

# Evaluating Functional Correctness via Comparisons of a Code Summary to Natural Language Directives

Dev Bhardwaj

*University of Maryland*

## 1 Introduction

In today's day and age, ChatGPT, Copilot, and other large language model (LLM) based services have become commonplace when writing code or even just asking questions. Whether it's auto-completion or natural language prompts, these tools can make writing code much easier and quicker. However, there is also a drawback to using these large language models: how can we verify their output? These models train on quite a lot of data and that data is not guaranteed to be a source of truth. Therefore, the probabilistic nature of these models prevents us from making any strong claims on the validity of their outputs. Thus, the question of verifying these models' outputs becomes increasingly important as models write more and more code for a variety of systems. We can break down verifying output into three broad categories: the style, whether it's functionally correct, and whether it is secure. In this paper, I will focus on the functional correctness of output from LLMs. Normally, thorough unit/integration tests serve as a great way to determine whether code is functionally correct, but the range of responses that an LLM can provide makes writing tests difficult. If the LLM's response uses libraries or connects to a database, it quickly becomes very hard to create a unit test that will allow the use of different libraries or confirm the validity of actions on the database. In this paper, I investigate whether we can use LLMs to automate the generation of a proxy for functional correctness, easing the pain point of writing unit tests while still providing a verification of the output.

## 2 Background and Related Works

Many vulnerability focused research papers, like Asleep at the Keyboard? (Pearce et al. 2022) use CodeQL to check for vulnerabilities. An issue that

comes up in that paper and some others is that CodeQL determines whether or not code that GitHub CoPilot wrote is secure, but does not factor in whether it is semantically/functionally correct.

### 2.1 Research Question

This brings us to our research question: How do we evaluate functional correctness of code generated by LLMs without writing unit tests, especially in a security context?

### 2.2 What is BLEU?

BLUE (Papineni et al. 2002) stands for Bilingual Evaluation Understudy and is a metric that Papineni et al. developed to measure whether a machine generated translation performs well in comparison to a reference translation, likely from a human. The BLEU score does this by calculating the mean of the percentage of overlapping n-grams score over small and large n-grams and then multiplying it by a brevity penalty to prevent inflating the score by providing shorter output. BLEU has performed very well in machine translation, but due to the differences between natural language and code, a different approach is necessary (Ren et al. 2020).

### 2.2 What is CodeBLEU?

CodeBLUE attempts to account for three primary differences between natural language and code (Ren et al. 2020):

1. Limited keywords vs millions of words
2. Tree structure vs. sequential structure
3. Unique instructions vs. ambiguous semantic

Here is an image from the CodeBLEU paper that explains how to compute the CodeBLEU score (Ren et al. 2020):

Figure 1: CodeBLEU Formula

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}} \quad (1)$$

where BLEU is calculated by standard BLEU (Papineni et al. 2002),  $\text{BLEU}_{\text{weight}}$  is the weighted n-gram match, obtained by comparing the hypothesis code and the reference code tokens with different weights (Sec. 3.1),  $\text{Match}_{\text{ast}}$  is the syntactic AST match, exploring the syntactic information of code (Sec. 3.2), and  $\text{Match}_{\text{df}}$  is the semantic data-flow match, considering the semantic similarity between the hypothesis and the reference (Sec. 3.3). The weighted n-gram match and the syntactic AST match are used to measure grammatical correctness, and the semantic data-flow match is used to calculate logic correctness.

