# Creating Local LLMs for Test Assertion Generation: A Comparative Study of Knowledge Distillation from CodeT5

Georgi Dimitrov[1]

Supervisor(s): Mitchell Olsthoorn[1], Annibale Panichella[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 22, 2025

Name of the student: Georgi Dimitrov
Final project course: CSE3000 Research Project
Thesis committee: Mitchell Olsthoorn, Annibale Panichella, Petr Kellnhofer

# Creating Local LLMs for Test Assertion Generation: A Comparative Study of Knowledge Distillation from CodeT5

Georgi Dimitrov

Delft University of Technology
Delft, The Netherlands

## Abstract

Effective test assertions are important for software quality, but their creation is time-consuming. While Large Language Models (LLMs) show promise in automated assertion generation, their size, cost, resource demands, and need for online connection often render them impractical for widespread developer use. Knowledge Distillation (KD) offers a solution to bridge that gap by transferring capabilities from a large "teacher" LLM to smaller "student" models (SLMs). However, the majority of the ground work on KD has been focused on classification tasks and not on generative problems. This paper investigates the feasibility of a test assertion generation task using response-based Knowledge Distillation (KD) from a CodeT5-base teacher. We specifically explore the impact of three parameters on assertion quality and model efficiency - those being student model size (number of layers), pretraining initialization, and loss weighting. Our results demonstrate that distilled small student models (231 MB), particularly those initialized from pretrained checkpoints and fine-tuned with specific loss weight ($\alpha = 0.5$) for the ground truth and distillation losses, can retain a significant portion of the teacher's assertion generation performance when considering the defined metrics - achieving around 83.9% of the CodeBERTScore of the teacher with just 25.9% of the size. This work provides empirical insights into creating specialized SLMs for test assertion generation, highlighting practical configurations for deployment in development environments.

## 1 Introduction

Software testing is crucial for developing reliable and quality software [15]. An important, yet very time consuming part of testing involves writing effective test assertions [31]. While techniques like Search-Based Software Testing (SBST) have demonstrated effectiveness in generating assertions [13], they typically generate them based on observed runtime values, assuming the initial correctness of the software. As a result, developers need to manually review the generated assertions for correctness in order for them to lead to effective tests. The recent advancements in Large Language Models (LLMs) allow for new approaches to assertion generation. Studies have shown their potential in generating more readable and semantically richer assertions [11, 20], and specifically in enhancing test assertion generation by leveraging their understanding of natural language and code [30], potentially reducing some aspects of manual review compared to traditional techniques like SBST.

However, state-of-the-art LLMs (e.g. GPT-4) often cannot run on a single machine or require online connection, making them impractical for developers. The usage of cloud-based LLMs also leads to privacy concerns [3]. Locally deployable models address those concerns directly. They also have many use cases, among which is in-IDE offline code generation [26], and more specifically assertion generation. While locally deployable models (e.g., DeepSeek Coder 6.7B) could perform well when generating assertions for tests, bigger local models still require significant local resources which reduces their practicality (with DeepSeek Coder 6.7B in particular requiring around 16GB of VRAM).

Knowledge Distillation (KD) [27] offers a promising approach to bridge this gap. KD involves training a smaller "student" model to mimic the behavior of a larger, more capable "teacher" model. This approach aims to enable efficient, high-quality test assertion generation without the computational cost of large LLMs. Furthermore, state-of-the-art KD techniques have shown to be able to outperform small models trained from scratch [2]. However, most of the ground work on KD so far has focused on classification tasks [12, 22], and not on generative ones.

This paper investigates the feasibility of distillation of a CodeT5-base teacher to create efficient student models for test assertion generation. We specifically explore trade-offs in assertion quality and model efficiency arising from three different factors of the student model configuration: model size (number of layers and parameters), pretraining initialization, and loss function weighting (balancing ground truth learning against mimicking the teacher). Our primary goal is to identify practical configurations for deploying small, local models that retain a lot of the performance of the teacher model.

Our approach is an empirical evaluation after employing a response-based KD [21]. We use CodeT5 (codet5-base version) as the teacher model. We train multiple student models - smaller than 1GB and with different sizes, initializations, and loss functions, that are more thoroughly explained together with the experiment setup. The evaluation is done by comparing assertion quality (using metrics like CodeBLEU [18] and CodeBERTScore [32]) and computational efficiency (based on model size and inference speed) against the teacher model and other baselines.

Our contributions can be summarized as follows:

- An empirical investigation into the feasibility of using knowledge distillation from CodeT5-base to create efficient, local student models for test assertion generation.
- A systematic evaluation of student model configurations, analyzing the impact of initialization, loss function weighting, and model size on the quality-efficiency trade-off.
- A publicly available replication package including model configurations and evaluation scripts [5, 6].

## 2 Background and Related Work

In this section, we describe the background knowledge required to understand the paper and discuss the related work.

## 2.1 Software Testing and Test Assertions

Software testing is a phase of the software development lifecycle [8]. It is crucial for maximizing reliability, quality and correctness of software systems [15]. A core component of that phase is testing the software system under predefined user inputs and then verifying the outputs against the expected outcomes. The verification part usually happens through test assertions [31].

A test assertion is a predicate that evaluates to true or false based on the current state of what is being checked. It is embedded within a test case and checks if a specific condition holds true at a particular point during the test's execution. Effective assertions are critical as they precisely define the expected behavior and are the final arbiters of whether a test passes or fails. Well-crafted assertions directly target the functionality under test. However, writing such effective assertions manually is a notoriously time-consuming and error-prone task for developers, often becoming a bottleneck in the testing process [31]. This challenge motivates the exploration of automated approaches for assertion generation.

