CCTEST: Testing and Repairing Code Completion Systems

Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, Cuiyun Gao

Abstract—Code completion, a highly valuable topic in the software development domain, has been increasingly promoted for use by recent advances in large language models (LLMs). To date, visible LLM-based code completion frameworks such as GitHub Copilot and GPT are trained using deep learning over vast quantities of unstructured text and open source code. As the paramount component and the cornerstone in daily programming tasks, code completion has largely boosted professionals’ efficiency in building real-world software systems.

In contrast to this flourishing market, we find that code completion systems often output suspicious results, and to date, an automated testing and enhancement framework for code completion systems is not available. This research proposes CCTEST, a framework to test and repair code completion systems in black-box settings. CCTEST features a set of novel mutation strategies, namely program structure-consistent (PSC) mutations, to generate mutated code completion inputs. Then, it detects inconsistent outputs, representing possibly erroneous cases, from all the completed code cases. Moreover, CCTEST repairs the code completion outputs by selecting the output that mostly reflects the “average” appearance of all output cases, as the final output of the code completion systems. With around 18K test inputs, we detected 33,540 inputs that can trigger erroneous cases (with a true positive rate of 86%) from eight popular LLM-based code completion systems. With repairing, we show that the accuracy of code completion systems is notably increased by 40% and 67% with respect to BLEU score and Levenshtein edit similarity.

I. INTRODUCTION

Large language models (LLMs) such as GitHub Copilot, OpenAI’s Codex, and Jurassic-1 are increasingly promoted for use within the software development domain. Machine learning (ML) over vast quantities of unstructured text, including websites, books, and open source code, is used to build such models, enabling them to produce “completions” given inputs made up of code and comments (documentation). To date, de facto LLM-based code completion frameworks are advocated with the aim to provide an “AI pair programmer” capable of automatically generating programs from natural language specifications or code snippets.

Despite being a compelling and promising component in augmenting modern software development, we observe that code completion systems are not perfect: they frequently generate confusing and possibly erroneous results. The “sub-optimal” and even buggy behavior of code completion systems are practically undesirable since it undermines the reliability and usability of code completion. Nevertheless, it is yet neglected by today’s research community, whereas existing studies on LLM-based code completion frameworks mainly focus on their security implication, cost reduction, or potential extension in different domains.

Considering Fig. 1(a), where a popular code completion system, Codegen [53], generates a code snippet as the completion of the input. However, by slightly tweaking the input, we observe that Codegen outputs a dramatically different code snippet, as illustrated in Fig. 1(b). While it is generally obscure to somehow directly decide the “correctness” of these two completed code snippets, given that the two inputs are identical to a human programmer, the high distinction between two completed code snippets indicates that Codegen’s outputs are of low consistency, which is a sign of unwanted outputs. Overall, this apparent inconsistency in the generated code completions motivates us to test and enhance code completion systems.

To do so, we form a testing oracle with the intuition that the completed code snippets should be structure-consistent.

We present CCTEST, an automated testing and enhancing framework for code completion systems. Our research thrusts are two-fold. First, we design a set of program mutation strategies, namely program structure-consistent (PSC) mutations, to generate mutated code snippets with similar or identical program structures.

1To clarify, “structure-consistent” in this paper refers to the process of transforming code snippets (i.e., inputs of code completion systems) into mutated code snippets with identical or little altered program structures.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
in $O$ by defining and comparing distances of generated code completions. A code completion output exhibits unusually large distances from other outputs will be deemed spurious. Second, we design a code enhancement strategy over completion outputs by selecting output $o$ that is mostly close to the “average” appearance of $O$. We show that $o$ generally manifests higher consistency with the ground truth, extensively improving the accuracy of code completion systems. Furthermore, CCTEST treats code completion systems as a “black box”, so we do not assume any specific implementation details of the code completion systems or their underlying LLMs.

CCTEST offers an up-to-date assessment of de facto LLM-based code completion systems and the quality of their outputs, whose defects impede the full potential of modern “AI pair programmer” in software development. From a total of 182,464 test inputs used for this study, we found 33,540 programs exposing code completion errors from eight widely-used LLM-based code completion systems, one of which (Copilot) is a popular commercial tool, and the other seven are either actively developed and maintained by the non-profit organization (CodeParrot [25]) or hosted by the industrial companies (EleutherAI’s GPT-Neo [14] and Salesforce’s Codegen [53]). With enhancement, we show that the average performance of code completion systems is notably increased by 40.25% and 67.43% with respect to BLEU score and Levenshtein edit similarity. In summary, we make the following contributions:

- We introduce and advocate a new focus on testing and enhancing code completion systems. This is a very important, timely topic, yet no existing testing work has been launched. We thus envision that our efforts can guide future research that aims to use or improve code completion tools.
- CCTEST is an automated testing and enhancement framework for code completion systems even in a black-box, remote API setting. PSC transformations, though relatively simple, are practical and low-cost, and specifically designed to deliver structure-consistent mutations. CCTEST also features a black-box scheme to enhance the quality of completion outputs without the need for retraining.
- We evaluate one commercial (Copilot) and seven popular free code completion systems, in which we successfully found 33,540 programs causing inconsistent code completion outputs. Our enhancement further improves the accuracy of the code completion systems by a large margin. We made various observations and obtained inspiring findings regarding modern code completion systems.

**Artifact availability.** We have released CCTEST to facilitate further research [12].

II. PRELIMINARY

This section introduces the background of code completion systems to deliver a self-contained paper. Note that the automated testing and enhancement pipeline shipped by CCTEST treats each code completion system as a black box, which means we do not require prior knowledge of the models or the underlying implementation. Nevertheless, given the dominating usage of LLMs in today’s code completion systems [1, 2, 26], we primarily introduce LLM-based code completion.

Fig. 2 presents a holistic view of LLM-based code completion systems, and explains how CCTEST fits its workflow for testing and enhancement. Benefiting from the prosperous development and success of Transformer-based natural language models such as OpenAI’s GPT2 and GPT3 [15, 19, 42], the code completion task, as a typical conditioned open-ended language generation task, is extensively improved with much higher accuracy and applicability.2

Though training data details are often obscure, modern LLMs-based code completion systems are advertised as being trained with millions or even billions of lines of code [2]. Typically, the input training data are dissected into sentences and further into tokens, where each token comprises a sequence of characters before being fed to LLMs. For instance, the tokenizer of Codex and GPT-3 is configured to produce tokens with an average of four characters. Tokens are encoded with unique numeric identifiers, up to user-defined vocabulary size. This process is often referred to as byte pair encoding (BPE) [28], allowing the models to encode any rare words with appropriate subword tokens. Various techniques are proposed to improve the performance of LLMs [85], such as learning from rare tokens and deciding on a proper stop word.