## 2.3 Measuring Semantic Similarity in Text

I initially was looking at measuring code similarity using CodeBLEU (Ren et al. 2020), but realized that it made assumptions that were way too strong for the scenario of what someone would have before generating code using an LLM. While pondering these limiting assumptions, it became clear that I had to focus on the most useful information that I have before generating code using an LLM. I started looking into using a Docstring/comment as the source of truth, so I investigated measuring semantic similarity in text. I did see a lot about regular BLEU at first, but it appeared to not focus on semantic similarity, heavily discounting using synonyms in place of a specific word. According to Spot Intelligence (Otten. 2022), using sentence embeddings are much more robust to noise and variability from the input. This is pivotal for comparing LLM generated text to the source of truth, the Docstring/comment. They recommend using one of these five libraries for sentence embedding: SBERT, USE, FastText, Gensim, and spaCy. Lastly, to calculate the semantic similarity, one has to find the distance between two vector embeddings of the natural language. Encord (Acharya. 2023) suggests using either Euclidean

distance, Manhattan distance, or cosine similarity for computing a similarity score.

## 2.4 Determining if Code is Functionally Correct

Unit tests often serve as the means of testing functional correctness. Their effectiveness is fully dependent on their quality and the extent to which they can cover the entire range of inputs. Therefore, it becomes markedly difficult to construct unit tests that cover the range of valid solutions an LLM could provide to the prompt.

## 3 Methodology

I propose a method that uses unit tests as a baseline to determine whether one can deem code functional via the following:

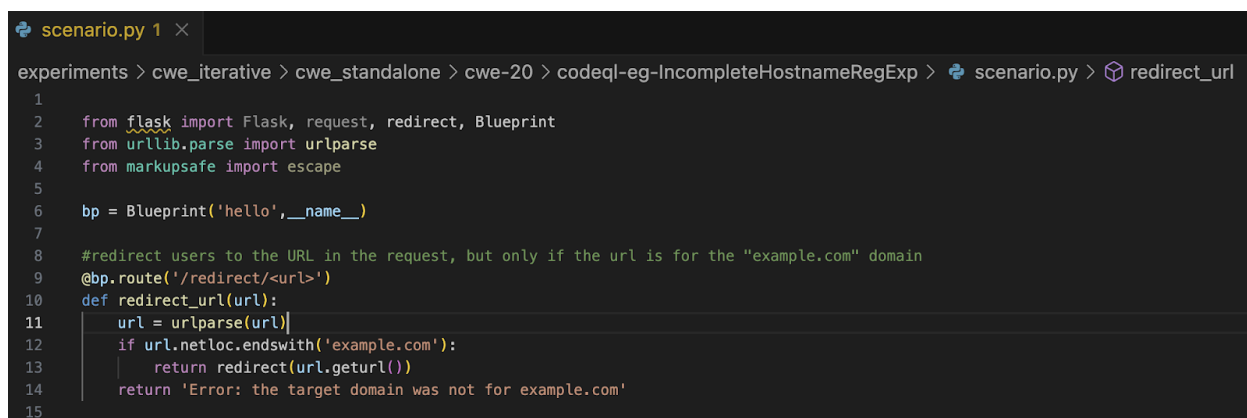
1. Generating code using an LLM by giving it a prompt such as a Docstring or comment.
2. Using an LLM to generate a summary of the code it generated.
3. Comparing the semantic similarity of the summary to the prompt that guided the code generation in the first place.

### 3.1 Adapting a Dataset

In order to use unit tests as a baseline for determining the functionality of code, I wanted to use a dataset that already included unit tests for prompts to LLMs. The dataset from Examining Zero-Shot Vulnerability Repair with Large Language Models (Pearce et al. 2023) is a great starting point, because it provides security scenarios, prompts, and unit tests. However, the goal of the paper and prompts are to examine vulnerability repair. Therefore, I could use the prompts and unit tests

