Can Small Beat Big?

Evaluating Fine-Tuned CodeT5 Models on Assertion Generation Quality and Efficiency

Thimo van Leeuwen[1]

Supervisor(s): Annibale Panichella[1]**, Mitchell Olsthoorn**[1]

[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: Thimo van Leeuwen
Final project course: CSE3000 Research Project
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Alexios Voulimeneas

Abstract

Testing is a core practice in software development for detecting faults and checking that code behaves as expected. With the recent advent of Large Language Models (LLMs), code generation has never been more widespread. In assertion generation, where the focus is on the oracles that assess the state of the program, fine-tuned code language models have emerged. One such model, AsserT5, is a CodeT5-large (770M parameters) fine-tuned on focal-method and test-method pairs. Although it achieves state-of-the-art performance when measured by exact match to the ground truth, it remains unclear how the top-1 predictions of the smaller variants (CodeT5-small, 60M; CodeT5-base, 220M) perform on mutation score when the same fine-tuning procedure is applied.

Across ten real-world Java projects and 541 assertion-generation tasks, we find that the fine-tuned 60M CodeT5-small matches the 220M and 770M variants on mutation score (within 0.2 p.p.), achieving the highest score of the three by generating more assertions that compile. Among the larger code-specific baselines (Qwen2.5-Coder 3B, 7B, and 14B), CodeT5-small underperforms only the 14B model, and only by 0.6 p.p. This advantage is concentrated in just two of the ten projects, and the 14B model attains it at the cost of 38x more memory (9.00 GB vs 0.24 GB) and 2.6x slower inference. Because the difference is small and confined to two out of ten projects, we recommend the fine-tuned CodeT5-small to practitioners seeking local assertion-generation assistance at reasonable computational cost.

1 Introduction

Software testing is essential for detecting faults and preventing regressions during software development. A *test class* contains one or more *test methods* or *test cases*, which consist of a *test prefix*, which brings the program under test into a particular state, and one or more *test oracles* or *assertions*, which check whether the observed behavior is equal to the intended behavior [1]. Usually, a test method verifies the behavior of a specific *focal method*, i.e., the production method under test, which is inside the *focal class* [15].

While creating relevant test prefixes is not necessarily easy, writing useful assertions is especially demanding: the developer must reason about the intended behavior of the focal method in the given program state. Automating assertion generation can therefore reduce testing effort by allowing developers to focus on constructing relevant program states, while an automated assistant proposes the corresponding oracle.

Recent work in assertion generation has adopted pretrained code language models [4, 10]. In particular, AsserT5 is a CodeT5-large fine-tuned for assertion generation on focal method and test method pairs [10]. It reports substantially higher exact-match accuracy to the ground truth than earlier fine-tuned and general-purpose LLMs (like ChatGPT). However, exact-match accuracy only measures whether the generated assertion is textually identical to the developer-written assertion. A generated assertion may differ syntactically from the original while still verifying the same or more behavior of the focal method or it may be textually similar while failing to detect faults entirely.

For this reason, assertion generation models should also be evaluated using execution-based metrics. In particular, mutation testing estimates the fault-detection strength of a test suite by measuring whether tests detect artificially injected faults, called mutants [6]. Prior work has shown that text-based similarity metrics do not necessarily reflect mutation-testing effectiveness [11], and Molinelli et al. [7] confirm this empirically for assertion generation: LLM-generated assertions achieve mutation scores competitive with developer-written ones despite low textual similarity. However, their mutation analysis covers only a single selected model configuration, and their study targets general-purpose LLMs rather than models specifically fine-tuned for assertion generation.

Conversely, AsserT5 demonstrates the effectiveness of fine-tuning CodeT5-large for assertion generation [10], but evaluates the generated assertions primarily through exact-match accuracy and does not investigate their mutant-killing effectiveness. Moreover, the fine-tuning procedure is applied only to CodeT5-large, leaving out the small and base variants, which we hypothesize could achieve similar performance at lower computational cost.

This leaves two open questions. First, it is unclear how model size affects the quality of fine-tuned CodeT5 models for assertion generation when quality is measured by their ability to kill mutants. Second, it is unclear how such task-specific fine-tuned models compare against larger code-specific LLMs on the same benchmark under identical mutation analysis. This comparison matters because it pits specialization against scale: whether task-specific fine-tuning can match or exceed a substantially larger model that is expected to bring broader knowledge of code, at increased computational cost.

This paper studies these questions by evaluating fine-tuned CodeT5 models of different sizes on JUnit 4 and 5 assertion generation. Given a focal method and a test prefix possibly containing other assertions, each model generates a single (top-1) candidate assertion, which is inserted back into the test case and evaluated through compilation rate, execution validity rate, and mutation score. We compare fine-tuned CodeT5-small (60M parameters), CodeT5-base (220M), and CodeT5-large (770M) to each other and against Qwen2.5-Coder models of different sizes (3B, 7B, and 14B). The evaluation is performed on ten real-world open-source Java projects.

The results show that the fine-tuned CodeT5-small achieves a mutation score similar to that of the larger fine-tuned variants. It does so by generating more conservative assertions that stay closer to the test prefix, compile more often, and thus have more opportunities to kill mutants. Among the larger code-specific models, only Qwen-14B outperforms the CodeT5 models on mutation score, and only by 0.6 percentage points. This advantage is concentrated in two projects, where Qwen-14B reproduces a memorized numeric constant and a focal-method expression that the CodeT5 models do not.

The paper is structured as follows. Section 2 covers the background on the models we evaluate and on mutation testing. Section 3 covers the study design, where we lay out the research questions and metrics. Section 4 covers our adapted methodology from Molinelli et al. [7]. Section 5 reports the results. Section 6 discusses the

results. Finally, Section 9 provides conclusions and recommendations for future work.

2 Background and Related Work

In this section we first describe the models we evaluate and the models they are based on (2.1). We then explain what mutation testing is (2.2). Finally, we review how prior studies use mutation testing to evaluate generated assertions and state the methodology we follow (2.3).