## 2.2 Automated Test Assertion Generation

Techniques like Search-Based Software Testing (SBST) have shown promise in automatically generating test inputs to achieve high code coverage [13]. However, SBST derives assertions from observed runtime values of the program under test, implicitly assuming the initial correctness of the code. This means that manual review and refinement by developers is required to ensure the assertions actually reflect the intended behavior.

Other traditional techniques range from dynamic analysis that infers assertions from observed program behavior [7] to static analysis that identifies properties from source code [23]. While effective in certain contexts, these methods often face limitations in scalability, handling complex program logic, or once again requiring significant manual effort. More recently, data-driven techniques, including those leveraging information retrieval, have been explored to retrieve and adapt existing assertions from code repositories [29]. Such techniques rely on the existence of assertions that are similar enough to be adapted.

Recent advancements in Large Language Models (LLMs) have opened new opportunities for automating various software engineering tasks. LLMs, mainly based on the Transformer architecture [24], are pretrained on a significant dataset of text and source code, enabling them to understand and generate human-like text and syntactically valid code [30]. Their capabilities extend to code generation, code completion, bug detection, and, relevant to this work, automated test generation. Several studies have explored the potential of LLMs in generating entire test cases and, more specifically, test assertions. For instance, LLMs' ability to improve the readability of generated tests has been demonstrated [11], and an empirical evaluation of using LLMs for automated unit test generation has been provided [20]. Focusing on assertions, LLMs can leverage their understanding of natural language and code to enhance assertion generation [30]. These studies highlight the promise of LLMs but also implicitly point to limitations when using the most powerful models (e.g., GPT-4), such as high operational costs, inference latency, and reliance on online APIs, which can raise privacy concerns for private codebases.

## 2.3 Knowledge Distillation Principles

Knowledge Distillation (KD) is a model compression technique that aims to transfer the "knowledge" from a large "teacher" model to a smaller "student" model [12, 27]. The primary goal is to create a student model that retains a significant portion of the teacher's performance for a specific task, while being substantially more efficient in terms of size and inference speed. This makes KD particularly attractive for deploying powerful models on devices with limited resources or in latency-sensitive applications. However, most of the KD work so far has been done with a focus on classification tasks [12, 22].

The knowledge transfer in KD can take several forms. Response-based KD, which is the primary approach in this study, trains the student to mimic the teacher's output. This involves matching the teacher's soft probability distributions over the output vocabulary (using metrics like KL divergence) or matching the teacher's hard predictions (argmax outputs, using cross-entropy). Other forms include feature-based KD [19], where the student learns to reproduce the teacher's intermediate layer activations, and relation-based KD [16], focusing on relationships between different parts of the teacher's representations.

The application of KD to LLMs, particularly for code-related tasks, is an active area of research [27]. Knowledge Distillation in theory can help the student model retain more general "knowledge" from the teacher, even while training for a specific task - in our case, a student model can develop an "understanding" of natural language and code generation even while being specifically trained to only replicate the teacher's output for an assertion generation task. Therefore, studies have shown that well-executed distillation can lead to student models that outperform models trained from scratch on the same data, especially when the teacher provides a richer, smoother learning signal [2]. However, effectively distilling complex reasoning or generation capabilities for nuanced tasks like test assertion generation into very small student models remains a challenge and requires careful exploration of different student configurations.

Our work utilizes CodeT5, specifically the codet5-base variant, as the teacher model. CodeT5 is an encoder-decoder LLM [24] pretrained on code and code-related natural language text, making it proficient in code understanding and generation tasks [25].

## 2.4 Evaluating Generated Code

In order to empirically evaluate the different configurations of the student model, we need suitable metrics. One metric in particular is CodeBLEU (BLEU [9] - Bilingual Evaluation Understudy), which has shown promise in evaluating code [18]. The BLEU metric works by dividing the output code into tokens (in our case - using the model's tokenizer), and then comparing similarity to the tokens of the ground truth using n-gram comparison (n usually going from n=1 to n=4). It is a more refined metric compared to percentage of matching assertions (which is too strict) or number of matching symbols per assertion (which leads to high similarity between two opposite assertions). Leveraging the model's tokenizer allows BLEU, despite lacking inherent understanding of the code's purpose, to gain some insight into the significance of the tokens based on how the model itself interprets the data. Furthermore, it's worth

noting that while BLEU focuses on surface-level similarity, it has shown the ability to correlate with the functional correctness of the generated code, especially when combined with appropriate training methodologies [32]. CodeBLEU consists of four parts - n-gram matching (which is the described BLEU metric), weighted n-gram matching (which is similar to the BLEU metric, but the weighting involves assigning weights to n-grams based on their frequency or importance), syntax match (a score derived by comparing the ASTs [1] of the generated and ground truth code) and data flow (which we will not focus on in this paper).

Another metric that was used is CodeBERTScore, which has proven to be one of the most effective methods to evaluate code generation. It uses a CodeBERT (in our case the microsoft/graphcodebert-base) model to find the embeddings of the generated and the reference code and then compare those embeddings. This metric greatly benefits from CodeBERT's specific training for code-related tasks and natural languages [10].

## 3 Methodology

This section details the methodology that was followed during the distillation process of the CodeT5 (codet5-base) teacher model for the task of automated assertion generation. Here, we describe our distillation pipeline, the dataset that was used for the training, the configurations of the teacher and the student models, the training process itself, and the evaluation metrics that were used. We will cover key decisions that were made for the methodology and the reasoning behind them.

