Beyond the Exact Match: Investigating the Relationship Between Syntactic and Semantic Equivalence in Human and LLM Test Assertions.

Ruben van der Giessen[1]
Supervisor(s): Annibale Panichella, Mitchell Olsthoorn
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Ruben van der Giessen
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Alexios Voulimeneas

Abstract

Test assertions form a critical component of software tests, as they are the component that actually verifies whether the code under test is exhibiting the desired behaviour. However, writing test assertions is time consuming, and thus research has been carried out on how to help in automation of this task. Since the emergence of Large Language Models (LLMs) in recent years, interest in their application for assertion generation has grown. LLMs have shown promise, with LLM-generated assertions achieving mutation scores similar to human-written assertions. However, existing research evaluates the assertions based on either exact matches or mutation scores in isolation, thus not investigating the relationship between syntactic and semantic equivalence. This matters because syntactically different assertions can have the same semantics. Punishing the LLM for writing `assertEquals(a, b)` instead of `assertTrue(a.equals(b))` leads to systematically under-reported LLM performance.

In this paper we investigated the extent to which LLM-generated assertions differ syntactically but remain semantically equivalent to human-written reference assertions. We construct a dataset with 177 filtered entries drawn from open source projects and generate assertions using gpt-oss-20b. We then measure the syntactic similarity via normalised tree edit distance and related metrics. We approximate the semantic similarity based on Jaccard and Ochiai similarity between the sets of mutants killed with PIT mutation testing. We find a moderately strong correlation between normalised tree edit distance and the Jaccard similarity of the killed mutants ($\rho = -0.685$, $p < 0.001$), indicating that the two metrics are related but not interchangeable. Open coding of 41 semantically equivalent but syntactically different pairs revealed ten transformation categories. The LLM showed a universal preference for the omission of assertion messages and for replacing boolean checks with equality assertions. We use open coding to evaluate the syntactic differences between semantically equivalent assertions. Finally we use a decision tree to generate a threshold allowing us to effectively distinguish between datapoints likely and unlikely to be semantically equivalent. We find this threshold to be 0.41 for the normalised tree edit distance, showing a median Jaccard similarity of 0.5290 below it and a median of 1.000 above it. Our findings suggest that exact match evaluation significantly underestimates LLM assertion generation performance, and that syntactic similarity with a fixed threshold offers a more useful metric for assertion quality.

1 Introduction

Software testing has an important role in the software development lifecycle. It is critical for improving correctness and reliability of software systems, allowing developers to discover problems early on. In automated software testing, developers usually write test cases that consist of input data, method calls and assertions. These assertions act as test oracles, encoding correct system behaviour. Given an input to a system, the challenge of determining whether the output is correct or not is what is known as the *oracle problem* [2]. Automation of test oracles is seen as a bottleneck inhibiting further test automation.

Research into automating test generation has long been performed, with varying strategies. A recent comparative study by Abdullin et al. [1] compared search-based software testing (SBST), symbolic execution and LLM-based approaches for generating unit tests. They find that LLMs outperform the other techniques in mutation score, but lack in fault detection. This indicates that the assertions generated by LLMs are still lacking, motivating further research into LLM assertion generation.

Two broad evaluation strategies have emerged in research papers on LLM assertion

generation. One strategy entails looking for exact string matches between the LLM and human reference assertions. This approach is followed by Primbs et al. [6] and Watson et al. [9]. AsserT5 [6], a fine-tuned language model achieved an exact match rate of 59,5%, but was only able to catch 33 out of 138 bugs in a dataset with real bugs. This gap suggests that exact match rate might not be a good metric for assertion quality.

The second strategy is based on mutation testing. Mutation testing works by introducing small changes to the original source code (mutations) and checking whether the assertions can detect them. Because the mutations can detect behaviour of the assertions, the set of killed mutants can be used as a proxy for the semantics of the assertion set. Molinelli et al. [5] follows the mutation score approach, and they find that LLMs can achieve mutation scores comparable to those of humans. The LLM-generated assertions obtained a mutation score of 43%, while the human reference assertions achieved 45%.

These two groups leave an important gap, as neither investigates the relationship between syntactic similarity and semantic equivalence. This matters because two assertions might be identical semantically, while looking very different (e.g. `assertTrue(a.equals(b))` and `assertEquals(a, b)`), causing an exact match evaluation approach to unfairly penalise the LLM. The reverse is also true, as a small syntactic change can lead to very different semantics, like replacing `assertEquals` with `assertNotEquals`. Understanding when syntactic difference leads to semantic divergence is important for both developers employing LLM-based solutions and researchers evaluating them.

