, 2023. [Online]. Available: https://hamcrest.org/

[47] L. Ouyang et al., “Training language models to follow instructions with human feedback,” in *Proc. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 27730–27744.

[48] D. Guo et al., “DeepSeek-Coder: When the large language model meets programming—The rise of code intelligence,” 2024, *arXiv:2401.14196*.

[49] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, “Massive exploration of neural machine translation architectures,” in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Sep. 2017, pp. 1442–1451.

[50] Z. Feng et al., “CodeBERT: A pre-trained model for programming and natural languages,” in *Proc. Findings Assoc. Comput. Linguistics: Conf. Empirical Methods Natural Lang. Process.*, Nov. 2020, pp. 1536–1547.

[51] D. Li and L. Murr, “HumanEval on latest GPT models—2024,” 2024, *arXiv:2402.14852*.

[52] “GPT-3.5 Turbo models documentation,” OpenAI. Accessed: Jun. 27, 2023. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5-turbo

[53] G. Ryan, “TOGA github issues—running TOGA with an assertless dataset,” GitHub. Accessed: Jun. 12, 2023. [Online]. Available: https://github.com/microsoft/toga/issues/3#issuecomment-1231773267

[54] S. B. Hossain, A. Filieri, M. B. Dwyer, S. Elbaum, and W. Visser, “Neural-based test oracle generation: A large-scale evaluation and lessons learned,” in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 120–132.

[55] J. Shin, H. Hemmati, M. Wei, and S. Wang, “Assessing evaluation metrics for neural test oracle generation,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 9, pp. 2337–2349, Jul. 2024.

[56] H. Coles et al., “PIT mutation testing,” Pitest. Accessed: Oct. 14, 2023. [Online]. Available: https://pitest.org/

[57] gvlasov. collections-utils. GitHub. Accessed: Jul. 15, 2023. [Online]. Available: https://github.com/gvlasov/collections-utils

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.

[58] arnohaase. a-foundation. Accessed: Jul. 15, 2023. [Online]. Available: https://github.com/arnohaase/a-foundation

[59] jcodec. jcodec. Accessed: Jul. 15, 2023. [Online]. Available: https://github.com/jcodec/jcodec

[60] Chat GPT's API is significantly slower than the website with GPT Plus. OpenAI discussion forum. Accessed: Jun. 3, 2023. [Online]. Available: https://community.openai.com/t/chat-gpts-api-is-significantly-slower-than-the-website-with-gpt-plus/98113/35

[61] C. Donahue, M. Lee, and P. Liang, "Enabling language models to fill in the blanks," in *Assoc. Comput. Linguistics*, Jul. 2020, pp. 2492–2501.

[62] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. Int. Conf. Softw. Eng.*, 2007, pp. 75–84.

[63] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Proc. Companion 22nd ACM SIGPLAN Conf. Object-Oriented Program. Syst. Appl. Companion*, 2007, pp. 815–816.

[64] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2011, pp. 416–419.

[65] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *Proc. Int. Conf. Softw. Testing, Verification Validation*, 2012, pp. 260–269.

[66] A. Blasi et al., "Translating code comments to procedure specifications," in *Proc. Int. Symp. Softw. Testing Anal.*, 2018, pp. 242–253.

[67] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proc. Int. Symp. Softw. testing Anal.*, 2016, pp. 213–224.

[68] J. Zhai et al., "C2S: Translating natural language comments to formal program specifications," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 25–37.

[69] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language API descriptions," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 815–825.

[70] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 288–298.

[71] R. Hodován, A. Kiss, and T. Gyimóthy, "Grammarinator: A grammar-based open source fuzzer," in *Proc. ACM A-Test*, New York, NY, USA: ACM, 2018, pp. 45–48, doi: 10.1145/3278186.3278193.

[72] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," 2019, *arXiv:1909.09436*.