During the query phase, the code completion system’s input is often referred to as a prompt, denoting an incomplete code snippet. Similarly, the prompt code snippet is first abstracted into sentences and further into tokens, for which the code completion system can predict a list of suggestions (ranked by the confidence scores) that are more likely to continue/complete the input prompt. For instance, the code completion system is frequently assessed by completing a function body, given the function prototypes and some statements in the function prologue. Note that modern code completion systems can process prompts of different types, including the expected program’s code snippets and natural language descriptions (comments). Such comments are usually divided into background, input, and output to describe a competitive programming problem [44].

CCTEST is designed to test the code completion system by mutating the seed prompts and identifying bug-triggering prompt variants by cross comparing the outputs. Moreover, CCTEST can enhance the code completion output by analyzing outputs of prompt variants to select cases that are closest to the “average” appearance of $O$.

2 A full introduction of transformer and its usage in open-ended language generation is beyond the scope of this paper. Audiences can refer to [71].
III. APPROACH OVERVIEW

Fig. 3 presents an overview of CCTEST in terms of testing and enhancing code completion systems. In particular, 

1. **Prompt Variants Generation.** CCTEST launches PSC mutations and generates a set of structure-consistent variants \( P \) of an input prompt \( p_0 \). Here, we propose a novel approach, PSC, that mutates a prompt with code structure-consistent transformations. The mutated prompts manifest closely consistent structures from human perspective.

2. **Testing Oracle Generation.** The target code completion system outputs a set of completion outputs in accordance with the prompt variants. Here, we form a testing oracle over completion outputs. The key observation is that the code completion outputs should invariably manifest high (visual) consistency for prompt variants produced by using PSC. Thus, “outlier” completion outputs are deemed erroneous, whose corresponding prompt variants are defect-triggering inputs.

3. **Completion Output Enhancement.** The testing pipeline can be extended to enhance code completion outputs. To this end, CCTEST identifies output \( o \) that is closest to the “average” appearance of \( O \) (with respect to program distance metrics; after excluding outliers in 2). This step does not assume any prior knowledge of the code completion system implementation and, therefore, applies to black-box scenarios.

**Study Scope.** We primarily target the erroneous code completion outputs, denoting “stealthy logic bugs” of code completion systems. As detailed in Sec. III-B, our testing approach can automatically expose inconsistency defects in code completion systems, meaning that when given a set of structurally consistent prompts, the completion outputs are deemed to share closely similar appearances as well.

During the preliminary study, we also find that certain (mutated) prompts may simply impede code completion systems from generating any outputs. To some extent, this is comparable to identifying a “crash” of the code completion system. Although such obviously anomalous states are not the primary focus of CCTEST, we still record and report all such defects we encountered during the evaluation.

Note that we are not using extreme (broken) prompts to stress code completion systems. Modern code completion models are on the basis of LLMs, and our preliminary study shows that providing some trivial code snippets as prompts may make the code completion system generate meaningless outputs. The current implementation of CCTEST focuses on generating syntactically valid Python code snippets as testing prompts, given the Python language’s popularity and representativeness. Nevertheless, our method is not limited to Python; we leave supporting other programming languages as one future work.

A. Program Structure-Consistent (PSC) Mutations

To test code completion systems, our key observation is that programs with identical/mildly-changed control structures will retain such consistency in the completion outputs. Hence, by observing certain completion outputs manifesting high inconsistency, potential code completion errors can be flagged.

### Design Goal

To generate structure-consistent mutants, we implement a set of PSC transformations, which mutate a seed prompt from different levels of the program hierarchy. Table I lists all the proposed PSC transformations. Each of them is designed to be “lightweight,” in the sense that it only slightly changes the prompts and retains the structure consistency of mutated prompts. However, we make an encouraging observation that such straightforward, incremental mutations impose notable challenges for code completion systems to process inputs and accordingly generate consistent completion outputs. We now introduce each PSC scheme below.

**REP_R & REP_C.** Our first two schemes rename function parameters. Overall, they will replace one function parameter with a new identifier, and every usage of this parameter will be identified (by matching identifier names) and replaced accordingly. The naming can either be “regulate” or “context.” Considering the following case,

```python
// seed prompt
def add(a, Param1):
    res = a + Param1

def add(a, b):
    res = a + b
```

where for the “regular” scheme (REP_R), we replace the function parameter \( b \) with \( \text{Param1} \). As REP_C, which takes the specific context into account, we extend the function parameter \( b \) with a new identifier that subsumes both function name \( \text{add} \) and \( \text{b} \). It is evident that these two methods choose a relatively fixed “pattern” to mutate prompts. However, we clarify that they are designed by taking account developers’ programming habits to replace the parameters, which shall help (LLM-based) code completion systems avoid generating irrelevant outputs when they behave correctly. This design intuition applies to following PSC transformations as well, and they are very effective to provoke code completion errors.

### Table I: Transformation schemes implemented in CCTEST.

| Class    | Methods                          | Abbreviations |
|----------|----------------------------------|---------------|
| Identifier Level | rename parameter regulate        | REP_R         |
|          | rename parameter context         | REP_C         |
|          | rename local variable regulate   | REL_R         |
|          | rename local variable context    | REL_C         |
| Instruction Level | instruction replacement          | IDR           |
|          | replace boot expression          | RBT           |
| Block Level | garbage code insertion regulate  | GRA_R         |
|          | garbage code insertion context   | GRA_C         |
|          | print statement insertion        | INT           |
REL_R & REL_C. These schemes rename local variables. Similar to renaming parameters, these schemes randomly select a local variable whose scope is in the function. Then, we replace all of its references with a new identifier. Considering the following case,

where for REL_R, we replace the local variable res with LocalVar1. As for REL_C, which takes the context information into account, we rewrite the local variable res with a new identifier that subsumes both function name compare and res.

IRR. This scheme implements a set of mapping rules to search and replace certain common arithmetic operators with semantic equivalent, yet syntactically different forms. Considering the following case,

where the += operator is replaced with an addition. We implement four mapping rules over different common arithmetic operators; audiences can refer to our codebase for details [12].