Therefore, this paper evaluates the following research question: "To what extent do LLM-generated assertions differ syntactically but remain semantically equivalent to reference assertions?", which we divide into the following sub-questions:

- **RQ1**: *What is the correlation between syntactic and semantic equivalence?* We measure the syntactic and semantic similarity using several metrics. We then calculate their correlation coefficients to get an idea for how effectively one can be used to predict the other.
- **RQ2**: *What types of syntactic transformations does the LLM often apply, while preserving semantic equivalence?* We analyse semantically equivalent but syntactically different assertion pairs to see if the LLM systematically rewrites assertions in a certain manner, like opting for `assertTrue(a.equals(b))` over `assertEquals(a, b)` to find which transformations are common and safe.
- **RQ3**: *At which threshold for syntactic equivalence do the LLM-generated assertions generally fail to be semantically equivalent?* We derive a threshold for syntactic equivalence from the dataset that can be used to predict whether assertions are (un)likely to be semantically equivalent, allowing for practical large-scale filtering.

Our main contributions are:

- The identification of a relationship between syntactic and semantic equivalence, allowing evaluation to move past simple binary metrics like exact matches to more practical and realistic measures.
- Analysis of a collection of common syntactic transformations often applied by LLMs, without harming semantic equivalence.
- Data-driven syntactic similarity threshold for distinguishing semantically equivalent from non-equivalent assertions pairs.

- A replication package.[1]

2 Background and related work

When testing a system, test oracles may be used to verify whether the system is behaving as expected. They describe the correct output based on the input to the system. Test oracles can take many shapes and forms. For example, a test oracle could be a human or an assertion.[2]

Given an input to a system, the challenge of determining whether the output is correct or not is what is known as the oracle problem.[2] Automation of test oracles is seen as a bottleneck inhibiting further test automation.

Research into automated test assertion generation has been going on for years. An example of an older work is the work done by Watson et al. [9], where the authors introduce a system for automatically generating test assertions called ATLAS (AuTomatic Learning of Assert Statements). This model is based on a Neural Machine Translation approach and was trained to predict JUnit assertion statements based on pairs of test prefixes and focal methods. The metric used to evaluate the model is exact string matches with the original reference assertions. Its top-1 prediction achieved an exact match 31% of the time.

A similar, newer work is that of Primbs et al. [6]. Here, the authors created a model called AsserT5 by fine-tuning the pre-trained CodeT5 model. The model shows a large improvement over ATLAS, achieving a 59.5% exact match rate. However, when evaluated on a dataset with known bugs the model's assertions were only able to catch 33 out of 138 bugs. This shows a large discrepancy between syntactic matches and performance in actual application.

Molinelli et al. [5] investigates the effectiveness of LLMs for generating test assertions. Differently from the previous two papers, they evaluate the assertions based on mutation scores, comparing them to the mutation scores achieved by humans. Their results showed the LLM assertions gaining a mutation score of 43%, similar to that of humans, which was 45% in the study. This indicates that LLM assertions can rival human assertions in mutation score, and thus potentially in semantic strength, without necessarily matching the original human assertions syntactically. This begs the question as to what extent syntactic similarity relates to semantic equivalence.

3 Methodology

3.1 Dataset

We base our dataset on the GitBug-Java [8] dataset, a collection of Java projects with bug-fixing commits. We chose GitBug-Java because it is a recent (2024), diverse dataset of compilable projects with human-written test suites, and has been used in research before. Other options were not chosen as they do not fill all these criteria, like being very old (Defects4J) or not used in other research (a custom set of projects). Another reason to base our dataset on an existing dataset is to save time.

To simplify our benchmark we only used projects that are built using Maven, use JUnit 4 or 5, and only use assertions written in pure JUnit syntax (i.e. no AssertJ or Hamcrest).

[1] https://github.com/parsnip9755/thesis-replication-package

Table 1: Project counts in dataset before and after filtering.

Project Name	**Before Filtering**	**After Filtering**
ari-proxy	18	5
aws-secretsmanager-jdbc	100	–
beanshell	100	12
cbor-java	87	3
chesslib	60	26
cloudsimplus	100	13
crawler-commons	100	21
dataframe-ec	49	–
epubcheck	100	7
formatter-maven-plugin	16	–
grammaticus	51	10
jansi	18	–
jsoup	100	50
markedj	22	10
nbvcxz	22	–
urnlib	91	13
whatsapp-business-java-api	79	–
word-wrap	43	7
Total	**1,156**	**177**

We created the dataset by downloading the selected projects' code, and running a Python script that processes all files ending with `Test.java`, performing the following steps:

1. Check for a corresponding non-test Java file using the simple heuristic of looking for a matching file without the "Test" suffix in the main package. If this file cannot be found, skip test file.
2. Find all methods annotated with `@Test`. Then do the following steps for each test.
3. Find all assertions in the test method. Replace them with placeholders and store the original assertions in a list.
4. If there are no assertions, skip the test.
5. Find which methods in the class under test are called (focal methods).
6. Find string literals in the test method that point to a file. Check whether the files are smaller than 1000 bytes and have text content (extra context).
7. Put the final datapoint together, saving the project name, file path to the test file, original test case, test case with assertions replaced with placeholders (template test case), the original list of assertions, the methods called in the class under test, and the contents of files discovered in the previous step.