2.1 Code Language Models

Table 2 lists the six models we evaluate. Three are fine-tuned CodeT5 models. To describe them we first introduce CodeT5 (2.1.1) and AsserT5 (2.1.2), whose fine-tuning procedure they reuse, before turning to the models themselves (2.1.3). The other three are Qwen2.5-Coder models (2.1.4).

2.1.1 CodeT5. This is an encoder-decoder family pretrained on code and meant to be fine-tuned for code understanding and generation tasks [14], such as assertion generation. It comes in three sizes: small, base, and large. Their weights take respectively 0.24 GB, 0.87 GB, 2.89 GB of memory and parameter count are respectively 60M, 220M, 770M [14].

2.1.2 AsserT5. It is a CodeT5-large fine-tuned on a modified *methods2test* dataset [10, 13]. Figure 1 illustrates the task the model is fine-tuned on using the dataset. Given a focal method before the `<SEP>` token, and a test method containing a masked assertion with `<ASSERTION>`: generate an assertion replacing the mask.

Although the model is tasked with generating a single assertion for a single test method, the model performs better when other assertions are provided in the test prefix [10].

2.1.3 Fine-Tuned CodeT5. The CodeT5 models were fine-tuned by Annibale Panichella [1]. He further cleaned up the modified *methods2test* dataset used by Primbs et al. [10]. Afterwards he applied the same fine-tuning procedure as them. Table 1 shows the distribution of assertions in the training data.

2.1.4 Qwen2.5-Coder-Instruct-Q4_K_M. This is a decoder-only LLM family trained on 5.2T tokens of code, text, and math in a 7:2:1 ratio [5]. Rather than fine-tuning, this study applies it to assertion generation through instructions, as Molinelli et al. do [7].

2.2 Mutation testing

Mutation testing gauges a test suite's fault-detection ability by deliberately seeding the program under test with small artificial faults [6]. Each fault comes from a *mutation operator*: a rule that applies one localized edit to the source, for instance swapping an arithmetic operator or inverting a conditional. The result is a *mutant*, a version of the program identical to the original except at that single point [9]. Every mutant is then exercised against the test suite. If some test method that previously passed on the unmodified program fails on the mutant, the mutant is *killed*; otherwise it *survives*. The *mutation score* is the proportion of mutants killed, with a higher value signaling a test suite is more sensitive to changes in program behavior.

[1] Personal communication, 2026.

```
char last(String s) { return s[s.length-1]; } <SEP> @Test
void testLast() { char res = last("abc"); <ASSERTION> }
```

↓ AsserT5

```
assertEquals(res, 'c');
```

Figure 1: The assertion generation task AsserT5 is fine-tuned on. This example is from Primbs et al. [10].

Because assertions can cause a test to fail when behavior changes, mutation score is suitable for measuring assertion quality rather than merely measuring whether an assertion is similar to the ground truth. In this study, we use PIT to generate mutants as it is a commonly used Java mutation testing tool [2].

2.3 Mutation Testing for Assertion Generation

To find out how to perform mutation testing effectively, we consulted the literature and found two studies.

Hossain and Dwyer generate test prefixes with EvoSuite [4], a search-based tool that derives tests from the runtime behavior of the program [3]. Then they append a single generated assertion to each prefix, and use PIT to generate mutants.

Molinelli et al. [7] instead start from real-world test methods in open-source Java projects. Their methodology revolves around isolating each assertion of a test method into its own distinct generation task. This means that a test method containing N assertions is split into N test methods. Each of these split test methods differ in the number of assertions they contain. The first split contains the code of the original test method up to and including the first assertion, the second split contains the code up to and including the second assertion. Thus the Nth split is equal to the original method containing N assertions. Each split test method is then turned into a generation task by masking the last assertion (the *target* assertion). Afterwards, the models are prompted to generate an assertion replacing the mask. Finally, to obtain the mutation score of the model, they use PIT using the test suite obtained by replacing the *target* assertion of each split test method with a single generated assertion.

In this paper, we adopt the methodology of Molinelli et al. rather than that of Hossain and Dwyer, because their task is built on human-written test methods. This matches how an assertion-generation model would be used in practice: a developer writes the test prefix themselves, possibly including earlier assertions, and uses the model as an assistant to generate an assertion to append. It also aligns with how the fine-tuned CodeT5 models in this study are trained, since they have learned to generate assertions for human-written test methods.

3 Study Design

We answer the following research questions:

RQ1: *How does model size among fine-tuned CodeT5 models affect the assertion generation quality and efficiency?*

Table 1: Distribution of JUnit assertion types in the dataset of the fine-tuned CodeT5 models

Assertion Method	Share (%)
`assertEquals`	60.86
`assertTrue`	18.01
`assertFalse`	8.73
`assertNotNull`	6.62
`assertNull`	4.17
`assertThrows`	0.97
`assertNotEquals`	0.65

Table 2: Models evaluated in this study. The size refers to the size of the weights. The sizes of the Qwen models are from Ollama [8].

Model	Parameters	Size (GB)
Fine-Tuned CodeT5-small	60M	0.24
Fine-Tuned CodeT5-base	220M	0.87
Fine-Tuned CodeT5-large	770M	2.89
Qwen2.5-Coder-3B	3B	1.90
Qwen2.5-Coder-7B	7B	4.70
Qwen2.5-Coder-14B	14B	9.00

RQ2: *How do fine-tuned CodeT5 models compare to larger code-specific LLMs on assertion generation quality and efficiency?*

We assess assertion quality through a staged evaluation funnel, where each stage is a precondition for the next: a generated assertion must compile, then pass the test, before it can be assessed on its ability to kill mutants. We report three metrics accordingly:

Compilation rate. The fraction of generated assertions that compile when appended to their test method.