RTF. Besides mutating arithmetic expressions, we also implement RTF to mutate boolean expressions, particularly expressions used in forming branch conditions, with their semantics equivalent variants. Considering the following case,

where the boolean expression is extended with an always-true condition. This would not alter the functionality of the input prompt, nor does it largely change the program structures. However, we find that such mutated boolean expressions can effectively impede code completion systems from generating structure-consistent outputs, as shown in Sec. V.

GRA_R & GRA_C. These schemes insert a small chunk of “garbage code” into the program, which does not alter program semantics, but slightly increases program control flow complexity. We use always-false conditions to form an if condition, then insert a small set of statements, which will never be executed, into the “dead” branch. Similar to mutations mentioned above, creating garbage code may take context information into account. Considering the following case,

where GRA_R inserts a branch whose if condition expression is “False” with a new variable named TempVar. As for GRA_C, which considers context (local variables and parameters), we create an always-false condition on the basis of the syntactic form of the function parameters, and use it to form the if condition expression. Similarly, variable names in the enclosed branch also subsume both function name and TempVar.

INI. The last mutation scheme inserts a print statement into the prompt. Though INI only slightly changes a prompt, we find that its induced code completion output effectively improves the output quality of code completion systems; see details in Sec. V-C.

Clarification. While applying most PSC transformations preserve the control-flow structures of a seed prompt, garbage code insertion (GRA_R and GRA_C) schemes slightly alter program structures. Intuitively, code completion outputs for GRA_R- and GRA_C-mutated prompts should appear more distinct than completion outputs induced by other PSC schemes. However, our preliminary study and observation show that more extensive PSC transformations may not inevitably “dominate” outliers in the code completion outputs. In fact, as reported in Sec. V-B, all PSC schemes contribute reasonably to uncovering code completion outliers. Thus, we believe designing PSC schemes like garbage code insertion is proper.

Alternative: Mutating Natural Language Comments. Besides processing prompts in the source code form, modern code completion systems can also generate code completions given prompts in natural language sentences. In such cases, the input sentences are usually code comments or descriptions of the intended code functionality. Careful readers may wonder about the feasibility of mutating natural language comments to test code completion models, whose expected workflow may be similar to recent advances in testing machine translation systems [68, 69]. In fact, we tentatively explored this direction. We clarify that unlike machine translation system testing, whose inputs are arbitrary natural language sentences, code comments often have limited mutation space; for instance, a vast majority of code comments have no adjectives [58, 60]. This distinction makes it hard to mimic testing methods for machine translation.

B. PSC Testing: Forming Testing Oracles

Given a list of code completion outputs \( O \) in accordance with the mutated prompts \( \mathcal{P} \), we compare each output \( o_i \in O \) with the remaining cases in \( O \) and decide if it is an “outlier.” The complete algorithm is shown in Algorithm 1. We first iterate each case in \( O \), decide its similarity with the remaining cases (lines 2–3), and normalize the scores (line 4). Here, Sim is implemented using the Levenshtein string-level edit similarity provided by fuzzywuzzy [3], a standard algorithm to decide the edit similarity among two programs. Then, we employ a threshold \( T \) to decide whether a case exhibits anomalously low similarity with other cases for more than \( T \) times (lines 5–12) and deem the abnormal case as an outlier. The corresponding \( p_j \) is deemed as an error-inducing input. Overall, given that we have implemented \( N \) (\( N = 9 \) in the current implementation
of CCTEST) schemes to mutate a prompt, $T$ is a configurable hyperparameter ($T \leq N$), such that a code completion output deems an outlier if its edit distance scores with $T$ code completion outputs are less than the average distance score (computed by Median) of code completion output pairs.

$T$ will be decided with empirical evidence, as will be discussed in Sec. V-B. After performing Algorithm 1, we keep the remaining code completion outputs $O^* = O \setminus \mathcal{L}$ for usage in the enhancement phase (Sec. III-C).

**Alternative: Sim Metrics.** One may question the current implementation of Sim, which is based on string-level edit distance rather than structure-level distance. During preliminary study, we explored using structure-level distance metrics to implement Sim. Nevertheless, speed is a main bottleneck, as tree-edit distance is much slower. Moreover, code completion outputs are usually code snippets consisting of approximately ten (or fewer) lines of code whose “structures” may be less significant; we find that standard tree-edit distance metrics often give vague results. Thus, we deem that using Levenshtein string-level edit distance proper and practicable for our testing.

![Algorithm 1 Outlier selection algorithm.](image)

We clarify that this oracle is often too strict. Fig. 4 compares the code completion output of Copilot and the ground truth code snippet (this code snippet is from LeetCode [7]). While they have identical semantics, their syntactic forms are distinct. The ground truth uses a helper function inorder to deliver a classic inorder traversal of the input binary tree, whereas Copilot output appears to be leveraging recursive calls. Overall, though the code completion output may frequently appear syntactically distinct from ground truths, the completion outputs should be deemed as functionally correct. Therefore, comparing the syntactic equality of completion outputs with ground truth may not faithfully reflect true defects of code completion systems. Such a strict oracle may induce many false positives.

Checking Functionality According to Ground Truth. Recent works explored strictly checking the functionality correctness of code completion outputs in accordance with ground truth. [52] tests Copilot using prompts created from LeetCode programs, then counts the number of passed LeetCode test cases over the code completion outputs (each completion output forms a LeetCode solution, when put together with the prompts). Similarly, HumanEval [18] assesses the functionality correctness of code completion outputs with hand-written programming problems. Overall, for those approaches, more passed test cases indicate that the completion outputs are of higher quality. Those approaches, however, may be less desirable than our formed oracle. Modern code completion systems are not championed to replace human programmers; rather, it aids human programmers by suggesting useful code snippets. Therefore, strictly correct functionality may not be the first design target for a code completion system, e.g., some sloppy arithmetic errors in completion outputs may be easily fixed by users. Thus, we believe it is more desirable to cross compare the structural-level consistency (as in CCTEST), rather than using test inputs to check the functionality correctness.