To improve diversity, we sample at most 100 entries per project (before filtering). Table 1 shows the final distribution of projects in our dataset after filtering (which is described in 3.4.1).

3.2 Assertion Generation

To generate the LLM assertions for each dataset entry we used the gpt-oss-20b model from OpenAI, served from the cloud through Ollama. We selected this model as it is a relatively capable open-weight model that is large enough to be useful, but not too big to reduce the chance of data leakage. For each dataset entry, the LLM was given the following prompt:

```
You are an expert Java developer writing JUnit assertions.

Below are the focal methods (the methods under test):
```java
{focal_methods}
```

Extra context (additional source files):
{extra}

Here is the test case template containing {n_assertions} assertion placeholder(s):
```java
{template}
```

Your task:
Generate Java assertions to replace the placeholders <ASSERTION_0>, <ASSERTION_1>, etc.
- Each assertion must be a valid Java expression without a trailing semicolon (the template already has one).
- Use only JUnit-style assertions (e.g., assertEquals, assertTrue, assertFalse, assertNull, fail, etc.).
- Make sure assertions use the correct variables from the template.

Return ONLY a JSON object in this exact format:
{"assertions": ["assertEquals(...)", "assertTrue(...)"]}
Do not wrap the JSON in markdown code blocks.
```

When generating the assertions we used a fixed seed and a temperature of 0 in an attempt to ensure determinism.

3.3 Evaluation Metrics

3.3.1 Syntactic Similarity

We consider a few types of metrics for syntactic similarity. In the case of a datapoint having multiple assertions, the assertions are first combined into one string by merging the assertions into a single string with a semicolon as separator.

- **Text edit distance**: We compute both the absolute and normalised (divided by length of longer string) Levenshtein distance. The normalised variant makes the metric comparable between assertion pairs of different lengths. We include this metric as it is widely known and easy to calculate.
- **Tree edit distance**: We compute the edit distance between the abstract syntax trees of both assertion strings using the Zhang-Shasha algorithm. [10]. Similarly to the text edit distance, we calculate both the absolute and normalised edit distance. We include this metric as we expect it to be similar to the text edit distance, while being less sensitive to formatting differences, making it more suited for determining the syntactic differences.

- **CodeBLEU** [7]: We also compute a more advanced metric combining several techniques. We include it to see how a more advanced metric specifically made for automatic evaluation of generated code performs.

Of these, we pick the normalised tree edit distance as our primary syntactic metric as it is a simple, known metric that captures syntactic difference effectively while ignoring formatting changes and remaining comparable among entries.

3.3.2 Semantic Similarity

As it is not possible to directly measure the actual semantic similarity between two sets of assertions we instead use a proxy metric to approximate the similarity. We do this through comparing the set of mutants killed by the assertions using mutation testing with PIT (Pitest) [3]. We use the default set of mutators as this set was carefully chosen to avoid identical mutants and to save time when running our mutation tests.

- **Jaccard similarity of killed mutants**: This metric was chosen as the Jaccard index is a standard metric of the similarity between two sets.
- **Ochiai similarity of killed mutants**: This was chosen as it is a common alternative to the Jaccard index that is less sensitive to large size differences between the sets being compared.

We use the Jaccard similarity as our primary semantic metric because it is the more known metric between the two.

The process applied for every datapoint can be described as follows:

- Find the test file and remove all test methods (`@Test`, `@ParameterizedTest`, `@RepeatedTest`, `@TestFactory` and `@TestTemplate`) that are not the target method from the test case.
- Save this modified code to a new file.
- Perform mutation testing using PIT.
- Read the generated CSV and save the set of mutants killed.
- Replace the test method with the version with LLM assertions.
- Perform mutation testing using PIT.
- Read the generated CSV and save the set of mutants killed.
- Compute the Jaccard and Ochiai similarity between the sets of mutants killed by both methods.

3.4 Analysis

3.4.1 Filtering

Before analysis we remove entries that cannot be used:

- Both the human-written reference and LLM-generated assertions must compile and pass. If the test does not compile or pass we cannot derive the mutations killed and thus cannot measure the semantic similarity.