Execution validity rate. The fraction of compiled assertions whose test method executes successfully on the original program version. Following Molinelli et al. [7], this study assumes the code under test is correct. Therefore an assertion that fails on the original program is considered incorrect: it rejects valid behavior rather than capturing it.

Mutation score. Our primary metric for assertion quality, as defined in Section 2.2 (killed mutants / total mutants).

We assess the efficiency of generation according to two metrics:

Inference time. The time it takes for an assertion to be generated.

Size of weights. The amount of memory occupied by the model weights during inference. For the CodeT5 models this is the native FP32 checkpoint; for the Qwen2.5-Coder models it is the 4-bit quantized weights as served by Ollama. The weights are static and visible in Table 2.

Because the CodeT5 models share one architecture and training setup and differ only in size, RQ1 provides a controlled measurement of the effect of scale within the family.

RQ2 compares the fine-tuned CodeT5 models against Qwen2.5-Coder 3B, 7B, and 14B. This comparison is motivated by practical deployment: if small fine-tuned models achieve comparable assertion quality to larger code-specific LLMs, they potentially enable lower-cost assertion generation on a broader range of developer hardware.

We chose the Qwen2.5-Coder family because all three sizes fit within our 16GB VRAM budget, which our inference time measurements require. Additionally, Molinelli et al. [7] used this model family, allowing us to reuse their refined prompt structure.

4 Methodology

Our methodology is centered around a dataset of 541 datapoints (assertion generation tasks) mined from 10 real-world open-source Java projects (4.1). Then we construct prompts for both model families (4.2). After we run inference for each model (4.3). Finally, we evaluate the generated assertions with mutation testing (4.4).

4.1 Dataset

We build the dataset by selecting suitable projects from GitBug Java (4.1.1) and then mining test methods from them, which are transformed into data points (4.1.2). We follow the data point definition of Molinelli et al. [7] and extend it so that each data point additionally possesses a unique, heuristically identified focal method, as shown in Algorithm 1.

4.1.1 Project Selection. We use GitBug Java as the source of projects for our study. The corpus contains 55 Java projects collected in 2023 and covers a variety of real-world software systems, including parsers, APIs, and cloud orchestration tools [12]. This diversity makes it suitable for evaluating assertion generation across different types of Java projects.

To be compatible with the dataset-construction algorithm of Molinelli et al. [7], a project was included only if it satisfied the following requirements:

(1) it uses Maven as its build system;
(2) it contains a single top-level `pom.xml` file;
(3) it declares JUnit as a test dependency.

Of the 55 GitBug Java projects, 43 use Maven. Of these, 36 also satisfied the second and third requirements. From these 36 projects we extracted the data points whose target assertion is a JUnit assertion; 27 projects yielded at least one such data point. We then ran inference for all of these projects and used PIT to perform mutation analysis, which produced complete mutation-testing results for 16 projects. Finally, we discarded projects with fewer than 10 data points, leaving 10 projects with 541 data points in total. Table 3 shows the characteristics of the resulting dataset.

4.1.2 Data point construction. We follow the dataset construction procedure of Molinelli et al. [7] and extend it with the focal-method requirement of the fine-tuned CodeT5 models; the lines we changed in their algorithm are shown in Algorithm 1, and we describe the full procedure here.

The pipeline first pairs each test class with its focal class. It assumes that production and test code follow corresponding package structures under `src/main/java` and `src/test/java`, as is conventional in Maven projects, and infers the focal class from the test class name by removing the `Test` suffix.

For each test class with an identified focal class, the test methods are normalized and then split so that every assertion is evaluated separately: a method with N assertions yields N data points, where the i-th data point contains the test code up to and including the i-th assertion. In each data point the final assertion is masked to become the target assertion, and the statements preceding it form the test prefix.

Within each data point we additionally require a unique focal method, identified using the two heuristics of Primbs et al. [10]: a name match between the (affix-stripped) test method and a method of the focal class; if that fails, a unique focal-class method invoked in the test prefix and the target assertion. We include the target assertion in this search because, like Watson et al. [15], we assume a practical setting, such as an IDE, in which the focal method is always available; for the same reason, we discard any data point for which no focal method can be identified.

4.2 Prompt Construction

For each data point, we construct two prompts:

Fine-Tuned CodeT5. These models have a context window of 386 tokens [10]. For 106 data points, the input (focal method + `<SEP>` + masked test method) exceeded this limit, and we applied the following truncation strategies:

(1) We remove lines from the end of the focal method until the input fits or only the focal method's signature remains. This follows how focal methods were truncated in the training dataset of the fine-tuned CodeT5 models [10]. For 26 data points (≈5% of the dataset) this was sufficient.
(2) The remaining 80 data points still exceeded the context window. For these, we additionally remove lines from the end of the test method while retaining the assertion mask and the closing bracket, as Primbs et al [10] do.

Qwen2.5-Coder. These models have a context window of 32K tokens [5]. When providing the same level of context as the fine-tuned CodeT5 models (focal method + test prefix), we found no cases where the context window was exceeded. This creates an information asymmetry, which we accept because in a practical setting the input would not be truncated unnecessarily

4.3 Inference Setup

Using the generated prompts, inference is performed separately for each family. For the fine-tuned CodeT5 models we use PyTorch 2.9.2 with beam search (`num_beams = 10`), `max_new_tokens = 64`, and `max_src_length = 386`. We set `top_k = 10`. But we only take the first prediction as the task requires a single assertion rather than a diverse set of candidates.

The Qwen2.5-Coder models are queried through Ollama [8] 0.24.0 with default settings. Because the Qwen2.5-Coder models are nondeterministic under these settings, we repeat inference and the subsequent mutation analysis six times and report the average results across runs. All inference is performed sequentially on an AMD Radeon RX 9070 XT 16GB GPU on Windows 11 25H2.

Table 3: The projects and statistics of the dataset.