### 3.1 Distillation Pipeline

Our approach is centered on response-based KD. The core idea is to train a student model to mimic the output of a CodeT5 (codet5-base) teacher model. For this, the general pipeline (as shown in Figure 1) involves:

(1) Finding a dataset of code snippets.
(2) Preprocessing the dataset.
(3) Fine-tuning the CodeT5 teacher model for the purpose of assertion generation.
(4) Using the CodeT5 teacher to generate test assertions for the preprocessed snippets.
(5) Training various student model configurations on the same input, using the output of the teacher model and the ground truth in order to calculate the loss.
(6) Evaluating the performance and efficiency of the distilled student models.

For the distillation process itself, we primarily focus on transferring knowledge via the teacher's logits (output probability distributions for token generation), complemented by an optional loss component against ground truth assertions when available and applicable. This means that for the scope of this research, we do not consider the teacher's hard predictions (argmax outputs).

We will cover each of the pipeline steps more thoroughly in the other subsections of this section.

### 3.2 Dataset

The original dataset used comes from a Zenodo replication package - specifically "AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model (Replication Package)" [17]. This dataset contains Java codebases that have tests which we assume to be correct. The dataset then underwent preprocessing to suit our needs, with the final dataset containing the following data for each entry:

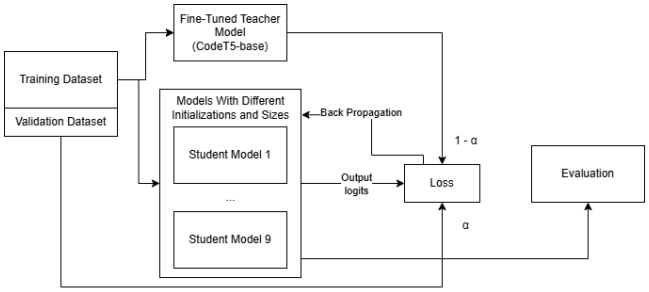


**Figure 1: Distillation pipeline.**

- **The test with masked assertions** - the assertions were masked so the student model can generate them instead.
- **The original assertions** - used for the loss function and for some evaluation metrics (such as accuracy).
- **The focal method** - the method that is being tested, this allows for the model to have some extra context. By giving just the focal method instead of the focal file we reduce the risk of going over the model's context window.
- **Teacher generated assertions** - while not used for teacher hard loss, this is still part of our dataset, allowing us to easily run evaluation metrics on the teacher without having to rerun the entire model to generate those assertions.
- **Teacher logits** - The teacher's output logits for the same focal method and masked test as input, compressed using the LZ4 algorithm [28].

Furthermore, the dataset was filtered so that only tests that only use the focal method (and not other methods of the focal file) remain. This was again done due to the smaller context window of CodeT5, and only influences the performance of the teacher model - meaning that it has no impact on the knowledge distillation itself, which is the core topic of this paper.

The assertions generated by the teacher and the teacher logits were derived from the output of a fine-tuned CodeT5-base model. The teacher model itself was fine-tuned on the rest of the fields of the dataset - using the masked test and focal method as inputs, and using the original assertions for the loss function. Due to the computational power needed, the teacher fine-tuning and collection of the output logits was done on a server provided by TU Delft, while the student model training was done on another device. Therefore, the data needed to be transferred, which is why LZ4 compression was used.

Finally, the dataset was split into a training and a validation subset, with sizes of 18000 and 2000 entries respectively, in order to allow for everything described in this paper to be fully reproducible.

### 3.3 Teacher model: CodeT5 (codet5-base)

The teacher model, as already mentioned, is CodeT5, and more specifically the codet5-base variant. We utilize the pretrained checkpoint Salesforce/codet5-base available through the Hugging Face

Transformers library. CodeT5-base, with 220 million parameters, serves as a strong baseline for code understanding and generation tasks, while being small enough to efficiently fine-tune and gather data for the training of the student model. Another reason for the choice of this model is that its outputs have smaller size than other popular models (like DeepSeek Coder 6.7B). The outputs are smaller due to the context window of the model and its embedding size. This is important for our process due to the data transfer between devices that was already described.

## 3.4 Student models

All student models were derived from CodeT5-small (60M parameters), which is the smaller version of CodeT5-base (220M parameters), allowing us to evaluate distillation on a much smaller model. When changing the size of the student model, we changed the number of encoder and decoder layers only. This allowed for a direct comparison of scaling effects and the benefits of pretraining, while staying within the same architectural family, using the same tokenizer.

Student models can be initialized randomly or from pretrained checkpoints. Starting from a pretrained model (e.g., a smaller variant of the teacher's architecture) often provides a better starting point, potentially leading to faster convergence and superior final performance, as the model already possesses some foundational language or code understanding. We compared the Salesforce/codet5-small checkpoint to an equivalent architecture that was not pretrained, in order to learn the effects of pretraining the model before KD.

Another crucial factor is the student model size. There is an inherent trade-off: smaller models are more efficient but generally have lower capacity, potentially limiting their ability to learn complex patterns from the teacher or achieve high performance. We experimented with three different sizes, with 3, 6, and 12 layers (both the number of encoding and decoding layers). This meant experimenting with 38M, 60M, and 105M parameters.

A key aspect of KD is the design of the loss function. The total loss for training the student often combines a distillation loss (measuring how well the student mimics the teacher) and a student loss (measuring how well the student performs on the ground truth task, typically using standard cross-entropy). These are often combined via a weighted sum: $L_{\text{total}} = \alpha \cdot L_{\text{student}} + (1 - \alpha) \cdot L_{\text{distill}}$, where $\alpha$ is a hyperparameter balancing the two objectives. We compared models with weight for the distillation loss 1.0, 0.75, 0.5, 0.25 and 0.0 respectively. This allowed us to indirectly compare knowledge distillation ($\alpha = 0.0$) to normal training based on ground truth ($\alpha = 1.0$).