- The reference assertions must kill more than 10 mutants. We did this because entries with very few mutants are not reliable: when there are only two mutants just one not being killed has a massive impact on the Jaccard similarity.

After filtering the final dataset contains 177 diverse entries from 12 different projects. See Table 1 for a per-project overview.

3.4.2 RQ1 Correlation Between Syntactic and Semantic Equivalence

To evaluate the relationship between syntactic and semantic equivalence we calculate the coefficients of correlation between all syntactic and semantic metrics. We consider two methods of calculating the coefficient of correlation: Pearson correlation coefficient and Spearman's rank correlation coefficient. We chose between these two based on the distribution of the data. If based on the Shapiro-Wilk test a normal distribution cannot be assumed ($p < 0.05$), we use Spearman, otherwise we use Pearson. We do this because Spearman makes no assumptions about the distribution of the data and is robust to non-linear relationships.

3.4.3 RQ2 Syntactic Transformation Taxonomy

To characterise the syntactic differences between human reference and LLM-generated semantically equivalent assertions we first filter our dataset for entries where the assertions are not an exact match but the Jaccard similarity equals 1, meaning that the assertions killed the exact same set of mutants. For the purposes of analysis we assume that these entries are actually semantically equivalent.

We then perform open coding on the assertion pairs, rather than assigning from a set of pre-defined categories. Open coding is the appropriate choice as our goal is exploratory: we want to find out which transformations the LLM applies, rather than confirm a hypothesis. If we used a pre-defined set of categories we risk missing categories we did not anticipate beforehand. We go over our entries and create and assign categories as we go. An entry can be assigned to multiple categories as multiple transformations might be present.

3.4.4 RQ3 Threshold for Syntactic Equivalence

To identify a threshold for the syntactic similarity below which semantic equivalence is unlikely to hold we considered two possible methods: piecewise linear regression and a decision tree. Piecewise linear regression requires the data to be piecewise linear and homoscedastic. However, we expect our data to be quite chaotic as a small change (e.g. `==` to `!=`) can entail a large semantic shift. Thus, we believe the decision tree to be a better choice, as it makes no assumptions on the data distribution. We use the decision tree with a maximum depth of 1 to yield a single threshold segmenting the data into a low and a high semantic equivalence zone. We report the median and standard deviation of both segments to evaluate the usefulness of the threshold.

4 Results and Discussion

4.1 RQ1: What is the correlation between syntactic and semantic equivalence?

Results. Figure 1 shows a scatterplot of syntactic similarity (1 − normalised tree edit distance) against the semantic equivalence (Jaccard similarity of killed mutants), with a

trend line fitted to the datapoints. The scatterplot shows a positive trend, pairs with higher syntactic similarity tend to also have a higher semantic similarity. However, we do see significant scatter, particularly around the center for the syntactic similarity (0.2 – 0.8), where the Jaccard similarity values span the full range from 0 to 1. Above 0.8 we see consistently high Jaccard similarity, while below 0.2 the data remains chaotic.