Name	Commit	DPs	Number of Test	
			Classes	Methods
chesslib	cf68677	124	5	16
cloudsimplus	61c8b942d	105	16	57
cbor-java	cabd70d	94	16	49
nbvcxz	ee8d5c6	91	6	10
urnlib	106be8d	32	11	32
crawler-commons	2c2cb3b	31	5	8
UT4X-Converter	e719841	21	4	6
epubcheck	0759a82a	18	3	11
solarpositioning	79c0044	13	2	8
word-wrap	e59eedf	12	2	10
Total		**541**	**70**	**207**

For the inference time measurements, we exclude the first prompt of each project from all models, as it includes Ollama's model-loading time for the Qwen models. We measure inference time for each prompt using the round-trip latency of the local API call.

4.4 Mutation Analysis

After inference we collect our metrics using the mutation analysis algorithm of Molinelli et al. [7]. The algorithm proceeds in three steps. First, it executes `PIT` for each project using test classes containing the split test methods as the target tests and all corresponding focal classes as the target classes. This way we obtain the mutation score for the original test methods (*human* baseline).

Second, it removes the target assertion from each split test method for which an assertion was generated and runs PIT again to obtain a *no-oracle* mutation score. Lastly, for each generated assertion, it appends the assertion to its corresponding split test method and checks whether the method compiles and executes successfully. If either fails, the assertion is discarded; the split test method reverts to its version in the *no-oracle* baseline.

We run PIT with the `ALL` mutation operator group to obtain the most fine-grained results, using pitest-maven-1.23.1 and pitest-junit5-plugin 1.2.2 for JUnit 5 projects.

5 Results

This section presents the results per research question. Tables 4, 5, and 6 report the three metrics per project; the best value(s) per project is shown in bold. The values of the Qwen models are means over six runs, with ± denoting the standard deviation across runs. Table 6 additionally reports the human and no-oracle baselines from Section 4.4. These baselines respectively indicate how the models perform in relation to developer-written assertions and how much of a model's performance is due to the test prefixes and other test methods killing mutants. Projects are sorted by the gap Δ between these baselines.

5.1 RQ1: Model Size of Fine-Tuned CodeT5

Table 4 shows the compilation rate. CodeT5-small achieves the highest average (72% vs. 65% for base and 62% for large) and strictly

Algorithm 1 Building dataset D of oracles (adapted from Algorithm 2 of Molinelli et al. [7]; our additions in green).

Require: set S of ⟨focal class, test class⟩ pairs, built from repository r as in [7]
Ensure: dataset D of 6-tuples $\langle tc, prefix, fc, fm, invoked, a \rangle$

$D \leftarrow \emptyset$
for $\langle fc, tc \rangle \in S$ **do**
  for $t \in tc.tests$ **do**
    $t.body \leftarrow$ EnrichTestCase($t.body$) ▷ as in [7]
    $A \leftarrow$ ExtractAssertions($t.body$)
    for $a \in A$ **do**
      $prefix \leftarrow$ statements of $t.body$ before a
      $fm \leftarrow$ SelectFocalMethod($prefix \cup \{a\}$, fc)
      if $fm = \perp$ **then continue** ▷ drop datapoint
      end if
      $invoked \leftarrow \{ mc.decl \cup mc.decl.body : \text{method call } mc \in prefix \}$
      $D \leftarrow D \cup \{\langle tc, prefix, fc, fm, invoked, a \rangle\}$
    end for
  end for
end for
return D

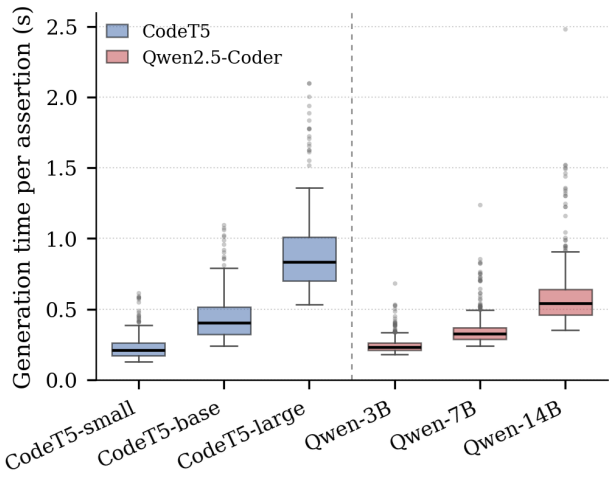

Figure 2: Inference time of each assertion for each model. Each inference time of the Qwen models is the median of the 6 runs.

outperforms both larger variants in 6 of 10 projects; conversely, both larger variants outperform it only in `word-wrap` and `urnlib`. The largest gap occurs in `UT4X-Converter`, where CodeT5-small's assertions compile almost twice as often as those of the larger variants (81% vs. 43%). Across the sizes, average compilation rate decreases monotonically with model size.

Table 5 shows the execution validity rate. The three variants cluster within three percentage points: 51% for small, 54% for base, and 53% for large. Unlike compilation rate, execution validity rate is therefore not monotone in model size.

Table 6 shows the mutation score. Despite the differences in the preceding metrics, the average mutation scores are nearly identical (46.2%, 46.1%, and 46.0% for small, base, and large), and all three models achieve exactly equal scores in 7 of 10 projects. The largest difference occurs in `nbvcxz`, where CodeT5-small outperforms base and large by 0.8 percentage points.

In summary, each larger CodeT5 variant decreases performance by 0.1 percentage points in the most important quality metric: mutation score. Furthermore, average compilation rate decreases monotonically from 72% for CodeT5-small to 62% for CodeT5-large and average execution validity varies non-monotonically by 3 percentage points. CodeT5-small thus performs the best of the fine-tuned models and at higher efficiency than CodeT5-large: 4x lower median per-prompt inference time (0.21 s vs. 0.83 s for CodeT5-large; Figure 2) and the weights consuming 12x less memory (0.24 GB vs. 2.89 GB for CodeT5-large). In addition, CodeT5-small performs 2.2 percentage points higher than the no-oracle baseline (46.2% vs. 44.0%; Table 6), while performing worse than the human baseline by 3.6 percentage points, thus recovering 38% of the average Δ between the projects.