### C. Completion Output Enhancement

This section presents the technical pipeline of enhancing the code completion outputs. By launching the testing in Sec. III-B, we collected a set of code completion outputs $O^*$, with outliers being excluded. After that, we aim to identify a code completion output that appears mostly close to the “average appearance” of $O^*$. To this end, we measure the average pair-wise edit similarity $s$ of every $o_1, o_2 \in O^*$. Note that at this step, we re-use the Sim function implemented in Alg. 1, meaning that we compute the Levenshtein string-level edit distance as the similarity score. Then, we search for an $\tilde{o} \in O^*$, whose average pair-wise similarity scores with all other elements in $O^*$ is the closest to $s$. This $\tilde{o}$ will be the repaired code completion output returned to the users.

To clarify, CCTEST aims to enhance the quality of code completion outputs, in the sense that the repaired output is closer to the “average-looking” output. As in our evaluation (RQ3; see Sec. V-C), we use edit similarity and BLEU scores as metrics to unveil that CCTEST can find better completions, by showing that the repaired outputs have closer distance with the ground truth under both metrics. Nevertheless,
CCTest cannot guarantee that the repaired outputs become “semantically correct”, if the original outputs are not. In other words, CCTest does not directly repair the semantics-level defects/inconsistency. Semantics-level code repairing is inherently hard. More importantly, it is unaligned with the design goal of CCTest. As already noted in Alternative Oracle above, we believe it is less proper to check the functionality correctness in this research context.

IV. IMPLEMENTATION AND EVALUATION SETUP

CCTest is implemented in Python, with about 5k LOC. CCTest currently focuses on mutating Python code, given the popularity of this language in software development and the maturity of corresponding code completion systems. We now discuss the implementation and evaluation setup.

Parsing and Mutating Programs. We parse Python code with tree-sitter [16], a mature and dependency-free parser generator tool with an open-source license. It is widely used in code-related projects such as Codebert [26], and it does not require the input code to be executable, meaning that incomplete code fragments without a building script can also be parsed. We first parse the prompt code into a concrete syntax tree, and conduct several sanity checks (see in Sec. V-A) on the target code to see which PSC transformation could be performed. Then, after applying feasible PSC transformations, CCTest will output the corresponding transformed prompts with IDs of the applied PSC transformations.

| Split          | Total # seed programs | Average # token per seed | Max # token among seeds | Total generated variants | Average # token per mutated seed | Max # token among mutated seeds |
|---------------|-----------------------|-------------------------|------------------------|--------------------------|---------------------------------|-------------------------------|
| LeetCode       | 613                   | 149.4                   | 513                    | 4,296                    | 151.6                           | 525                           |
| CodeSearchNet  | 2,297                 | 115.4                   | 623                    | 15,602                   | 117.3                           | 629                           |

**Seed Programs.** Consistent with most research in this field, we form our evaluation dataset from a popular repo of LeetCode solution programs [6] and CodesearchNet [39]. LeetCode is an online platform to practice algorithmic coding challenges for technical interviews. As will be noted in Sec. III, since LLM-based code completion systems limit the max token size, we only select programs whose program length is between 32 and 2048. Overall, each seed program contains a medium size function with generally complex control structures. As in Table II, we pick 613 code snippets from the LeetCode solution repo as the seed programs. The total number of generated variants for LeetCode programs is 4,296 (recall not every PSC transformation is applicable to arbitrary Python code).

CodeSearchNet is particularly designed in the research of code representation learning. The Python component of this dataset contains train, validation, and test splits; we use the test split for the evaluation. Similar to LeetCode, we only select programs with token length between 32 and 2048, and for the code snippets which share the same attribute named “path”, we only maintain one to keep the result balanced.

We clarify that both LeetCode and CodeSearchNet are deemed proper for neural code learning tasks like code completion [39] and they are extensively used in benchmarking relevant models [26, 33, 34, 81]. Thus, errors and enhancement demonstrated in our evaluation should mostly reflect common obstacles and improvement users can expect in real-life scenarios. In contrast, overly complex prompts (e.g., real-world complex software) may unavoidably impede the understanding and assessment of LLM-based code completion systems.

**Statistics of Test Cases.** Table II reports statistics of the test cases. We collect a total of 2,910 programs from LeetCode/CodeSearchNet as test seeds for each code completion system. The total number of generated variants is 19,898 (see Table VII for the breakdown). For each seed program with its variants, we equally split the function into two parts: the first part is used as a prompt, and the remaining is used as “ground truth” to assess our enhancement, see details in Sec. V-C.

**Code Completion Systems.** Table III reports the code completion systems used in the evaluation. First, we use Github Copilot, one highly visible commercial code completion system that generates quite a buzz in the community [82]. We purchase the standard commercial license (for single user) to unleash its full potential. As a “black-box” commercial product, it is unclear about their implementation (marked as ? in Table III).

We also evaluate several well-known LLM models for code completion, including CodeParrot [25], GPT-Neo [14], GPTJ [24], and CodeGen [53]. All of these models are deemed to employ large-scale language models, given that up to billions of parameters are involved in their underlying models, and tens of thousands of vocabularies are considered. CodeParrot is a GPT-2 model trained specifically for Python code generation. We use two variants, CodeParrot-small and CodeParrot. The smaller variant contains less amount of parameters and is trained over fewer data. Nevertheless, both variants manifest a high level of code generation capability. As for GPT-Neo, we use two variants, GPT-Neo-125M and GPT-Neo-13B, which are both GPT3-like models. Our observation shows that GPT-Neo-125M is prone to generate less diverse but more straightforward code snippets compared to its larger variant. Given that said, we believe both models are well-suited for code generation and manifest reasonably high robustness under our testing campaign.

**TABLE III**

| System Name | # Params | # Vocab (tokens) | # Max. tokens |
|-------------|---------|-----------------|---------------|
| Copilot     | 110M    | 32,768          | 1024          |
| CodeParrot-small | 1.5B | 32,768          | 1024          |
| CodeParrot | 125M    | 50,257          | 2048          |
| GPT-Neo-13B | 1.3B    | 50,257          | 2048          |
| GPT-J     | 6B      | 50,400          | 2048          |
| Codegen-2B | 2B      | 50,400          | 2048          |
| Codegen-6B | 6B      | 50,400          | 2048          |
include the vast majority of GitHub’s open-source code. GPT-J and GPT-Neo are trained on the Pile dataset [30], a diverse language modeling dataset. We download their pre-trained models from huggingface [4] and run the code completion locally for all LLMs except Copilot. LLMs support different search (sampling) strategies to generate code completions, such as beam search, temperature sampling. These factors make our testing pipeline less deterministic. Thus, in our experiments, we follow Copilot’s interface to only choose the completion result with the highest confidence score as its “output.” For other tested models, we disable their sampling strategy.