This led us to researching the following student models - five models for investigating the $\alpha$ weight parameter, two for doing the same on randomly initialized models, and two more for investigating the impact of different sizes of the models:

- **Student Model 1** - pretrained, $\alpha = 0.0$, 6 layers
- **Student Model 2** - pretrained, $\alpha = 0.25$, 6 layers
- **Student Model 3** - pretrained, $\alpha = 0.5$, 6 layers
- **Student Model 4** - pretrained, $\alpha = 0.75$, 6 layers
- **Student Model 5** - pretrained, $\alpha = 1.0$, 6 layers
- **Student Model 6** - randomly initialized, $\alpha = 0.0$, 6 layers
- **Student Model 7** - randomly initialized, $\alpha = 1.0$, 6 layers
- **Student Model 8** - randomly initialized, $\alpha = 0.0$, 3 layers
- **Student Model 9** - randomly initialized, $\alpha = 0.0$, 12 layers

## 3.5 Training Process

Student models were trained in the following way:

*3.5.1 Loss Function.* The total loss $L_{\text{total}}$ for training the student models is a weighted combination of a distillation loss $L_{\text{distill}}$ and, for certain experiments, a ground truth loss $L_{\text{student}}$: $L_{\text{total}} = \alpha \cdot L_{\text{student}} + (1 - \alpha) * L_{\text{distill}}$

- Distillation Loss ($L_{\text{distill}}$): This component encourages the student to mimic the teacher. It is calculated as the Kullback-Leibler (KL) divergence between the softened probability distribution of the teacher's logits (using a temperature $T_{\text{distill}} = 1.0$) and the student's logits.
- Ground Truth Loss ($L_{\text{student}}$): When human-written ground truth assertions are used, this loss is the standard cross-entropy between the student's predicted token probabilities and the one-hot encoded ground truth assertion tokens.
- Weighting ($\alpha$): We experimented with different values of $\alpha$, specifically 0.0, 0.25, 0.5, 0.75. 1.0, to understand the impact of balancing direct learning from ground truth versus mimicking the teacher. When $\alpha = 0$, only distillation loss is used.

*3.5.2 Optimizer and Hyperparameters.* All models were trained using the AdamW optimizer [14] with a batch size of 4 per GPU, weight decay of 0.01, and a learning rate of 5e-5. Training was conducted for a maximum of 5 epochs, as overfitting was noticed afterwards. Early stopping was based on validation loss with a patience of 3 epochs.

# 4 Study Design

This section outlines our research questions and the experimental design that was used to answer them. Our experiments follow the distillation pipeline and model configurations detailed in Section 3 (Methodology).

## 4.1 Research Questions

For all of the following research questions, assertion quality will be measured by similarity, accuracy, parsability, CodeBLEU, and CodeBERTScore, while computational efficiency will be measured by student model size and inference speed. Those metrics are described in the Dependent Variables subsection (4.2.2).

(1) How does the assertion quality and computational efficiency of a distilled student model compare to its CodeT5-base teacher?
(2) To what extent does KD improve the assertion generation quality of a small pretrained model (codet5-small) compared to its baseline performance before task-specific training?
(3) What is the impact of varying the loss function weight $\alpha$ (balancing ground truth vs. teacher signals) on the assertion quality of distilled pretrained student models?

(4) For randomly initialized student models, how does pure knowledge distillation ($\alpha = 0.0$) compare to pure ground-truth fine-tuning ($\alpha = 1.0$) in terms of assertion quality and learning?
(5) How does varying the size of randomly initialized distilled student models impact the trade-off between assertion quality and model efficiency?

## 4.2 Variables

*4.2.1 Independent Variables.* The independent variables that differentiated the models were model size, loss function weight ($\alpha$), and model initialization as described in the student models section (Section 3.4).

*4.2.2 Dependent Variables.*

- **Similarity** - Similarity of the original assertions and the predicted assertions, measured by longest matching sequence of characters as a percentage of the total length.
- **Accuracy** - Percentage of generated assertions that completely match the original assertions as a string.
- **CodeBLEU** - Calculated using the Python codebleu library. Described more thoroughly in the Background and Related Work section.
- **CodeBERTScore** - Calculated using the Python bert-score library. Uses the GraphCodeBERT (microsoft/graphcodebert-base) model for embedding generation. Described more thoroughly in the Background and Related Work section.
- **Parsability** - Percentage of the generated assertions that can be parsed into valid Java code using the Python javalang library.
- **Model Size** - The model size represented in MB.
- **Inference Speed** - Average time (ms) to generate a single assertion on DelftBlue [4] on a single Nvidia A100 GPU.

## 4.3 Common Experimental Procedures

For all of the research questions, the evaluation was done on the validation dataset. All defined quality metrics were compared, with a focus on CodeBERTScore and CodeBLEU. Results were presented in a comparative table, containing only the final results of the completely trained models.

## 4.4 Specific Experimental Procedures per Research Question

*4.4.1 RQ1. Performance of Distilled Student Model vs. Teacher Model.* This RQ aims to study how effective KD can be in creating a well-performing student model. The two models that were compared were the teacher model and Student Model 3. All defined efficiency metrics were also added to the comparative table.

*4.4.2 RQ2. Performance of Pretrained Student Model vs. Distilled Pretrained Student Model.* This RQ aims to quantify the improvement gained specifically from the distillation process when starting with a pretrained student model. The two models that were compared were Student Model 3, and the CodeT5-small pretrained model before the distillation was done.