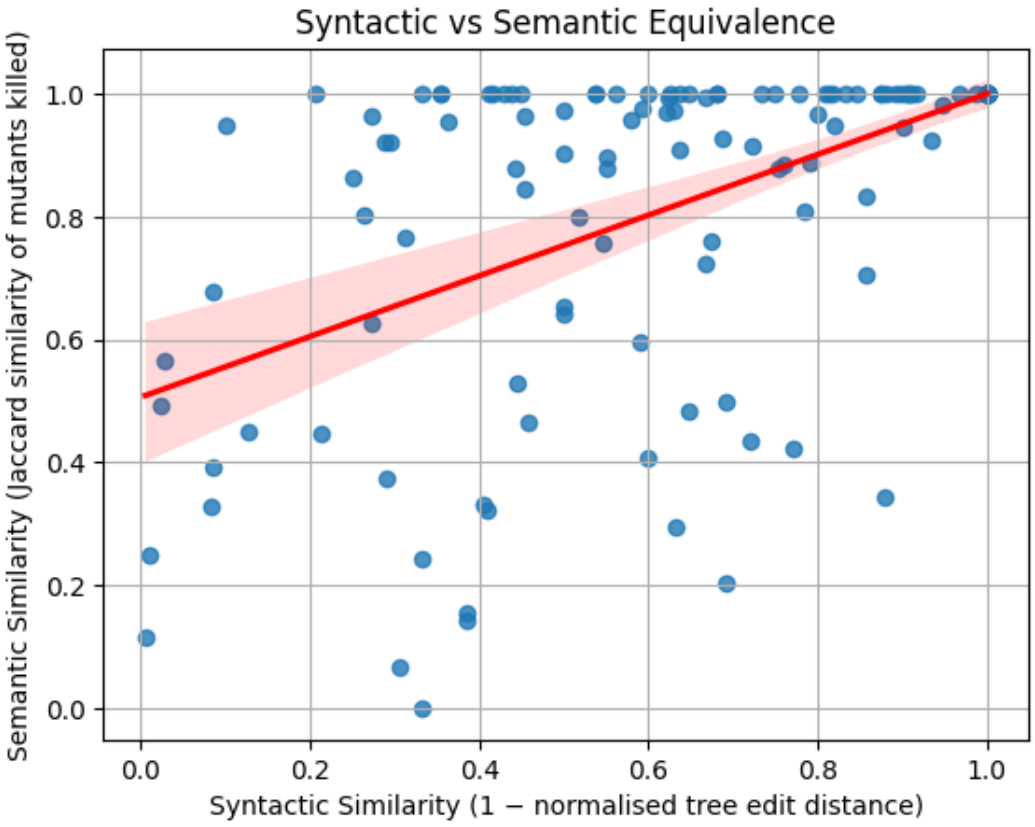

Figure 1: Scatterplot of syntactic and semantic similarity pairs. Red line indicates the trend.

Table 2 shows the Spearman rank correlation coefficients between all syntactic metrics (rows) and semantic metrics (columns). As described in the methodology, Spearman's rank correlation was used instead of Pearson's because the data is not distributed normally. All metric pairs show a correlation with an absolute value ranging from 0.637 to 0.687, suggesting our finding is not sensitive to the choice of syntactic or semantic metric. The normalised variants of the text and tree edit distance correlate slightly more strongly than their absolute variants, which is expected as they account for assertion length. The strongest correlation can be found between normalised text edit distance and the Jaccard similarity at $\rho = -0.687$. The normalised tree edit distance to the Jaccard similarity follows as a close second at $\rho = -0.685$ ($p < 0.001$). CodeBLEU shows a positive coefficient, as it is a similarity metric instead of a distance metric. Surprisingly, CodeBLEU, an advanced metric specifically developed for evaluating generated code seems to show the weakest correlation ($\rho = 0.638$). There seems to be no large difference between Jaccard and Ochiai similarity.

Table 2: Spearman correlation coefficients between syntactic metrics (rows) and semantic metrics (columns).

Syntactic metric	**Semantic metric**	
	Jaccard	Ochiai
Text edit distance (absolute)	−0.653	−0.652
Text edit distance (normalised)	−0.687	−0.686
Tree edit distance (absolute)	−0.648	−0.647
Tree edit distance (normalised)	−0.685	−0.683
CodeBLEU	0.638	0.637

Discussion. A correlation coefficient of $\rho = -0.685$ indicates a moderately strong relationship between syntactic and semantic similarity (with Jaccard similarity as a proxy for semantic similarity). This confirms that the two are meaningfully connected. Decreasing syntactic similarity tends to mean a smaller overlap between mutants killed. However, the substantial scatter visible in Figure 1 shows that syntactic similarity cannot serve as a reliable predictor for semantic equivalence, instead only being able to indicate whether the assertions are more or less likely to be semantically equivalent.

The consistency of the magnitude of the correlation coefficient across metrics indicates that the relationship is not a result of the chosen metrics, but an underlying property of the relationship between syntactic and semantic similarity. The slight but consistent advantage of normalised edit distance over absolute edit distance points to the importance of accounting for assertion length.

4.2 RQ2: What types of syntactic transformations does the LLM often apply, while preserving semantic equivalence?

Results. Of the 177 datapoints, 93 have a Jaccard similarity of 1. Of these, 41 are also syntactically different and were used for open coding. The resulting categories and the amount of datapoints per category is shown in Table 3.

Assertion set restructuring is the largest group. This covers cases where the LLM generates a different set of assertions that performed the same checks. Test input value substitution is the second largest group with 6 datapoints, followed by assertion message removal, boolean predicate to equality assertion and assertion reordering with 5 datapoints each.

In some categories the LLM shows a consistent preference, while in others it doesn't. In every datapoint where the human had added a message to the assertion, the LLM omitted it. Similarly, the LLM seems to prefer equality assertions (e.g. `assertEquals(a, b)`) over boolean checks (e.g. `assertTrue(a.equals(b))`). However, the LLM does not seem to exhibit any specific preference for equivalent method usage or assertion ordering. A worthy mention for the test input value substitution cluster is that four of the points came from a single project, where the behaviour was the same each time: simplification of an absolute URL to a relative one.

Discussion. The universal removal of assertion messages might indicate a preference towards minimalism, preferring to omit the unneeded as the messages do not affect any semantics. We could interpret the preference for equality assertions over boolean checks as a

Table 3: Overview of transformations applied by the LLM. Pairs may carry multiple labels; total instances therefore exceed the number of datapoints.