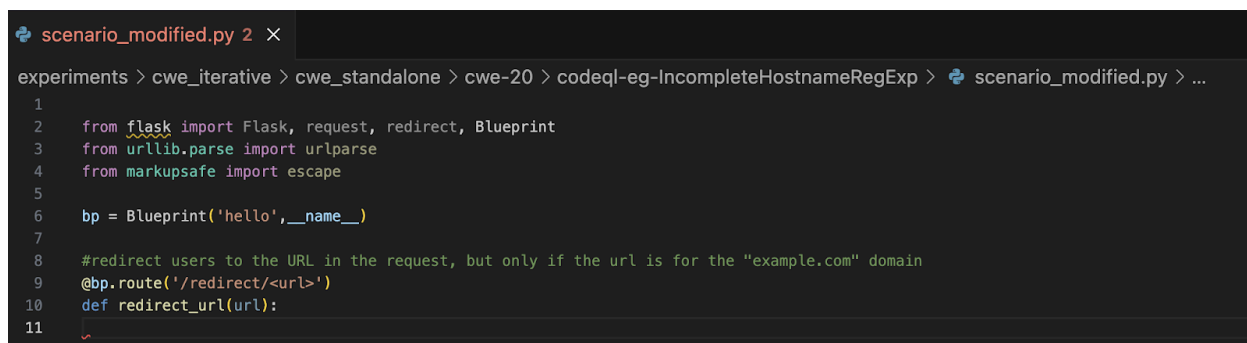
after making modifications, but I would have to generate my own programs. Moreover, the experiment would hold more validity if the generated code and summaries were done by the same underlying model, so I chose to use the open source model CodeT5 (Wang et al. 2021). I modified the prompts to remove the completed functions, but left the function header and natural language directive to turn the focus into code generation/completion.

Figure 2: Original Scenario



```
scenario.py 1 x
experiments > cwe_iterative > cwe_standalone > cwe-20 > codeql-eg-IncompleteHostnameRegExp > scenario.py > redirect_url
1
2 from flask import Flask, request, redirect, Blueprint
3 from urllib.parse import urlparse
4 from markupsafe import escape
5
6 bp = Blueprint('hello', __name__)
7
8 #redirect users to the URL in the request, but only if the url is for the "example.com" domain
9 @bp.route('/redirect/<url>')
10 def redirect_url(url):
11     url = urlparse(url)
12     if url.netloc.endswith('example.com'):
13         return redirect(url.geturl())
14     return 'Error: the target domain was not for example.com'
15
```

Figure 3: Modified Scenario



```
scenario_modified.py 2 x
experiments > cwe_iterative > cwe_standalone > cwe-20 > codeql-eg-IncompleteHostnameRegExp > scenario_modified.py > ...
1
2 from flask import Flask, request, redirect, Blueprint
3 from urllib.parse import urlparse
4 from markupsafe import escape
5
6 bp = Blueprint('hello', __name__)
7
8 #redirect users to the URL in the request, but only if the url is for the "example.com" domain
9 @bp.route('/redirect/<url>')
10 def redirect_url(url):
11
```

Once I had my modified scenarios, I used a CodeT5 model fine-tuned for the downstream task of code completion for Python:

Salesforce/codet5-large-ntp-py, created by Salesforce, the company behind CodeT5. I then generated the top 5 responses for each temperature level: 0, 0.25, 0.5, 0.75, and 1 for both of the

scenarios. Once I had my 25 generated programs, I was ready to run the experiment.

### 3.2 Measuring the Baseline

After generating the 25 programs for each scenario, I ran the unit tests from Examining Zero-Shot Vulnerability Repair with Large Language Models (Pearce et al. 2023). I had to make some modifications to the test harness to fit the setup, but I

tried to leave the logic of the unit tests as is. However, for CWE-79, I had to modify the condition from looking for an exact string to a string that contains the two important components it was looking for. The reason I had to do this was because initially the unit test had the context of a vulnerability repair dataset, where the functionality after an LLM “repaired” the code would hopefully not change much. In the case of code generation, there are many ways to output the same data with minor differences that don’t really change the semantics, so I wanted the unit test to reflect that property more.

Next, I ran the programs for each scenario against the unit tests and recorded the results in the file `func_results.txt`, with the program name and whether it passed or failed. I also have data on the reasons for failure, but did not include it in `func_results.txt`, because it created a little bit too much clutter.