*4.4.3 RQ3. Influence of Loss Function Weight ($\alpha$) on Pretrained Student Models.* This RQ aims to determine the optimal balance between learning from ground truth and learning from teacher logits for a pretrained student model. The models that were compared were Student Models from 1 to 5. A graphic comparing the CodeBERTScore over the 5 epochs for all of the models was included.

*4.4.4 RQ4. Distillation vs. Ground Truth Fine-Tuning for Randomly Initialized Models.* This RQ aims to compare the effectiveness of distillation versus standard ground-truth fine-tuning when starting from a randomly initialized student model of a fixed size. The models that were Student Models 6 and 7. A graphic comparing the parsability rate metric over the 5 epochs for both models was included.

*4.4.5 RQ5. Impact of Student Model Size.* This RQ aims to analyze the trade-off between model size, assertion quality, and training efficiency for student models trained from scratch (random initialization due to the lack of existing CodeT5 architecture pretrained models of different sizes) via distillation. The models that are compared were Student Models 6, 8 and 9. Instead of comparing the final results in a table, here we will have two graphics - one for the change of validation loss (based only on ground truth loss) over epochs, and one for change of CodeBERTScore over epochs.

## 4.5 Technical Specifications

The training of all the student models was done on a high-performance computing cluster node, using an NVIDIA A100 GPU with 4 CPUs and 5GB of RAM per CPU. The final evaluation of the models was done on a personal device (with an NVIDIA GeForce GTX 1050 Mobile GPU and 16GB of RAM), due to the codebleu Python library (and more specifically the 0.7.1 version) not being available on the DelftBlue HPC.

# 5 Results

This section presents the empirical results obtained from our experiments, structured according to the research questions.

## 5.1 RQ1. Performance of Distilled Student Model vs. Teacher Model

We evaluated both models on the same 500 data points from a validation dataset. We present the result in Table 1, including all of the assertion quality and efficiency metrics.

The fine-tuned CodeT5-base teacher establishes a strong upper bound for assertion generation quality. The distilled student model achieves a significant portion of this performance, particularly in semantic metrics like CodeBERTScore (where it retains 83.9% of the teacher's knowledge), while offering substantial improvements in model size (being only 25.9% of the size of the teacher in MB, and just 231 MB in total). The student model performs significantly worse when it comes to accuracy (exact matches in assertions), achieving only 13.4% accuracy compared to the teacher's 44.4%, but its assertions are still mostly parsable code (with a rate of 96.2%) and as already stated the generated assertions seem to be semantically similar to the ground truth.

**Table 1: Distilled Student Model vs. Teacher Model**

| Model | Accuracy (%) | Similarity (%) | CodeBERTScore | CodeBLEU | Parsability rate (%) | Size (MB) | Inference speed (ms) |
|---|---|---|---|---|---|---|---|
| CodeT5-base | 44.4 | 83.9 | 0.875 | 0.664 | 99.2 | 892 | 355 |
| Student | 13.4 | 66.9 | 0.734 | 0.395 | 96.2 | 231 | 118 |

## 5.2 RQ2. Performance of Pretrained Student Model vs. Distilled Pretrained Student Model

The results of the evaluations of both models are presented in Table 2.

As evident from the table, the CodeT5-small model before distillation and fine-tuning performed significantly worse than the distilled student model that we compared it to (and that started from the same checkpoint). Distillation significantly improves all assertion quality metrics.

## 5.3 RQ3. Influence of Loss Function Weight ($\alpha$) on Pretrained Student Models

We trained the pretrained Salesforce/codet5-small checkpoint using five different values for the loss function weight parameter - 0.0 (pure distillation), 0.25, 0.5, 0.75, 1.0 (pure ground truth learning).

*5.3.1 Final performance.* The results in Table 3 indicate that configurations with a mix of distillation and ground truth loss ($\alpha = 0.5$ and $\alpha = 0.75$) slightly outperform pure distillation ($\alpha = 0.0$) and pure ground truth fine-tuning ($\alpha = 1.0$) in terms of CodeBERTScore and CodeBLEU (which, as discussed, are our most sophisticated metrics), though the differences are marginal. Pure distillation ($\alpha = 0.0$) achieves the highest accuracy among these configurations.

*5.3.2 Performance trend over epochs.* Figure 2 illustrates the CodeBERTScore of all five models across the five epochs of training. We can see that the $\alpha = 0.5$ model performs very well even after the first epoch, showing that 0.5 loss function weight yields acceptable results very fast. All of the models seem to begin converging around epoch 4 and they reach a similar result towards the end, even though the $\alpha = 0.75$ model begins outperforming the $\alpha = 0.5$ in the later epochs.

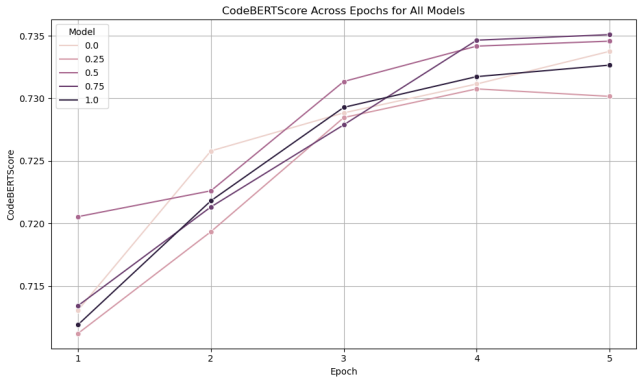


**Figure 2: Comparison of CodeBERTScore over epoch for all five models.**

## 5.4 RQ4. Distillation vs. Ground Truth Fine-Tuning for Randomly Initialized Models

We trained two codet5-small models with randomly initialized weights: one using only distillation loss ($\alpha = 0.0$) and one using only ground truth loss ($\alpha = 1.0$).