5.2 RQ2: Comparison to Larger Code-Specific Models

On compilation rate (Table 4), CodeT5-small ranks second, behind only Qwen-14B (72% vs. 77%) and outperforms both smaller Qwen models (61% for 3B, 68% for 7B). CodeT5-base and large fall in between the two smaller Qwen models (65% and 62%). The two families trend in opposite directions with size: compilation rate decreases with size for CodeT5 but increases for Qwen2.5-Coder.

On execution validity rate (Table 5), all three CodeT5 variants perform at the lower end of the Qwen range (51–54%), level with Qwen-3B (51%) and below Qwen-7B (64%) and Qwen-14B (74%). As with compilation rate, execution validity rises monotonically with size for Qwen2.5-Coder but varies non-monotonically for CodeT5.

On mutation score (Table 6), all three CodeT5 variants outperform both smaller Qwen models by at least 0.5 percentage points (46.0–46.2% vs. 45.4% for 3B and 45.5% for 7B) and underperform only Qwen-14B, whose 46.8% average exceeds the best CodeT5 variant (CodeT5-small, 46.2%) by 0.6 percentage points. The Qwen-14B advantage is concentrated in two projects, `cbor-java` (66.3% vs. 62.6–62.7%) and `solarpositioning` (30.3% vs. 27.1%); on the remaining eight projects all three CodeT5 variants stay within 0.7 percentage points of Qwen-14B, and CodeT5-small within 0.4.

In summary, all CodeT5 models perform at least 0.5 percentage points better on mutation score than Qwen-3B and 7B, despite the CodeT5 models having at least 10 percentage points decrease in execution validity rate compared to Qwen-7B (54% for CodeT5-base vs. 64% for Qwen-7B). However, all models are behind in mutation score when compared to Qwen-14B. The best performing CodeT5 model (CodeT5-small) performs 0.6 percentage points worse than Qwen-14B, however it is better on efficiency: 38x lower memory consumption (0.24GB vs. 9.00 GB) and 2.6x faster median inference time (0.21 s vs. 0.54 s).

Table 4: Per-project compilation rate in %, same project order as Table 6.

		Fine-Tuned CodeT5			Qwen2.5-Coder		
ID	Project	small	base	large	3B	7B	14B
1	cbor-java	88	79	62	$72_{\pm 11}$	$76_{\pm 4}$	$\mathbf{89}_{\pm 4}$
2	solarpositioning	**92**	77	**92**	$68_{\pm 16}$	$63_{\pm 3}$	$83_{\pm 8}$
3	nbvcxz	26	23	25	$86_{\pm 7}$	$90_{\pm 4}$	$\mathbf{94}_{\pm 2}$
4	cloudsimplus	**77**	73	70	$61_{\pm 10}$	$73_{\pm 4}$	$73_{\pm 6}$
5	word-wrap	58	**67**	**67**	$65_{\pm 3}$	$65_{\pm 3}$	$65_{\pm 3}$
6	epubcheck	**89**	**89**	67	$30_{\pm 5}$	$44_{\pm 7}$	$52_{\pm 10}$
7	urnlib	56	72	59	$46_{\pm 14}$	$64_{\pm 3}$	$\mathbf{76}_{\pm 4}$
8	UT4X-Converter	**81**	43	43	$52_{\pm 11}$	$56_{\pm 13}$	$79_{\pm 7}$
9	crawler-commons	**74**	71	71	$67_{\pm 12}$	$66_{\pm 6}$	$73_{\pm 5}$
10	chesslib	79	60	60	$67_{\pm 13}$	$\mathbf{89}_{\pm 4}$	$90_{\pm 3}$
Average		72	65	62	$61_{\pm 9}$	$68_{\pm 5}$	$\mathbf{77}_{\pm 5}$

Table 5: Per-project execution validity rate in %, same project order as Table 6.

		Fine-Tuned CodeT5			Qwen2.5-Coder		
ID	Project	small	base	large	3B	7B	14B
1	cbor-java	64	81	83	$68_{\pm 6}$	$76_{\pm 2}$	$\mathbf{87}_{\pm 1}$
2	solarpositioning	**25**	10	8	$8_{\pm 7}$	$8_{\pm 9}$	$24_{\pm 9}$
3	nbvcxz	25	14	9	$16_{\pm 8}$	$31_{\pm 7}$	$\mathbf{43}_{\pm 5}$
4	cloudsimplus	70	77	78	$70_{\pm 6}$	$81_{\pm 5}$	$\mathbf{87}_{\pm 3}$
5	word-wrap	43	38	50	$81_{\pm 7}$	$96_{\pm 7}$	$\mathbf{98}_{\pm 5}$
6	epubcheck	25	31	25	$25_{\pm 21}$	$64_{\pm 11}$	$\mathbf{73}_{\pm 11}$
7	urnlib	94	87	**100**	$77_{\pm 10}$	$89_{\pm 5}$	$88_{\pm 3}$
8	UT4X-Converter	41	**78**	**78**	$56_{\pm 11}$	$55_{\pm 9}$	$77_{\pm 5}$
9	crawler-commons	57	50	41	$48_{\pm 13}$	$56_{\pm 12}$	$\mathbf{78}_{\pm 10}$
10	chesslib	62	72	57	$66_{\pm 8}$	$78_{\pm 5}$	$\mathbf{84}_{\pm 2}$
Average		51	54	53	$51_{\pm 10}$	$64_{\pm 7}$	$\mathbf{74}_{\pm 5}$

Table 6: Per-project mutation score in %, sorted by the human − no-oracle gap (Δ). The *no-oracle* baseline is the score with the target assertion removed, capturing mutants killed by the test prefix and other test methods alone; the *human* baseline is the score of the original developer-written assertions (Section 4.4). Thus, a large Δ indicates a project where the target assertions kill many mutants that the test prefix and other test methods do not, leaving more room for a generated assertion to matter. Qwen values are means over six runs.