In all, LLM-based code completion systems presented in Table III, to the best of our knowledge and experience, represent the best systems available to the public. We tentatively explored code completion solutions based on conventional machine learning or rule-based approaches. We clarify that these conventional methods were seen to produce much worse and shallow code completion outputs compared with these LLM-based modern systems.

V. FINDINGS

In evaluation, we mainly explore the following research questions. RQ1: Can CCTEST generate high-quality and structure-consistent prompt variants? RQ2: How effective is CCTEST in detecting code completion defects? RQ3: To what extent can CCTEST enhance the quality of code completion outputs? We answer these RQs in subsections below.

A. RQ1: Effectiveness on Input Generation

Answering RQ1 requires assessing the quality of mutated prompts. At this step, for each seed prompt, we generate mutants and first check whether they pass the parsing. In particular, we generate up to 9 mutants for each prompt, leading to a total of 19,898 mutant prompts generated on top of 2,910 seed prompts. We use the standard ast module in Python to parse all transformed Python prompts into abstract syntax trees to check their validity. All generated mutant prompts are parsed without any error, indicating that they are grammatically valid.

| Distance | (0, 0.01] | (0.01, 0.1] | (0.1, 1] | (1, 1.5] | (1.5, 2] | (2, 3] | (3, 10] | (10, 100] |
|----------|-----------|------------|--------|---------|---------|-------|--------|----------|
| Freq (%) | 90.16     | 8.90       | 0.92   | 0.30    | 0.32    | 0     | 0      |          |
| Cumulative Freq (%) | 90.16 | 98.46 | 99.38 | 99.68 | 100.0 |          |        |

To illustrate the structure consistency of mutated prompts, we use pycode-similar [8], a well-performing tool to calculate Python code similarity. This similarity metric is based on AST structures, which is different from Levenshtein edit similarity that we use in deciding the “outliers”. Let the prompt AST have $n$ nodes, where $m$ nodes are matched toward nodes on the mutated prompt’s AST. pycode-similar returns ratio $\frac{m}{n}$, denoting how similar two programs are. We report the distribution of the “distance score” (distance is computed as $1 - \frac{m}{n}$) in Table IV. For the vast majority of mutated prompts (over 98%), structural-level distance is less than 0.1, meaning over 90% of AST nodes are matched. Recall that several PSC transformations (Sec. III-A) introduce only identifier-level changes. As expected, the distance scores between their mutated and seed prompts are zero, implying that program structures are retained. A few cases have slightly higher distances. With manual inspection, we find that it is primarily because the size of original prompts is short, leading to an increased distance $1 - \frac{m}{n}$ where $n$ is small.

Answer to RQ1: CCTEST can generate large-scale, grammatically valid, and structurally consistent prompt mutants.

B. RQ2: Bug Detection

To answer RQ2, we first launch testing to detect code completion defects. Table V reports our findings. Note that we use two datasets for testing, including 4,909 prompts from the LeetCode dataset and 17,899 prompts from the CodeSearchNet dataset. In Table V, “3003 + 12101” in the first “#Outliers $T = 1$” cell means that 3,003 outliers are found using the LeetCode mutated inputs, whereas 12,101 outliers are found using the CodeSearchNet mutated inputs. The same format applies for the “#No Results” column.

First, as shown in the second column of Table V, while nearly all mutated prompts can be processed, we still find 58 (0.03%) mutated prompts that trigger “no response” for the tested code completion systems. As expected, such cases are rare, given the high capability and comprehensiveness of LLM-based production code completion systems.

For the outlier detection evaluation, we assess the performance under different values of the hyper-parameter $T$. As clarified in Sec. III-B, CCTEST leverages a threshold $T$ to form the testing oracle: a code completion output is deemed an outlier if it has a small similarity score with $T$ code completion outputs mutated from the same seed prompt. Therefore, $T$ is an integer ranging from 1 to the total number of schemes (9 in our implementation). Table V reports the number of uncovered outliers in accordance with different models and thresholds. Overall, out of 182,464 test cases, a substantial number of defects are found under all five configurations. As expected, the increment of $T$ decreases the number of detected defects (outliers). As we illustrated in Algorithm 1, the selection stringency for outliers depends on the threshold $T$, where a higher $T$ represents a stricter standard. Also, different from some robustness testing papers (e.g., [31]) that merely detecting mutated inputs that lead to outlier predictions, bad completions are derived from either original (what users start with) or mutated prompts in this study. For instance, when $T = 9$, among all 33,540 (5912 + 27628) bad completions, 3,008 (8.9%) are outputs derived from the original prompts. That is, a reasonable portion of “bad completions” are from original prompts.

Copilot outperforms the other seven LLMs, given fewer inconsistency defects (293+2184 for $T = 9$). However, Copilot has the most “no response” failures (41 out of 58). We suspect that Copilot will refuse to return any output when its model in the remote server fails to generate code snippets with valid syntax or high confidence scores. Despite the small number of “no response” failures, all the other cases can be processed by code completion systems to produce non-trivial outputs.
Manual Validation. At this step, we first measure the true positive (TP) rate of the outliers found under different thresholds $T$. For each pair of $(T, model)$, we randomly sample 100 cases from two datasets, resulting in a total of 4,000 ($5 \times 8 \times 100$) cases. The first two authors check each case to manually decide if an outlier is TP or false positive (FP). Our manual inspection results are presented in the “TP” row of Table VI. It shows that with the increment number of $T$, the number of TPs increases. Particularly, when $T = 9$, out of 800 positive findings of CCTTest, we have only 111 (800 – 689) cases that are FPs. To better understand our findings, we further analyze those remaining 111 cases and summarize the following two main reasons for FPs.

(i) The completion results are highly robust for the majority of mutates. To select an outlier, Algorithm 1 will cross compare the outputs from the same seed program. However, it would be possible that all other results are exactly the same except for one mutant, which would be treated as an “outlier”, even if it is only slightly different from the others. Such cases count for the 59.46% of FPs. (ii) No “mainstream” results. The completion results for some vague prompts would vary drastically, and sometimes the vast majority, if not all completion outputs, appear to be different. During the manual inspection of such cases, authors would not deem any mutants as “outliers.” Such cases count for 26.13% of FPs.