[73] Y. Zhang, Z. Jin, Z. Wang, Y. Xing, and G. Li, "SAGA: Summarization-guided assert statement generation," *J. Comput. Sci. Technol.*, 2023. Accessed: Jul. 17, 2023. [Online]. Available: https://jcst.ict.ac.cn/en/article/doi/10.1007/s11390-023-2878-6

[74] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "SelfAPR: Self-supervised program repair with test execution diagnostics," in *Proc. Int. Conf. Automated Softw. Eng.*, 2023, pp. 1–13.

[75] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proc. Int. Conf. Softw. Eng.*, 2022, pp. 1506–1518.

[76] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-aware neural machine translation for automatic program repair," in *Proc. Int. Conf. Softw. Eng.*, 2021, pp. 1161–1173.

[77] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," in *Proc. Int. Symp. Softw. Testing Anal.*, S. Khurshid and C. S. Pasareanu, Eds., 2020, pp. 101–114.

[78] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, 2018.

[79] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *Proc. Int. Conf. Learn. Representations*, 2018. Accessed: Jan. 03, 2024. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[80] J. Patra and M. Pradel, "Nalin: Learning from runtime behavior to find name-value inconsistencies in Jupyter notebooks," in *Proc. Int. Conf. Softw. Eng.*, 2022, pp. 1469–1481.

[81] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 152–162.

[82] R. S. Malik, J. Patra, and M. Pradel, "NL2Type: Inferring JavaScript function types from natural language information," in *Proc. Int. Conf. Softw. Eng.*, 2019, pp. 304–315.

[83] M. Pradel, G. Gousios, J. Liu, and S. Chandra, "TypeWriter: Neural type prediction with search-based validation," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 209–220.

[84] J. Wei, M. Goyal, G. Durrett, and I. Dillig, "LambdaNet: Probabilistic type inference using graph neural networks," in *Proc. Int. Conf. Learn. Representations*, 2020. Accessed: Jul. 05, 2024. [Online]. Available: https://openreview.net/forum?id=Hkx6hANtwH

[85] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade, "DeepFix: Fixing common C language errors by deep learning," in *Proc. AAAI Conf. Artif. Intell., S.* P. Singh and S. Markovitch, Eds., 2017, pp. 1345–1351.

[86] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. Softw. Eng.*, vol. 47, no. 9, pp. 1943–1959, Sep. 2021.

[87] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *Proc. Int. Conf. Learn. Representations*, 2020. Accessed: Jul. 05, 2023. [Online]. Available: https://openreview.net/forum?id=SJeqs6EFvB

[88] G. A. Aye and G. E. Kaiser, "Sequence model design for code completion in the modern IDE," 2020, *arXiv:2004.05249*.

[89] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *Proc. Int. Conf. Softw. Eng.*, 2021, pp. 150–162.

[90] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. Int. Conf. Learn. Representations*, 2019. Accessed: Jul. 14, 2023. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[91] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. Int. Conf. Learn. Representations*, 2021. Accessed: Jul. 14, 2023. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[92] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Human Lang. Technol.*, Jun. 2021, pp. 2655–2668.

[93] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proc. Int. Conf. Automated Softw. Eng.*, 2020, pp. 473–485.

[94] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *Proc. Int. Conf. Automated Softw. Eng.*, 2022, pp. 443–455.

[95] B. Berabi, J. He, V. Raychev, and M. Vechev, "TFix: Learning to fix coding errors with a text-to-text transformer," in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 780–791.

[96] T. Le-Cong et al., "AutoPruner: Transformer-based call graph pruning," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 520–532.

[97] T. B. Brown et al., "Language models are few-shot learners," in *Proc. Neural Inf. Process. Syst.*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020, pp. 1877–1901.

[98] M. Chen et al., "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.

[99] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. Int. Symp. Mach. Program.*, 2022, pp. 1–10.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 03,2025 at 16:06:24 UTC from IEEE Xplore. Restrictions apply.