					Fine-Tuned CodeT5			Qwen2.5-Coder		
ID	Project	no_oracle	human	Δ	small	base	large	3B	7B	14B
1	cbor-java	54.2	69.7	15.6	62.6	62.7	62.7	$60.7_{\pm 1.5}$	$62.2_{\pm 1.7}$	$\mathbf{66.3}_{\pm 1.3}$
2	solarpositioning	21.0	32.4	11.4	27.1	27.1	27.1	$24.0_{\pm 4.7}$	$22.0_{\pm 2.5}$	$\mathbf{30.3}_{\pm 0.5}$
3	nbvcxz	37.5	47.6	10.1	**39.4**	38.6	38.6	$39.0_{\pm 0.4}$	$38.9_{\pm 0.2}$	$39.3_{\pm 0.4}$
4	cloudsimplus	26.6	35.6	9.1	28.9	29.0	28.4	$28.9_{\pm 0.9}$	$\mathbf{29.5}_{\pm 0.6}$	$29.1_{\pm 0.5}$
5	word-wrap	74.5	79.8	5.3	74.5	74.5	74.5	$74.6_{\pm 0.1}$	$\mathbf{74.9}_{\pm 0.0}$	$\mathbf{74.9}_{\pm 0.0}$
6	epubcheck	27.7	30.1	2.4	**28.6**	**28.6**	**28.6**	$27.7_{\pm 0.0}$	$28.0_{\pm 0.5}$	$28.3_{\pm 0.5}$
7	urnlib	63.1	64.9	1.9	**64.7**	**64.7**	**64.7**	$63.7_{\pm 0.4}$	$64.0_{\pm 0.4}$	$64.5_{\pm 0.4}$
8	UT4X-Converter	22.3	23.7	1.3	22.4	22.4	22.4	$22.3_{\pm 0.0}$	$22.3_{\pm 0.0}$	$\mathbf{22.6}_{\pm 0.1}$
9	crawler-commons	51.1	51.6	0.5	**51.5**	**51.5**	**51.5**	$51.1_{\pm 0.0}$	$51.1_{\pm 0.0}$	$51.1_{\pm 0.0}$
10	chesslib	61.9	62.1	0.2	61.9	61.9	61.9	$61.9_{\pm 0.1}$	$\mathbf{62.0}_{\pm 0.1}$	$\mathbf{62.0}_{\pm 0.0}$
Average		44.0	49.8	5.8	46.2	46.1	46.0	$45.4_{\pm 0.8}$	$45.5_{\pm 0.6}$	$\mathbf{46.8}_{\pm 0.4}$

6 Discussion

In this section we provide a qualitative analysis of why, within the CodeT5 family, both compilation rate and—to a lesser extent—mutation score increase as model size decreases (6.1). We then investigate why Qwen-14B outperformed all CodeT5 models on mutation score (6.2), examine the effect of prompt truncation on the results (6.3), and finally present the implications for practitioners (6.4).

6.1 Within the Fine-Tuned CodeT5 Family

We first examine UT4X-Converter (6.1.1), which has the largest compilation-rate gap between CodeT5-small and the two larger CodeT5 models (81% vs. 43%). We then look at the compilation rate per assertion type per model (6.1.2). Finally, we explain why the mutation score in `nbvcxz` is higher for CodeT5-small (6.1.3).

6.1.1 Explaining CodeT5-small's Higher Compilation Rate. We found two reasons for the higher compilation rate. The first is that CodeT5-small is more conservative with its predictions than the larger fine-tuned models. Figure 4 illustrates this. The test method is the 8th split, so the test prefix in the prompt contains 7 assertions. The focal method, `getMoverProperties()`, is a getter. All models try to use the information provided by the preceding assertion, which called `getPositions().size()` on the object returned by the focal method. The small model stays close to this context and simply calls `get(0).toString()`. The base model attempts a stronger assertion by guessing a type that does not exist, and the large model guesses an even stronger one, asserting not only the type but also its fields.

The second reason is that the truncation strategy from Section 4.2 widens the gap. The prompt is limited to 386 tokens, and a long split test method reaches this limit. If there are any consecutive splits of the same test method left, the model generates the same assertion for all of them. This occurred for the last 5 splits of `testConvertU1MoverToUT4`: the repeated assertion compiled for the small model and was counted as 5 compiling assertions, while the larger models' repeated assertion failed and was counted as 5 compilation failures.

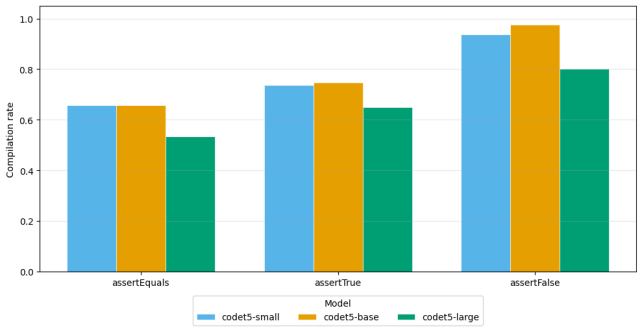

Figure 3: Compilation rates for the three most frequent generated assertion types across fine-tuned CodeT5 models. CodeT5-large consistently exhibits lower rates than its smaller counterparts.

6.1.2 Compilation Rate by Assertion type. Figure 3 reports the compilation rate of the three most frequently generated assertion types. Across all three, the large model has the lowest rate. Its lower overall compilation rate is therefore not driven by which assertion types it generates, but by the arguments those assertions contain: as illustrated in Figure 4, the large model passes arguments that reference symbols that do not exist.