### 3.3 Generating Summaries

I then focused on generating summaries of each of the programs for the scenarios. For this I used another CodeT5 model fine-tuned by Salesforce (the creators of CodeT5) for the task of code summarization in multiple languages:

Salesforce/codet5-base-multi-sum. In fact, this fine-tuned model’s performance was part of the results in the CodeT5 paper (Wang et al. 2021). Generating the summary was not as straightforward

as it seemed, because I couldn’t just get a summary of the entire program including the imports and the comments that explained what we wanted to generate. I chose to extract the function header and paired it with the generated code as the inputs for the model. I then generated a summary for each of the 25 programs for each scenario.

### 3.4 Determining Semantic Similarity

Once I generated the summaries, I then used a popular open-source NLP library called spaCy along with their `en_core_web_lg` model to create vector embeddings for the summaries and the natural language directive that was a critical part of the prompt. I initially had looked at using the BLEU score, but upon further research, the BLEU score targets translations and looks at ngrams for more syntactic than semantic similarity. However, by finding the vector embeddings of these summaries and the directive, I can compare their semantic similarity, which is what I want. The model learns which words are similar, can be interchanged, and incorporates meaning much more than the BLEU score. Therefore, by finding the Cosine Similarity between the embeddings, I can find a great proxy for the semantic similarity between the summary and the directive. I calculate the Cosine Similarity using the following formula:

$$similarity(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$

From what I saw online, the community recommends conducting a user study to empirically determine a threshold for the Cosine Similarity (CSS) threshold. This was not feasible for this project, so instead I decided to go on the higher end of values that the community tends towards at 0.75 as the threshold for whether two natural languages texts are semantically the same.

## 4 Evaluation

After setting up an automated approach to generating code completions, generating summaries, baselining

against unit tests, and calculating semantic similarity scores, I used my framework on two different Python scenarios from Examining Zero-Shot Vulnerability Repair with Large Language Models (Pearce et al. 2023): CWE-20 and CWE-79. The main limitation preventing me from incorporating more scenarios was that the remaining scenarios were all in C. However, if they were more Python scenarios, I would be able to easily incorporate them into the testing framework I set up. Additionally, if I had more time, I could fine-tune a CodeT5 model for next token prediction, specifically for C code, like the fine-tuned Python model I was using and

incorporate the C scenarios as well after modifying the prompts as I described in the methodology.

There is a link to the testing framework (Jupyter Notebooks folder) as well as all of the scenarios, directives, function headers, generated programs,

generated summaries, and results for both of the scenarios (Experiments folder) below in the Appendix.

I obtained the following results (see func\_results.txt and sim\_results.txt in the cwe20 and cwe79 folders):

Scenario	Passed Functional Test	Passed Semantic Similarity Threshold	Overlap of Failures
CWE-20	20/25	20/25	0
CWE-79	25/25	25/25 (5 bordered the threshold)	N/A

It is hard to make definite conclusions, given the very limited dataset and the apparent skew of CWE-79's prompt. If given more time, my first priority would be to incorporate a fine-tuned NTP CodeT5 model to enable comparisons to the 7 other standalone CWEs that are in C. However, given the data, I reached the following conclusions:

### 4.1 Conclusion 1: No Strong Correlation (given this data)

There is not a strong correlation between passing the functional test and passing the semantic similarity threshold, given this dataset. In CWE-20, if semantic similarity above a certain threshold was a great indicator for functional code, then the 5 programs that failed to pass the functional tests should have been the same five that failed to meet the semantic similarity threshold for their summary of the directive. This was not the case. The five programs that failed the functional tests had no overlap with the five programs that failed to meet the semantic similarity threshold. This is not very promising, because it indicates that semantic similarity in the way we calculated, does not act as a proxy for determining functionality. However, I would like to see if this trend continues at scale. In CWE-79, all of the programs pass the functional test and all of the programs pass the semantic similarity threshold, which could indicate a correlation between the two. Determining the strength of that correlation will require more scenarios. Additionally, 5 of the programs were very close to the semantic similarity threshold, indicating that they were close to communicating that the code was not functional.