Category	LLM preference	Count
Assertion set restructuring	No consistent preference	8
Test input value substitution	LLM tends to use simpler values (e.g. relative paths instead of absolute URLs)	6
Assertion message removal	LLM consistently omits assertion messages	5
Boolean predicate → equality assertion	LLM prefers explicit equality checks	5
Assertion reordering	No consistent preference	5
Equivalent API methods (e.g. `first()` ↔ `get(0)`)	No consistent preference	4
Whitespace or formatting only	—	4
Other	—	3
Argument order reversal (e.g. `assertEquals(a, b)` ↔ `assertEquals(b, a)`)	LLM prefers `assertEquals(expected, actual)`	2
Static import vs. qualified name (e.g. `RFC_2141` vs. `RFC.RFC_2141`)	LLM prefers fully qualified name	2

preference for more informative assertions.

The test input value substitution where relative URLs were used instead of the original absolute ones shows that our proxy metric is not perfect, as semantically different assertions can still kill the same mutants.

4.3 RQ3: At which threshold for syntactic equivalence do the LLM-generated assertions generally fail to be semantically equivalent?

Results. As we can see in Figure 1 our data is chaotic and heteroscedastic, confirming our hypothesis and choice for using a decision tree in the methodology. Applying the decision tree as described yields a threshold of 0.41 for syntactic similarity (measured as $1 -$ normalised tree edit distance). Figure 2 shows the resulting segmentation of datapoints using this threshold.

Below the threshold, the mean Jaccard similarity is 0.5712 and the median is 0.5290, with a standard deviation of 0.3298. Above the threshold, the mean rises to 0.9230 and the median to 1.000, with the standard deviation dropping to 0.1678. The reduction in variance is particularly notable. Not only is the semantic similarity higher on average above the threshold, but the predictions are also considerably more consistent.

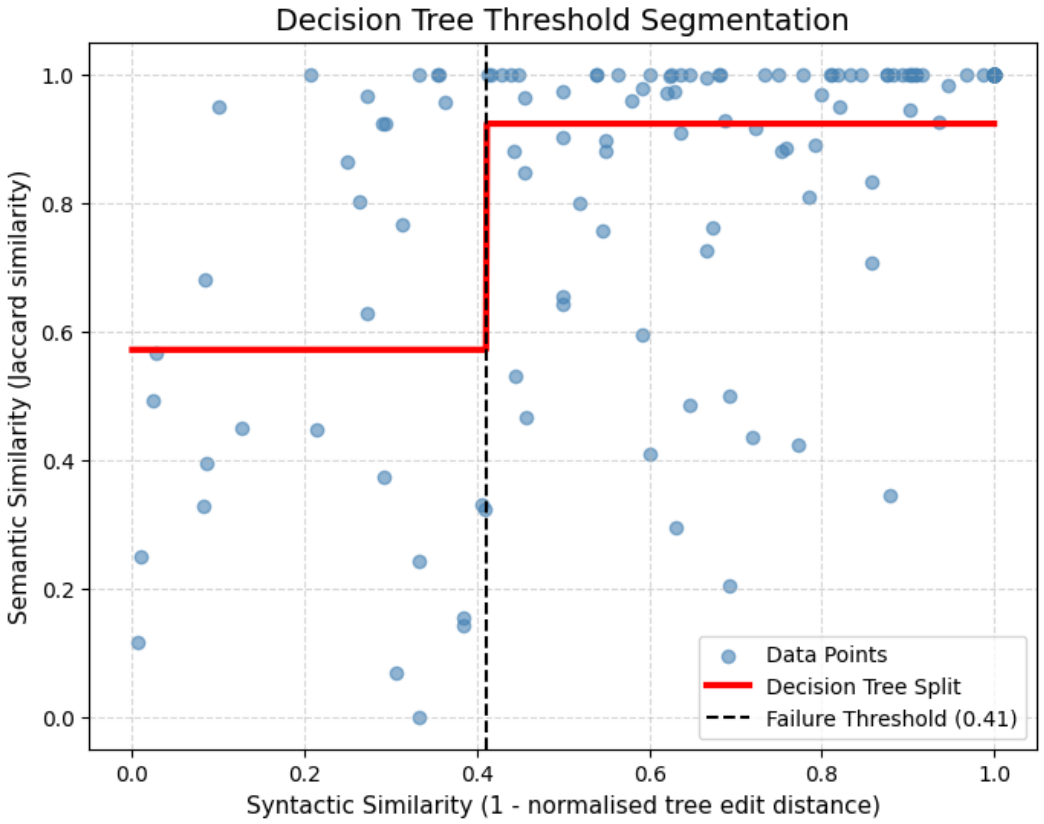

Figure 2: Segmentation of datapoints to identify a threshold. The vertical dashed line indicates the decision tree split at syntactic similarity 0.41.