6.1.3 Why Conservatism Improves Mutation Score. The same conservative behavior explains why the mutation score in `nbvcxz` is higher for the small model than for the larger ones. All three models correctly compared the `expected` and `computed` variables from the test prefix using `assertEquals`, and all three correctly selected the overloaded method that takes a tolerance argument. However, they diverged on how that tolerance was expressed. The larger models produced assertions referencing a `DELTA` constant as the tolerance, usually defined in test classes, whereas the small model stayed with the test prefix and generated a literal value instead. Because no `DELTA` constant was defined in this test class, the larger models' assertions failed to compile, whereas the small model's assertions compiled and killed the mutants.

6.2 Sources of Qwen-14B's Higher Mutation Score

Qwen-14B's average advantage in mutation score is almost entirely due to two projects.

`solarpositioning`. A single assertion accounts for the gap. Figure 5 illustrates it. Qwen-14B correctly predicted the Julian Date of the J2000.0 epoch (2451545.0), killing 17 mutants that all CodeT5 models missed. Manual inspection confirms this is an outlier rather than a pattern: across the seven `JulianDate` tests, Qwen-14B produced a passing assertion only for `testY2K`. Thus, we think this specific epoch is a well-known constant the model likely saw many times in the training data.

`cbor-java`. Qwen-14B kills 16 mutants that none of the fine-tuned CodeT5 models kill. These are killed by four test methods, and 14 of the 16 are `hashCode` mutants. The focal method computes the hash as `super.hashCode() ^ Objects.hashCode(field)`, which Qwen-14B reconstructs in the generated assertion. In contrast, the CodeT5 models omit the `^ Objects.hashCode(field)` term or XOR against the wrong operand. Thus, Qwen-14B reproduced the focal method's hash expression whereas the CodeT5 models do not.

6.3 Prompt Truncation Effect

As noted in Section 4.2, the 386-token context window of the fine-tuned CodeT5 models required us to truncate either the focal method alone, or additionally the test method of the prompt, for 106 of the 541 data points. The Qwen models, with a 32K-token context window, always received the full input. To test whether this truncation explains the performance difference between the families, we split the data points into two sets: those that fit every model, and those that are truncated for CodeT5.

On the truncated set, assertions from all three CodeT5 models compile roughly half as often as on the full set. Over the same set, Qwen-14B's compilation rate instead rises by 10 percentage points. This suggest that truncation lowers compilation rate for the CodeT5 models specifically, rather than the truncated data points simply being harder for every model.

However, the effect on mutation score is negligible. Across the entire truncated set, Qwen-14B kills only 7 more mutants over the no-oracle baseline than any CodeT5 model. These 7 mutants are diluted by large denominators: 2 fall in `chesslib` (denominator 3239) and 5 in `nbvcxz` (denominator 980). The compilation rate reduction from truncation therefore does not translate into a meaningful decrease in average mutation score of the CodeT5 models.

6.4 Implications for Practitioners

For a developer who writes the test prefix and wants a local assistant to generate the assertion, we recommend the fine-tuned CodeT5-small over the larger code-specific models. It matches the larger CodeT5 variants on mutation score and underperforms the best model, Qwen-14B, by only 0.6 percentage points, at 2.6x faster median inference and 38x less memory consumption.

This 0.6-percentage-point gap is concentrated in two of the ten projects, where it reduces to a few assertions: a memorized constant in `solarpositioning` and the hash-code expression of the `hashCode` focal method in `cbor-java`. On the remaining 8 projects the two models differ by at most 0.4 percentage points. Additionally, across the 202 mutants that either model kills over the no-oracle baseline, CodeT5-small alone recovers 147 (73%). The smaller model therefore captures most of the fault-detection capability of the best model at a fraction of its memory and inference cost.

7 Threats to Validity

Internal Validity: Data Leakage. Our dataset contains only repositories collected in 2023. These repositories predate the training cutoff of the Qwen2.5-Coder models, which are trained on GitHub data from before February 2024 [5], so they may be fully contained in that model's training set. There is also potential leakage for the fine-tuned CodeT5 models: methods2test was aggregated in 2022, giving a possible slight overlap with our projects [13]. We are unable to quantify the extent of either overlap.

```
// Prompt (focal signature + truncated test prefix)
public MoverProperties getMoverProperties() <SEP> @Test
void testConvertU1MoverToUT4() {
    ...
    assertEquals(1, moverBrush.getMoverProperties().getPositions().
      ↪ size());
    <ASSERTION>;
}
// small compiles
assertEquals("", moverBrush.getMoverProperties().getPositions().
      ↪ get(0).toString());

// base fails (type Vector3 does not exist):
assertTrue(moverBrush.getMoverProperties().getPositions().get(0)
      ↪ instanceof Vector3);

// large fails (type Vector3D does not exist):
assertEquals(new Vector3D(1480, 780), moverBrush.
      ↪ getMoverProperties().getPositions().get(0));
```

Figure 4: An example from `UT4X-Converter`. Larger CodeT5 models hallucinate symbols and fail to compile.

```
// Focal method
public double getJulianDate() {
    return julianDate;
}

// Test method
@Test
void testY2K() {
    ZonedDateTime utcTime = ZonedDateTime.of(2000, 1, 1, 12, 0, 0,
      ↪  0, ZoneOffset.UTC);
    JulianDate julDate = new JulianDate(utcTime);
    <ASSERTION>;
}
// Qwen-14B passes: it guessed correctly and kills 17 mutants
assertEquals(2451545.0, julDate.getJulianDate(), 0.001);

// CodeT5-small fails: generates the wrong Julian date
assertEquals(2000, julDate.getJulianDate(), 0);

// CodeT5-base fails: hallucinates .toEpochMilli()
assertEquals(utcTime.toEpochMilli(), julDate.getJulianDate());

// CodeT5-large fails similarly to CodeT5-small
assertEquals(2000.001, julDate.getJulianDate(), 0.0001);
```

Figure 5: In this example from `solarpositioning`, only Qwen-14B predicts the correct constant.

External Validity: Selection Bias. Each filter stage in Section 4.1 removes projects and biases the resulting dataset, reducing generalizability. To partially mitigate this, we relied on the initial GitBug-Java collection, spanning diverse range of real-world Java projects [12].