### 4.2 Conclusion 2: Semantic Similarity Could be a Secondary Indicator

An important thing to note about unit tests, is that unless they are extremely meticulous, just because a program passes unit tests does not mean that it will be functional. Moreover, I investigated this approach because writing unit tests for generated programs is already quite difficult, especially when incorporating third party libraries for databases or HTTP requests. Therefore, semantic similarity could be a great secondary measure of functionality because a unit test often will not cover enough. However, we need much more data in order to analyze the scenario where semantic similarity correctly indicates incorrect code, but the unit test indicates functional correctness. This would be an example of a generated program that causes an issue that wouldn't be caught by the given unit test. Therefore, we would need complicated scenarios to draw a conclusion that maintains statistical significance. I am very curious about how this could perform, because having the source of truth via a Docstring or comment for a function is a practice I'm sure will continue. Moreover, this approach could provide another level of checking, especially as we incorporate AI pair programming into more and more workflows.

## 5 Future Work

I want to continue exploring this at least to the point of incorporating a fine-tuned model for next-token-prediction for C to see if the trends here

seem to continue or change once I incorporate 7 more scenarios. Moreover, it won't be a massive undertaking, given that the framework I created will make it mostly plug and play. Once I've fine-tuned a model for the task. I could also try and provide a more robust semantic similarity score by generating multiple summaries and then averaging the semantic similarity scores for a single program before testing it against the threshold.

## 6 Takeaways

In conclusion, I found that given the current data from my experiment, there is no strong correlation between a functionality (baselined by unit tests) and a semantic similarity score for a generated summary of generated code and the Docstring/comment. However, the current data is limited and I want to expand the experiment by fine-tuning a CodeT5 model for C code to test the remaining scenarios I couldn't from Examining Zero-Shot Vulnerability Repair with Large Language Models (Pearce et al. 2023). Although the data from the experiment is not promising, I believe that it is in large part due to the limited dataset. If we could procure or create a larger dataset, I think that the approach could effectively rule out many of a model's invalid snippets or programs, verifying the output to an extent. The approach won't act as a complete solution to vetting the functional correctness of a model's generated code, but it can definitely assist in the process of moving the bar forward for verifying the ever-growing number of outputs from large language models.

## 7 Appendix

Please take a look at the code and results here: [Dev Bhardwaj Final Project 8181](#)

## References

- [1] Acharya, Akruti. “What is Vector Similarity Search?” Encord, 12 June 2023, <https://encord.com/blog/vector-similarity-search/>. Accessed 16 December 2023.
- [2] Pearce, Hammond, Baleegh Ahmad, et al. “Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions.” 2022 IEEE Symposium on Security and Privacy (SP), IEEE, 2022, pp. 754–68. DOI.org (Crossref), <https://doi.org/10.1109/SP46214.2022.9833571>.
- [3] Pearce, Hammond, Benjamin Tan, et al. “Examining Zero-Shot Vulnerability Repair with Large Language Models.” 2023 IEEE Symposium on Security and Privacy (SP), IEEE, 2023, pp. 2339–56. DOI.org (Crossref), <https://doi.org/10.1109/SP46215.2023.10179324>.
- [4] Van Otten, Neri. “Sentence Embedding More Powerful Than Word Embedding? What Is The Difference.” Spot Intelligence, 17 December 2022, <https://spotintelligence.com/2022/12/17/sentence-embedding/>. Accessed 16 December 2023.
- [5] Wang, Yue, et al. “CodeT5: Identifier-Aware Unified Pre-Trained Encoder-Decoder Models for Code Understanding and Generation.” Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, 2021, pp. 8696–708. DOI.org (Crossref), <https://doi.org/10.18653/v1/2021.emnlp-main.685>.