Recall that our test oracle uses string-level distance metrics to decide outliers, instead of directly using program structure-level distance metrics. We have clarified why string-level distance is preferable and practical to structure-level distance in Alternative Sim Metrics paragraph of Sec. III-A. From the above manual inspection, it is evident that using string-level distance metrics does not primarily add FPs; we find that when two completion outputs exhibit a high distance in string-level metrics, they generally look distinct from program structures as well. Overall, empirical results here further support using string-level distance metrics in deciding outliers.

The above study explores TP/FP over 4,000 “positive” findings. As a testing tool, CCTTest cannot avoid false negatives (FNs). Nevertheless, measuring FNs help understand the potential of CCTTest. Hence, at this step, we randomly select 4,000 negative samples from CCTTest’s findings. Two authors manually check the FN and true negative (TN) cases (the procedure follows how we confirm TP/FP). We report the number of FNs in Table VI. We accordingly compute the precision, recall, and F1 scores. It is observed from the “FN” row of Table VI that the number of FNs increases when $T$ grows. This is reasonable, given that the higher $T$ is, the more likely CCTTest may miss some findings. Nevertheless, CCTTest becomes more accurate when $T$ grows, as reflected by the F1 scores in Table VI. In short, when $T = 9$, CCTTest achieves reasonably high accuracy. We interpret the results as overall encouraging and reasonable, and we recommend using $T = 9$ as the default setup in practice.

| System | #No Results | #Outliers |
|--------|-------------|-----------|
| CodePert | 4712 | 2042 |
| CodePert-smart | 4712 | 2042 |
| GPT-J | 901 | 260 |
| GPT-NEXUS-13B | 901 | 260 |
| GPT-NEXUS-125M | 901 | 260 |
| Codector-28-mono | 219 | 67 |
| Copilot-44-mono | 211 | 70 |
| Total | 10148 | 4068 |

**TABLE VI**

Assessing CCTTest’s findings with manual investigation.

|       | $T=1$ | $T=5$ | $T=9$ | $T/7$ | $T/9$ |
|-------|-------|-------|-------|-------|-------|
| TP    | 216   | 342   | 429   | 537   | 689   |
| FN    | 26    | 38    | 70    | 90    | 104   |
| Precision | 0.707 | 0.427 | 0.536 | 0.671 | 0.861 |
| Recall | 0.892 | 0.900 | 0.859 | 0.856 | 0.868 |
| F1 score | 0.414 | 0.579 | 0.660 | 0.752 | 0.865 |

**TABLE VII**

Distribution of schemes which trigger outliers when $T = 9$. LC represents LeetCode and CNS represents CodeSearchNet.

| Pass name | IRR | IRS | IRS-C | IRP | IRP-C | IRP-J | IRP-J-C | IRR-C | IRS-C | IRP-C | IRP-J-C | IRR-C | IRS-C | IRP-C | IRP-J-C |
|-----------|-----|-----|-------|-----|-------|-------|---------|------|-------|------|--------|-------|-------|------|--------|
| #variants on LC | 254 | 3998 | 3987 | 4287 | 4287 | 4287 | 4287 | 4287 | 4287 | 4287 | 4287 | 4287 | 4287 | 4287 | 4287 |
| #variants on CNS | 153 | 2293 | 2297 | 2297 | 2297 | 2297 | 2297 | 2297 | 2297 | 2297 | 2297 | 2297 | 2297 | 2297 | 2297 |
| outliers on LC (%) | 3.51 | 17.03 | 17.57 | 11.35 | 8.33 | 12.70 | 4.94 | 17.03 | 11.35 | 8.33 | 12.70 | 4.94 | 17.03 | 11.35 | 8.33 |
| outliers on CNS (%) | 0.27 | 15.41 | 18.11 | 11.62 | 14.59 | 11.62 | 14.59 | 14.59 | 11.62 | 14.59 | 11.62 | 14.59 | 14.59 | 11.62 | 14.59 |

**Potency of PSC Transformations.** We measure the potency of all nine PSC transformations about their contributions to uncovering outliers. For both LeetCode and CodeSearchNet, we report the distribution of generated variants and triggered outliers in Table VII. Due to limited space, we report the average distribution across all code completion systems; see [12] for each system. Overall, all PSC transformations reasonably facilitate uncovering outliers, even though they are performed toward different levels of program hierarchical representation (e.g., identifier level or basic block level). In particular, nearly all schemes, except “IRR” and “RTF”, have comparable contributions. IRR has a relatively minor application scope, given that IRR looks for specific arithmetic operations like “*=-” that may not be pervasively used. Similarly, RTF requires the existence of an i f branch with a relatively simple condition, which may not be available in our test cases. As a consequence, these two schemes result in less amount of uncovered outliers, as shown in the last two rows in Table VII.

**Case Studies.** Table V illustrates that the inconsistency bugs in code completion systems are pervasive, and numerous defects can be found even for highly permissive consistency thresholds. We present two representative cases in Fig. 5 and Fig. 6. Fig. 5 presents a case such that the completion output of the seed prompt largely deviates from the ground truth. When only given
the completion output in Fig. 6(b), it is inherently challenging to
decide if the deviation between completion outputs in Fig. 6(a)
and Fig. 6(b) is due to model capacity or bugs. Nevertheless,
when referring to Fig. 6(c), it becomes evident that the tested
code completion system, GPT-J, is capable of generating high-
quality completion outputs with closer structure consistency
to the ground truth. We only tweak a little on the parameter
name for the prompt in Fig. 6(c) compared to the seed. Thus,
it should be accurate to consider Fig. 6(b) as code completion
bug, which may be likely repairable.

Fig. 7(a) presents another case, such that the code completion
output of the seed prompt appears to be highly similar to
the ground truth in Fig. 7(b). In contrast, when applying the
REL_R scheme to mutate a local variable’s name, the
code completion output of a mutated prompt (as in Fig. 7(c))
becomes largely distant from the ground truth, which clearly
denotes a bug in the code completion system.