Discussion. The threshold of 0.41 provides a practically useful boundary for distinguishing assertions likely to be semantically equivalent from those that are not, where Jaccard similarity is used as a proxy for semantic equivalence. A median of 1.000 above the threshold indicates that for the majority of datapoints above the threshold, they kill the exact same set of mutants. The halving of the standard deviation also indicates a considerably more reliable signal above the threshold.

The threshold however is not a hard boundary. There is a cluster of points with low semantic similarity above the threshold in Figure 2. This demonstrates that while a point being above the threshold indicates a higher chance of semantic equivalence, it does not guarantee it. Similarly, some points below the threshold are regarded as semantically equivalent. Because of this, how to balance these two sides must be considered before use.

Especially below the threshold we see very large variance, with both points with very high and very low semantic similarity being present. This means that our predictive power is lower in this region. This reinforces our conclusion from RQ1, that while there is a relationship between syntactic and semantic similarity, they are not interchangeable.

5 Responsible Research

5.1 Ethical

All code repositories mined for the creation of our dataset are licensed under an OSI-approved open source license, allowing use in academic research. Before use, code repositories are modified minimally to add the newest version of PITest.

A major concern for our research is data leakage [4]. The LLM we used for our experiment, gpt-oss-20b may have been trained on the same repositories as we based our dataset on as the training data for gpt-oss-20b is private. In such a case, we risk the AI simply memorising these assertions possibly leading to overly positive results. To reduce the likelihood of memorisation we have opted for a smaller model with 20 billion parameters to reduce the likelihood of memorisation.

Another aspect to note is the environmental impact of LLM inference and running of mutation tests performed during our experiment. While we did not measure the exact energy usage, we admit that this type of research can have an impact on the environment.

5.2 Reproducibility

To support the reproducibility of our experiment we provide the dataset and full code with dependencies pinned to fixed versions in a public repository. All versions of external tools (e.g. Java and PITest) are also provided.

The largest problem for the reproducibility of our experiment is LLM non-determinism. When an LLM is asked the same question multiple times, it usually returns a different answer every time. For our experiment we used gpt-oss-20b via Ollama with a fixed seed and the temperature set to 0. However, we noticed that despite this the LLM still did not return consistent results, likely due to floating-point non determinism. To reduce the impact of this, we include the LLM generated assertions in our dataset.

5.3 Generative AI usage

Generative AI was used to assist with drafting and editing parts of this paper. No AI-generated content was used verbatim, and was checked thoroughly before use. All claims were independently checked by the author.

6 Threats to validity

6.1 Construct validity

A major threat to construct validity is how we measure semantic equivalence. As it is very difficult to directly measure and quantise the semantic equivalence between two sets of assertions, we need to use some kind of proxy to approximate it instead. While the Jaccard similarity of the sets of mutants killed by the two sets of assertions can give a good indication as to what degree the two sets behave similarly, it does not guarantee semantic equivalence. Sets of assertions might kill identical mutants, yet behave differently for cases not covered by the mutants. Another linked threat is that the set of mutations applied might influence the results. We used PIT's default set of mutations, meaning that any behaviour resulting from mutations not included in the default set are invisible to our metric.

To our syntactic similarity metric applies a similar concern. While (normalised) textual and AST-based edit distance gives a decent idea of the syntactic similarity between two sets of assertions, it is not flawless. Two pairs of sets of assertions could have the same edit distance, while being quite different in reality. To mitigate this we calculate the syntactic similarity using multiple metrics, and verify that our findings hold regardless of the metric used, suggesting our findings are not merely because of the metrics used.

6.2 Internal validity

For assertion generation a singular fixed prompt was used. Different prompts may lead to different syntactic transformations to be applied, or differing behaviour of the generated assertions. This might lead to our results reflecting one specific prompting strategy, instead of LLM behaviour in general. To support transparency and replication we include the full prompt in Section 3.2 and include it in the replication package.

Another threat to internal validity is that before analysing the results we filtered out all datapoints with assertions that did not compile or pass. This is a necessity as we cannot measure killed mutants if the code doesn't compile or pass. However, the filter may introduce a bias by removing tests with unconventional control flow or complex setup, while leaving the more straightforward ones. This may leave the dataset skewed towards tests with assertions more syntactically similar to the human reference ones. For transparency we give statistics on the amount of datapoints before and after filtering in Table 1.