Construct Validity: Efficiency Metrics. We use the size of the weights in memory as one measure of generation efficiency. This is only one component of memory usage and therefore represents a floor on the true inference-time memory footprint. To partially account for this, we also report inference time as a second efficiency metric.

Conclusion Validity: Nondeterminism. The Qwen2.5-Coder models are nondeterministic under default Ollama settings. To partially mitigate the effect on our results, we ran inference with 6 different seeds (42–47), performed mutation testing on each run, and report the mean and standard deviation across the 6 runs. For the qualitative mutation-score analysis in the discussion, we use the run closest to the project mean, and when examining which mutants the Qwen2.5-Coder models kill, we count only mutants killed in a majority (at least 4 of 6) of runs.

8 Responsible Research

Reproducibility. Our study is designed to be fully reproducible. We build on the publicly available pipeline of Molinelli et al. [7] and publish our modifications to it, together with our prompts, generated assertions, and mutation-analysis results, at https://github.com/tvl007/Can-Small-Beat-Big. We additionally report the exact project commits used (Table 3), so that the dataset can be reconstructed identically.

Use of open models. All model families we evaluate (CodeT5 and Qwen2.5-Coder) are open-weight and freely available, rather than closed-source proprietary models. This supports reproducibility, since anyone can obtain and run the exact models we used without depending on a commercial service.

Human oversight. Our evaluation setting keeps the developer in the loop by design: the human writes the test prefix and the model only proposes the final assertion, which the developer can inspect, accept, or reject. The tools we study are therefore positioned to augment rather than replace human testing. We note, however, that the same models could be incorporated in more autonomous agentic pipelines, where this oversight is reduced.

Malicious intent. The results of our study can be used by bad actors to test malicious code locally off-grid. We cannot preclude that our results will be used for malicious ends as the results of our study are publicly shared.

AI tool usage. Anthropic's Claude (Opus 4.8) was used during this research to assist with editing prose for grammar, clarity, and structure; with LaTeX formatting; with generating figures and tables for the report; with understanding and debugging the code of Molinelli et al. [7]; and with writing scripts to analyze results, which we interpreted independently. All outputs – textual, code, or visual – were critically reviewed and revised by us, who retain full responsibility for the accuracy, originality, and integrity of the final work.

9 Conclusion and future work

Writing useful test assertions requires reasoning about a focal method's intended behavior, and automating it can relieve developers of a demanding part of writing tests. AsserT5 [10] showed that fine-tuning CodeT5-large on focal- and test-method pairs reaches state-of-the-art exact-match accuracy on this task. Two gaps, however, remained open. First, exact match assumes importance to the original developer-written assertion, even though a syntactically different assertion may detect the same or more faults. Second, the fine-tuning procedure had only been evaluated on the largest CodeT5 variant. We therefore re-evaluated assertion quality through mutation score, which measures how many injected

faults the generated assertions detect, and examined how model size within the fine-tuned CodeT5 family affected assertion quality. Using the top-1 prediction of each model and the mutation-testing methodology of Molinelli et al. [7], we compared fine-tuned CodeT5-small, base, and large against one another and against larger code-specific LLMs (Qwen2.5-Coder 3B, 7B, and 14B) across 10 real-world open-source Java projects.

Within the CodeT5 family, smaller models performed slightly better (46.2%, 46.1%, and 46.0% for small, base, and large). The generated assertions of CodeT5-small were the most conservative of the three: by staying close to the test prefix it referenced fewer symbols that do not exist and compiled more often, which in one project (`nbvcxz`) improved its mutation score slightly over the larger variants. It reached this quality far more efficiently than CodeT5-large, at 4x lower median inference time and 12x lower memory consumption. Larger code-specific models (Qwen2.5-Coder 3B, 7B, and 14B) were expected to outperform the smaller fine-tuned models. However, only Qwen-14B surpassed the CodeT5 models on mutation score by 0.6 percentage points. That advantage was concentrated in two projects, where Qwen-14B generated a memorized constant and a focal-method expression that the CodeT5 models did not. Thus, a 60M-parameter fine-tuned model achieves most of the fault-detection value of a model that uses 38x more memory and is 2.6x slower in median inference time.

Future work could develop and compare more sophisticated truncation strategies for the fine-tuned CodeT5 models to improve compilation rates. The same models could also be quantized, at the cost of a potential small reduction in quality in exchange for lower memory usage and latency. Finally, the evaluation could be broadened, both to more projects and beyond the top-1 prediction, by incorporating the top-5 and top-10 candidates in the evaluation.

Acknowledgments

I want to thank my supervisors Annibale Panichella and Mitchell Olsthoorn for their continued input during the research.

References

[1] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

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

[3] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[4] Soneya Binta Hossain and Matthew B Dwyer. 2025. Togll: Correct and strong test oracle generation with llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 1475–1487.

[5] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[6] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2011), 649–678.

[7] Davide Molinelli, Luca Di Grazia, Alberto Martin-Lopez, Michael D Ernst, and Mauro Pezze. 2025. 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)*. IEEE, 278–290.

[8] Ollama. 2025. *Ollama*. https://ollama.com/

[9] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in computers*. Vol. 112. Elsevier, 275–378.

[10] Severin Primbs, Benedikt Fein, and Gordon Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. In *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*. IEEE, 12–23.

[11] Jiho Shin, Hadi Hemmati, Moshi Wei, and Song Wang. 2024. Assessing evaluation metrics for neural test oracle generation. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2337–2349.

[12] André Silva, Nuno Saavedra, and Martin Monperrus. 2024. GitBug-Java: A Reproducible Benchmark of Recent Java Bugs. In *Proceedings of the 21st International Conference on Mining Software Repositories*. doi:10.1145/3643991.3644884

[13] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2Test: A dataset of focal methods mapped to test cases. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 299–303.

[14] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. doi:10.18653/v1/2021.emnlp-main.685

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