**Time & Cost.** We employ a GPU server for running the
involved models locally. The server has an Intel Xeon Platinum
8276 CPU, 256 GB memory, and 4 NVIDIA A100 GPUs.
Although processing time is generally not a concern, we
record and report that it takes 35 GPU seconds to finish
completing one prompt on average. For each seed, CCTEST
takes 2.7 CPU seconds to generate nine mutated prompts and
about 82 GPU seconds for code completion systems to infer
and obtain the results in parallel. The outlier detection and
enhancement (discussed in **RQ3**) steps take about 1.1 CPU
seconds. Overall, we admit that generating prediction from
many mutated prompts and “averaging” them will take time,
compromising interactive response time. Nevertheless, **RQ3**
illustrates that code completion outputs are of much higher
quality. Moreover, for commercial models, Table V has revealed
that roughly 10.86% percent of its code completion outputs
are spurious even when $T = 9$. In other words, we interpret
that about 10.86% percent of the code completion outputs are
highly confusing to the users and are thus “wasted.” This may
indicate an undesirable situation and potential financial loss,
given that modern cloud-based code completion systems may
feature a “pay-as-you-go” mode, where users are charged based
on how many queries they send to the services.

**C. RQ3: Enhancement Effectiveness**

To answer **RQ3**, we first present the improved accuracy of
code completion systems. Then, we assess the potency of each
PSC scheme about their contributions to enhancement. Lastly,
we conduct a human evaluation to verify the effectiveness of
CCTEST in enhancing code completion tools.

**Improved Accuracy.** We report the improved accuracy with
respect to both Levenshtein edit similarity [3] and BLEU score
[56] in Table VIII when $T = 9$. Note that both metrics are
commonly used in relevant research to assess the performance
of LLMs [33, 34]; a higher edit similarity score or BLEU
score between the “ground truth” and the completion output
indicates better performance. To clarify, Table VIII reports the
enhancement ratio. For instance, when assessing Copilot against
the LeetCode dataset, let the edit similarity or BLEU score be

---

**Answer to RQ2:** CCTEST identifies numerous defects when being used to test (commercial) code completion systems, despite the varying thresholds used in deciding outliers. We recommend configuring $T = 9$ as a presumably proper threshold (with the highest TP rates) in usage.
We compute the enhancement ratio as \( r = \frac{s^\prime - s}{s} \), where \( s^\prime \) is the edit similarity/BLEU score after enhancement. Similar to RQ2, “21.22 + 33.98” in the first Copilot cell means that the relative gain of average BLEU score on LeetCode dataset is 21.22% and on CodeSearchNet dataset is 33.98%.

Cost. As noted in Time & Cost paragraph in Sec. V-B, processing all prompts mutated from a seed takes about 82 GPU seconds on average, while the cost of outlier detection and enhancement selection is about 1.1 CPU seconds. Therefore, we estimate that an acceptable cost is incurred for a substantial improvement. Note, however, that commercial code completion systems may have a “pay-per-query” mode, in which each query is charged. Therefore, it may be preferable for the service provider (instead of users) to deploy CCTest.

Potency. Let completion output \( o_i \) be the output used for enhancement, and \( o_i \) is generated using the prompt mutated from PSC transformation \( t_i \). We deem \( t_i \) under this circumstance as the “optimal” transformation that contributes to the code completion enhancement. For both LeetCode and CodeSearchNet, we report the distribution of different transformations selected as “optimal” in Table IX. We find all PSC transformations are effectively served as “optimal” in both datasets. As clarified in the “potency” evaluation in RQ2, “IRR” and “RTF” have smaller application scope and fewer variants, which indicates their relatively lower proportion of being “optimal” on both datasets. However, an exception is that “IRR” contributes disproportionately well when using the LeetCode dataset. With manual inspection, we find that the programs in LeetCode are more likely to contain arithmetic operations such as “+ =” compared to programs in CodeSearchNet. In general, we interpret the evaluation results are highly encouraging, showing that almost all the designed schemes manifest high applicability and effectiveness in enhancing different models.

Human Study. We further conduct human evaluation to inspect the quality of the enhanced completion outputs. We randomly select 60 samples and create an online questionnaire for them. We invite twelve experts, including four industrial developers and eight academy researchers with expertise in LLMs, as the participants. We provide two completion outputs for each seed program without specifying whether they are the original or the CCTest’s enhanced outputs. On a scale from 1 to 5, the participants are asked to provide a score (1 for a completely not satisfying output and 5 for a fully satisfying output).

To ensure that all participants correctly understood the meaning of the questionnaire, we prepare and insert five sanity-check (SC) test items randomly into the questionnaire, and only participants who answer all SC correctly are considered legitimate. We evenly assigned 30 real samples with five SC to each participant, and ensured that each selected sample was examined by six participants. All participants passed the sanity check and the average completion time was 45 minutes.

The human evaluation results show that the average score for the original outputs is 2.3188, whereas for CCTest’s enhanced outputs, the average score is 3.1565, representing a relative gain of 36.12%. We analyzed all their answers. Respondents believe that for 2.2% of the cases, the enhanced completion outputs look worse than the original outputs, while the remaining 97.8% treat the enhanced outputs equal to or better than the original ones. The Fleiss’ Kappa score [27] for the questionnaire is 0.94, which can be interpreted as “almost perfect agreement” between human participants. These findings indicate CCTest’s effectiveness in enhancing code completion outputs.

Table IX: Distribution of “optimal” PSC transformations contributing to the enhancement when \( T = 9 \).

| Enhanced | BLEU on LeetCode (%) | BLEU on CodeSearchNet (%) | Edit Sim on LeetCode (%) | Edit Sim on CodeSearchNet (%) |
|----------|----------------------|---------------------------|-------------------------|------------------------------|
| PASS:    | 88.9                 | 86.6                      | 78.8                    | 76.9                         |
| FAIL:    | 21.2                 | 23.1                      | 21.2                    | 20.6                         |

Table VIII: Enhancement result when \( T = 9 \). Values denote improvement ratios against baseline, instead of absolute improvement amounts.

| Code Completion Systems | Copilot | CodePatrol | CodePatrol-small | GP-F-J | GP-F-NEU-1.1M | GP-F-NEU-1.2SM | Codegen-all-monot | Codegen-64f-monot | Avg. Enhancement |
|------------------------|---------|------------|------------------|--------|---------------|---------------|-------------------|-------------------|------------------|
| BLEU (%)               | 21.22 + 33.98 | 59.96 + 50.21 | 38.31 + 42.85 | 36.14 + 46.9 | 40.58 + 50.04 | 52.74 + 54.71 | 39.81 + 22.96 | 32.84 + 21.02 | 40.20 + 30.53 |
| Edit Sim (%)           | 0.27 + 5.89  | 4.77 + 38.77 | 56.74 + 64.45 | 53.41 + 65.74 | 87.93 + 80.16 | 102.3 + 82.8 | 100.69 + 110.84 | 52.81 + 72.93 | 37.48 + 67.17 | 62.01 + 72.85 |

Answer to RQ3: CCTest improve the outputs of different code completion systems. We also find that instead of one or a few PSC transformations that significantly enhance code completion, all schemes are shown as effective.