6.3 External validity

Only a single LLM was used for generating assertions. As different LLMs might behave differently, this means that our results might not be indicative of all LLMs in general. Additionally, we created our dataset using exclusively open source Java projects using JUnit 4/5, without any alternative assertion syntax like AssertJ or Hamcrest. This means that the diversity of the assertions is limited, possibly affecting the generalisability. In an attempt to improve generalisation we used realistic projects to create our dataset, and ensured diversity by taking at most 100 entries per project.

6.4 Conclusion validity

Our conclusions are based on a relatively small dataset, possibly impacting the statistical significance of the results we base our conclusions on. Our results for RQ1 and RQ3 are based on 177 datapoints, while our results for RQ2 are based on just 41 datapoints. While we reach statistical significance for RQ1 ($p < 0.001$), RQ2 and RQ3 are still based on a low amount of datapoints.

Additionally, we do not generate the assertions multiple times and average over the results, instead performing only one run. This means that our conclusion is based on just one realisation of the LLM's output, which might not be representative of the average behaviour. We mitigate this threat by including the set of generated assertions in our replication package and by generating the assertions using a fixed seed and a temperature of 0.

7 Conclusions and Future Work

In this paper we investigated the relation between syntactic and semantic equivalence between human and LLM-generated Java test assertions. Specifically, we worked to address the question: "To what extent do LLM-generated assertions differ syntactically but remain semantically equivalent to reference assertions?". We used the similarity of the set of mutants killed by the assertions as a proxy measure for the semantic similarity.

We found a statistically significant moderately strong correlation between normalised tree edit distance and Jaccard similarity of killed mutants ($\rho = -0.685$, $p < 0.001$). This confirms that the syntactic and semantic equivalence are indeed related, but the noise in the scatterplot in Figure 1 shows that you cannot rely on one as a reliable substitute for the other. Highly syntactically similar assertions can still have a very low semantic similarity, and vice versa.

Performing open coding on the 41 semantically equivalent but syntactically different datapoints revealed 10 different categories. Of these some showed universal tendencies. The LLM had a clear tendency to omit assertion messages and preferred equality assertions over boolean checks (e.g. `assertEquals(a, b)` over `assertTrue(a.equals(b))`). Other categories, like use of equivalent methods (e.g. `.get(0)` vs `.first()`) did not reveal any clear LLM preference.

We used a decision tree with a maximum depth of 1 to generate a threshold for the syntactic similarity of 0.41 (based on the normalised tree edit distance), that effectively splits the data. Below this threshold the median semantic similarity is 0.5290 with a high variance, while above the threshold the median semantic similarity is 1.000 with a much lower variance.

Altogether, our findings suggest that exact-match evaluation of LLM test assertions underestimates performance by penalising semantically equivalent, but differently expressed, assertions. Our findings also show that syntactic similarity, while not a perfect predictor, can be used for getting an idea of whether the assertions are likely to be semantically equivalent. For this the threshold we found for RQ3 can be used. A possible approach for evaluating large datasets could be to filter out entries highly unlikely to be semantically equivalent through removing entries below the threshold, and then performing mutation testing on the entries above the threshold.

Several options for future work still remain. Our dataset contains 177 entries after filtering, so a larger dataset would improve the reliability of all three findings, and in particular RQ2 and RQ3. Averaging over multiple runs per test case would reduce the influence of LLM non-determinism. Testing additional LLMs might reveal whether these findings are consistent for more LLMs, or limited to the model we used. As the threshold we found is based on a relatively small dataset it could be interesting to validate it on a larger dataset. Finally, we limited our assertions to pure Java JUnit assertions. Further research could also consider alternative assertion syntax, like Hamcrest.

8 Acknowledgements

We want to thank our supervisors and assistant professors Annibale Panichella and Mitchell Olsthoorn for providing support, guidance and feedback throughout the project.

References

[1] Azat Abdullin, Pouria Derakhshanfar, and Annibale Panichella. Test wars: A comparative study of sbst, symbolic execution, and llm-based approaches to unit test generation. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 221–232. IEEE, 2025.

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2014.

[3] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 449–452, 2016.

[4] Sayash Kapoor and Arvind Narayanan. Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, 4(9), 2023.

[5] Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D Ernst, and Mauro Pezze. Do llms generate useful test oracles? an empirical study with an unbiased dataset. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 278–290. IEEE, 2025.

[6] Severin Primbs, Benedikt Fein, and Gordon Fraser. Assert5: Test assertion generation using a fine-tuned code language model. In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 12–23. IEEE, 2025.

[7] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[8] André Silva, Nuno Saavedra, and Martin Monperrus. Gitbug-java: A reproducible benchmark of recent java bugs. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 118–122, 2024.

[9] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1398–1409, 2020.

[10] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.