*5.4.1 Final performance.* Table 4 compares their final performance after 5 epochs for the two models. When starting from random initialization, both pure distillation and pure ground truth fine-tuning achieve very low absolute performance compared to pretrained models. Pure ground truth fine-tuning ($\alpha = 1.0$) results in slightly higher accuracy and significantly better parsability, while pure distillation ($\alpha = 0.0$) achieves a marginally better CodeBERTScore and similarity.

*5.4.2 Performance trend over epochs.* Figure 3 shows the parsability rate over epochs. The distilled model has a good parsability score of over 90% a lot earlier than the ground truth learning model, but it eventually catches up and surpasses the distilled model.

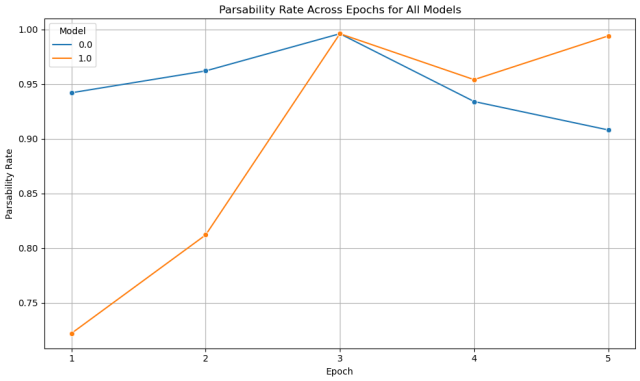


**Figure 3: Comparison of parsability rate over epoch for both models.**

## 5.5 RQ5. Impact of Student Model Size

The validation losses over each epoch for all three models can be observed on Figure 4, while Figure 5 shows the change in their CodeBERTScore. Furthermore, their sizes in MB were respectively 146MB, 231MB and 398MB, making all of them deployable on a single device with a dedicated GPU.

We can notice that the biggest model begins with a lower CodeBERTScore after the first epoch, but catches up and surpasses the others within the five epochs. Furthermore, it seems to have a much lower validation loss, indicating that it is the best model out of the three.

**Table 2: Pretrained Student Model vs. Distilled Student Model**

| Model | Accuracy (%) | Similarity (%) | CodeBERTScore | CodeBLEU | Parsability rate (%) |
|---|---|---|---|---|---|
| CodeT5-small | 0.0 | 15.8 | 0.356 | 0.113 | 15.0 |
| Student | 13.4 | 66.9 | 0.734 | 0.395 | 96.2 |

**Table 3: Student Models with Different $\alpha$ Values**

| Model | Accuracy (%) | Similarity (%) | CodeBERTScore | CodeBLEU | Parsability rate (%) |
|---|---|---|---|---|---|
| $\alpha = 0.0$ | 13.5 | 66.8 | 0.734 | 0.390 | 96.4 |
| $\alpha = 0.25$ | 11.6 | 66.2 | 0.730 | 0.393 | 97.4 |
| $\alpha = 0.5$ | 13.4 | 66.9 | 0.734 | 0.395 | 96.2 |
| $\alpha = 0.75$ | 12.7 | 66.8 | 0.735 | 0.394 | 97.0 |
| $\alpha = 1.0$ | 11.7 | 67.1 | 0.733 | 0.390 | 97.4 |

**Table 4: Randomly Initialized Student Models (Distillation vs. Ground Truth Learning)**

| Model | Accuracy (%) | Similarity (%) | CodeBERTScore | CodeBLEU | Parsability rate (%) |
|---|---|---|---|---|---|
| $\alpha = 0.0$ | 0.3 | 49.2 | 0.590 | 0.190 | 90.8 |
| $\alpha = 1.0$ | 0.4 | 48.6 | 0.579 | 0.191 | 99.4 |

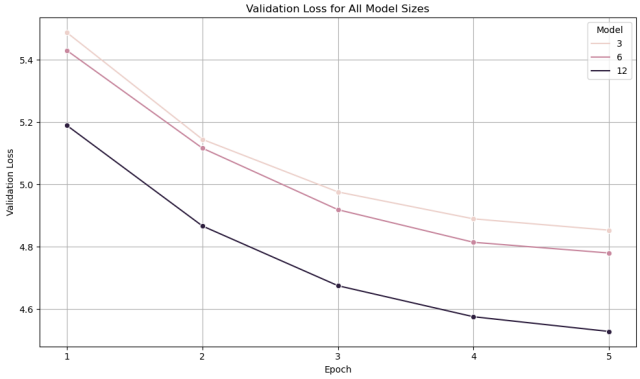


**Figure 4: Comparison of validation loss over epochs for all model sizes.**

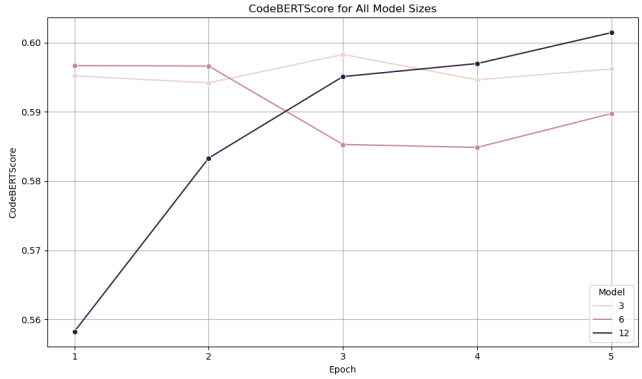


**Figure 5: Comparison of CodeBERTScore over epochs for all model sizes.**

## 6 Discussion

This section interprets the findings presented in Section 5 and discusses their implications, both in the context of the research questions and in the context of the broader question of using KD to create a student model that generates test assertions.