VI. DISCUSSION

Limitations and Threats to Validity. We now discuss the validity and limitations of this work. In this research, construct validity denotes the degree to which our metrics reflect the correctness of code completion systems. Overall, we conduct automatic testing and manual inspection to study the outputs of de facto code completion systems. Hence, while this practical approach detects their defects and reveals chances of enhancing their outputs, a possible threat is that our testing approach cannot guarantee the correctness of code completion systems. We clarify that our work roots the same assumption as previous testing works that aim to detect flaws of deep learning-based applications with dynamic testing rather than verification.

We check code completion outputs by comparing a set of outputs that are supposed to be (visually) consistent. The evaluation shows that the focus of structural consistency effectively unveils a large number of defects. However, a possible threat is that defects can be neglected in the completion output, in case all outputs share aligned yet erroneous code patterns. We deem this a general and well-known hurdle for invariant property-based testing techniques. We leave exploring solutions to this challenge for future work.
Besides, the potential threat exists that the proposed testing and enhancing framework, CCTEST, may not adapt to other types of code completion systems. Nevertheless, we mitigate this threat to external validity by designing a system and algorithm independent approach. As a result, our approach is anticipated to apply to other settings outside the current scope. We believe the proposed technique is general, and we give further discussions regarding other settings in this section.

**“Natural-Looking” Mutations.** CCTEST designs nine PSC transformations to mutate prompts. Nevertheless, one may question if all of these mutations are common in real-world programming. Holistically, we agree that some transformations, e.g., renaming variables, may lead to potentially rare names, e.g., using LocalVar1 as a variable name; as illustrated in under the REL_R scheme. Nevertheless, PSC transformations is intentionally designed as lightweight, for the seek of easing the follow-up outlier detection. We find that the present transformations are sufficient to expose many defects and harvest research insights. More importantly, we focus on designing structure-preserving transformations. Too aggressive transformations may easily break the structure consistency of mutated prompts, making our testing oracle inapplicable. Thus, while we also consider the specific program context (as noted in Table I) to enhance the “realism” of mutated prompts, realism is not our highest priority.

Overall, generating natural-looking, realistic inputs to test deep learning systems is inherently challenging. For instance, some computer vision (CV) related testing may need to use expensive generative models like GAN to generate more natural-looking images and test auto-driving systems [90]. It is unclear if those methods fit our scenarios. As a future work, we plan to study using advanced methods (e.g., LLMs) to generate mutated test inputs that are both “natural-looking” and structure-preserving. The cost may be a primary concern, as we generated around 20K mutated test inputs in the evaluation.

**Cross Comparison of Code Completion Outputs.** Overall, CCTEST individually tests each code completion system. The proposed approach constitutes program property-based testing (or metamorphic testing). With this regard, careful readers may wonder about the feasibility of conducting differential testing by processing the same prompt with different code completion systems and differentiating their outputs. However, we note that the code completion outputs can have drastically different representations since different code completion systems have different model training data and LLM model capacity. For instance, Copilot is seen to produce a large chunk of code snippets (with multiple statements), whereas some other well-known systems are prone to giving more succinct outputs for the same input prompts. Our preliminary exploration also shows that they manifest different tactics and translation templates in code generation. Thus, the similarity among the code completion outputs is deemed as low across different code completion systems. Overall, we leave it as one future work to explore practical methods to perform cross comparison, for instance, by extracting specific “semantics-level” signatures or regulating their output code patterns first.

**Other Settings.** CCTEST targets Python code completion, one challenging and popular task in software engineering. We believe it is feasible to migrate CCTEST to test code completion systems of other languages like C and Java. While extending CCTEST to handle other languages demands new parsers and re-implementing PSC schemes, we expect that the key technical pipeline, including mutation, outlier detection, and enhancing, are language independent. We thus believe migration is an engineering task rather than open-ended research problems.

**VII. RELATED WORK**

**Testing Neural Models.** Many works have applied testing methods to neural models [23, 51, 54, 55, 61, 73, 78, 87, 88, 90, 92]. The tested neural models include computer vision tasks like image classification, object detection [67, 72, 79], auto-driving [90, 92], as well as natural language processing tasks like sentiment analysis [29, 49, 65, 74], question answering [20], machine translation [35–37, 68], and specific properties like fairness [22]. Recent advances in machine translation testing inspire CCTEST. However, CCTEST addresses domain-specific challenges in mutating prompts, and for the first time, provides a systematic framework for testing code completion systems in black-box settings.

**Neural Code Comprehension.** Besides code completion task (the focus of CCTEST), neural models are used in other code comprehension tasks [13, 38, 62–64, 70, 84]. For instance, function naming decides the function name by summarizing the function body [10, 11, 94]. Often, code paths on the function ASTs are extracted for embedding and name prediction. Given the large volume of available paths, optimization schemes like attention are used to speed up the processing. Code classification and code search are two popular tasks. Recent methods learn from structural information (AST and CFG) for code classification [48, 50, 80] and code search [17, 32]. Tree-based convolutional neural networks and graphics neural networks are leveraged in these tasks [47, 75–77]. Software security applications have been built based on neural code comprehension, including plagiarism detection [43, 46], malware clustering [41], software component analysis [83, 86, 89], and vulnerability detection [45, 93]. CCTEST differs from these code comprehension tasks in that it focuses on testing code completion systems. While some recent works [13, 38, 62] treat neural models as a “white-box” for retraining and robustness augmentation on relatively simple DNNs, CCTEST delivers black-box repairing over LLMs.

**VIII. CONCLUSION**

We offer CCTEST, a testing and repairing tool for code completion systems. CCTEST uncovers thousands of defects, and its enhancement improves their output quality largely. This work may serve as a roadmap for researchers and users interested in using and improving code completion systems.

**ACKNOWLEDGEMENT**

We thank anonymous reviewers for their valuable feedback. HKUST authors are supported in part by a RGC ECS grant under the contract 26206520.