Our experiments demonstrate that KD can effectively transfer assertion generation capabilities from a CodeT5-base teacher to significantly smaller student models. The distilled students, while not matching the teacher's peak performance, offer a compelling trade-off between assertion quality and computational efficiency, making them viable for resource-constrained development environments.

### 6.1 Interpreting Key Findings

Here we interpret the key finding for each research question.

*6.1.1 RQ1.* Student Model 3's ability to retain 83.9% of the CodeBERTScore of the teacher, despite the lower accuracy (13.4% vs. 44.4% for the teacher model), suggests that while students may not replicate assertions identically, they capture much of the intended meaning. The significant reduction in model size (25.9% of teacher's MB size) and faster inference (237ms or 3 times faster) makes this a practical compromise for many applications that focus on semantic correctness and efficiency.

*6.1.2 RQ2.* The contrast between the codet5-small model's performance before and after distillation shows the necessity of task-specific training, even for models pretrained on code. KD proved highly effective in transferring task-specific knowledge for assertion generation, more than doubling the performance of the codet5-small model in all metrics.

*6.1.3 RQ3.* The similar results for all $\alpha$ values suggest that for a pretrained model, the primary bottleneck for further quality improvement might be its inherent capacity (or model size). However, configurations with a mix of distillation and ground truth loss showed slightly better results in some metrics during early epochs (as seen in Figure 2). Furthermore, it is important to consider that pure distillation achieved the highest accuracy, even though this model exclusively learns from a teacher that only has 44.4% accuracy. On top of that, the teacher logits were compressed and later decompressed, meaning that some of that accuracy may be lost during that process. This shows the potential of KD, especially if used on a better teacher model, or distilled to a slightly larger student model architecture.

*6.1.4 RQ4.* When training randomly initialized codet5-small models, the overall performance was considerably lower than their pretrained counterparts, highlighting the significant advantage of pretraining. Interestingly, pure distillation led to a model that achieved a reasonable parsability rate (over 90%) faster than pure ground truth fine-tuning, as illustrated in Figure 3. This suggests that KD helps the student model learn other useful knowledge, such as code understanding. Eventually, the other model surpasses that parsability, which is to be expected, considering it is learning to directly do the original assertions. The CodeBERTScore is higher on the distillation derived model, demonstrating that KD successfully captures some code and natural language understanding from the teacher even on non-pretrained student models.

*6.1.5 RQ5.* The biggest model had a higher potential than the other two models based on the huge difference in validation losses, as illustrated in Figure 4. On top of that, it took more time to learn, possibly due to the higher number of parameters (starting off with the lowest CodeBERTScore as evident in Figure 5), but given enough time it began to outperform the smaller models and did not seem to plateau after 5 epochs like them. This indicates that slightly bigger models can offer a big performance improvement given enough training time. Finally, the varying validation loss difference between the three models also signals a non-linear relationship between model size and performance.

## 6.2 Broader Implications

Our findings reinforce the potential of KD as a viable technique for creating specialized, efficient LLMs for software engineering tasks. Specifically for automated test assertion generation, this study suggests that developers can leverage distillation to create small, locally deployable models that, while not perfectly replicating a large teacher's output, can generate semantically relevant and largely parsable assertions, while being deployable on a wide range of devices.

# 7 Threats to Validity

This section covers several factors that may limit the generalizability and scope of our findings.

The performance of our distilled student models is inherently capped by the quality of the fine-tuned CodeT5-base teacher, which is itself a model that is not too large. If the teacher produces suboptimal assertions, the students will learn these imperfections. To mitigate that, we also evaluated the teacher model's outputs and compared our models to it as a baseline. However, factors like compression of the teacher's logits may also hinder the performance of the teacher after the dataset transfer.

Our experiments utilized a dataset derived from AsserT5, which focuses on Java. The findings might not directly generalize to other programming languages, project types, or different styles of assertions. Furthermore, the data underwent filtering for data entries that focus on a single focal method, which may also impact the results. When it comes to the dataset, we made sure that the validation and training datasets do not overlap in order to assure actual performance evaluation.

# 8 Responsible Research

This research was conducted with a consideration for responsible practices. We have clearly documented the methodology and dataset sources, and we have added files with model configurations for full reproducibility of results (the configurations also include the seeds for the randomness used in the training process). We used publicly available models (like the different CodeT5 versions from Hugging Face). The dataset that we used (AssertT5) is also publicly available and under the Apache License 2.0. Training LLMs can be computationally and resource-intensive. We utilized shared university HPC resources (DelftBlue) for more intensive tasks like training. The smaller student models that we discuss require less resources, potentially having a positive environmental impact. We do not foresee direct negative societal impacts or high risks of misuse from generating test assertions.

# 9 Conclusion and Future Work

Automated test assertion generation has significant importance in enhancing software quality and reducing developer effort. However, the practical application of powerful Large Language Models (LLMs) for this task is often hindered by their computational demands and need for access through an online API. This paper explored the feasibility of using response-based knowledge distillation (KD) to transfer assertion generation capabilities from a CodeT5-base teacher model to smaller student models. Our empirical evaluation systematically explored the impact of student model size, pretraining initialization, and loss function weighting ($\alpha$) on assertion quality and computational efficiency. Key findings show that distilled student models, especially those initialized from the Salesforce/codet5-small pretrained checkpoint, can retain a substantial amount of the teacher model's capabilities of assertion generation. One of the student models in particular (Student Model 3 with $\alpha = 0.5$) retained 83.9% of the teacher's CodeBERTScore, while being just 25.9% of the teacher's size in MB and having around 3 times faster inference speed.

In summary, this work provides empirical evidence that knowledge distillation is a viable strategy for creating specialized, computationally efficient student models capable of generating semantically relevant and largely parsable test assertions.

Based on this research, we can also see a few good directions for future research. While we employed several established metrics, future work could incorporate the mutation score to evaluate the fault-detection capabilities of the generated assertions. This would

involve generating mutants of the focal methods and assessing whether the assertions compile and pass on the original code and fail on the mutants, providing a more direct measure of assertion strength. Comparing pretrained models with different sizes is also something that could be explored, but such models with the CodeT5 architecture are not publicly available at the time of writing this paper. Finally, doing a qualitative analysis and human evaluation would help gain more insights into the quality of the generated assertions and find any patterns associated with it. By pursuing these directions, the development of even more effective and practical small language models for automated software testing tasks can be advanced.

## References


[1] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 368–377. doi:10.1109/ICSM.1998.738528

[2] Minh Duc Bui, Fabian David Schmidt, Goran Glavaš, and Katharina von der Wense. 2024. Knowledge Distillation vs. Pretraining from Scratch under a Fixed (Computation) Budget. arXiv:2404.19319 [cs.CL] https://arxiv.org/abs/2404.19319

[3] Badhan Chandra Das, M. Hadi Amini, and Yanzhao Wu. 2025. Security and Privacy Challenges of Large Language Models: A Survey. *ACM Comput. Surv.* 57, 6, Article 152 (Feb. 2025), 39 pages. doi:10.1145/3712001

[4] Delft High Performance Computing Centre (DHPC). 2024. DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2.

[5] Georgi Dimitrov. 2025. *gdimitrovdev/LLMDistillation: Zenodo Release.* doi:10.5281/zenodo.15712207

[6] Georgi Dimitrov. 2025. *LLMDistillation Dataset.* doi:10.5281/zenodo.15712212

[7] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45. doi:10.1016/j.scico.2007.01.015 Special issue on Experimental Software and Toolkits.

[8] Gerald D. Everett and Raymond McLeod. 2015. *Software testing: Testing across the entire software development life cycle.* IEEE Press.

[9] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. 2022. Out of the BLEU: how should we assess quality of the Code Generation models? arXiv:2208.03133 [cs.SE]

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL] https://arxiv.org/abs/2002.08155

[11] Gregory Gay. 2023. Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter. In *International Symposium on Search-Based Software Engineering*. Springer Nature Switzerland, Cham.

[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[13] Gunel Jahangirova and Valerio Terragni. 2023. SBFT tool competition 2023-Java test case generation track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE.

[14] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[15] Kshirasagar Naik and Priyadarshi Tripathy. 2008. *Software Testing and Quality Assurance: Theory and Practice.* John Wiley & Sons, Hoboken, NJ.

[16] Seungho Park, Jaeho Kim, Sangdoo Yu, and Alice Oh Choi. 2019. Relational knowledge transfer for inductive scene graph generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 5208–5217.

[17] Severin Primbs, Benedikt Fein, and Gordon Fraser. 2024. *AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model (Replication Package).* doi:10.5281/zenodo.14703162

[18] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

[19] Adriana Romero, Nicolas Ballas, Sébastien Kahou, Antoine Chassang, Carlo Guesdon, Yann Deleu, and Aaron C Courville. 2015. Fitnets: Hints for thin deep nets. In *International Conference on Learning Representations*.

[20] Max Schafer et al. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[21] Liangchen Song, Xuan Gong, Helong Zhou, Jiajie Chen, Qian Zhang, David Doermann, and Junsong Yuan. 2023. Exploring the Knowledge Transferred by Response-Based Teacher-Student Distillation. In *Proceedings of the 31st ACM International Conference on Multimedia* (Ottawa ON, Canada) *(MM '23)*. Association for Computing Machinery, New York, NY, USA, 2704–2713. doi:10.1145/3581783.3612162

[22] Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. 2019. Distilling Task-Specific Knowledge from BERT into Simple Neural Networks. arXiv:1903.12136 [cs.CL] https://arxiv.org/abs/1903.12136

[23] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. GoldMine: Automatic assertion generation using data mining and static analysis. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. 626–629. doi:10.1109/DATE.2010.5457129

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[25] 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*. 8696–8708.

[26] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (March 2022), 47 pages. doi:10.1145/3487569

[27] Chuanpeng Yang, Yao Zhu, Wang Lu, Yidong Wang, Qian Chen, Chenlong Gao, Bingjie Yan, and Yiqiang Chen. 2024. Survey on Knowledge Distillation for Large Language Models: Methods, Evaluation, and Application. *ACM Trans. Intell. Syst. Technol.* (2024). doi:10.1145/3699518 Just Accepted.

[28] Yann Collet and others. Accessed 2025-06-02. LZ4 - Extremely fast compression. https://lz4.org/

[29] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. 2022. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 163–174. doi:10.1145/3510003.3510149

[30] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2024. Exploring Automated Assertion Generation via Large Language Models. *ACM Trans. Softw. Eng. Methodol.* (2024). doi:10.1145/3699598 Just Accepted.

[31] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*.

[32] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. arXiv:2302.05527 [cs.SE] https://arxiv.org/abs/2302.05527