Distilling CodeT5 for Efficient On-Device Test-Assertion Generation

Combining response-based distillation and architectural tuning to deliver near-teacher quality on resource-constrained devices

Andrei Vlad Nicula[1]
Supervisors: 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 20, 2025

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

Abstract

Writing clear, semantically rich test assertions remains a major bottleneck in software development. While large pre-trained models such as CodeT5 excel at synthesizing assertions, their size and latency make them impractical for on-premise or resource-constrained workflows. In this work, we introduce a knowledge-distillation pipeline that transfers knowledge from CodeT5-base, a pre-trained encoder–decoder Transformer model based on the T5 architecture, into sub-1 GB student models tailored specifically for test-assertion generation. Our pipeline combines response-based distillation using soft labels and hard-label fine-tuning, and incorporates custom student architectures comparing pre-trained models vs random initialization, along with targeted regularization techniques. We instantiate students at various size points and conduct an empirical evaluation on standard assertion benchmarks, measuring exact-match accuracy, similarity, RAM footprint, and CPU and GPU inference latency. Our best 230 MB student retains over 80% of the teacher's assertion-generation accuracy on exact matches and over 90% of the similarity, while having an inference time of under 3 seconds on a single consumer-grade CPU, with a 75% reduction in RAM usage. These results demonstrate that distilled code-LLMs can deliver near-teacher assertion quality under tight memory and latency constraints, paving the way for fully on-device IDE integration and low-overhead continuous-integration workflows.

Keywords

Knowledge Distillation, Test-Assertion Generation, CodeT5, Model Compression, Software Testing, Large Language Models

1 Introduction

Contemporary software engineering depends on automated testing to verify correctness and prevent defects [29]. In particular, assertions, the code statements that check whether a program's actual behavior matches its expected behavior, are crucial for catching defects early. However, manually defining a comprehensive set of assertion checks for every possible scenario can be both time-consuming and error-prone, which spurred the development of Search-Based Software Testing (SBST) tools to automate the generation of input data and structural test scaffolding effectively [8]. Yet, they typically treat the program under test as a black box, observing runtime outputs and generating assertions solely based on those observed values. This approach fails to capture deeper semantic invariants of the code, and as a result, developers must manually inspect and refine generated assertions for correctness and expressiveness, an activity that can quickly become a bottleneck in large codebases [36].

Large language models (LLMs) trained on code, such as GPT-4 [16] and CodeT5 [30], show promise in synthesizing human-readable, semantically rich assertions [5, 23]. Fine-tuned code models like AsserT5 and AugmenTest further illustrate the potential of this approach [11, 18]. However, their hundreds of billions of parameters lead to high inference latency, substantial resource demands, and the need for cloud connectivity, limitations for on-premise or resource-constrained workflows.

In contrast, models with fewer than 100 million parameters (< 400 MB on disk) can run locally with minimal latency but often lack the nuanced reasoning required for accurate assertions [9, 22]. Knowledge distillation bridges these extremes: by training a compact "student" to mimic a large "teacher," it transfers semantic knowledge while drastically cutting resource requirements [6]. Prior work has shown students under 1 GB can retain over 95% of teacher performance on related code analysis and classification tasks [17, 24], whereas we train for generative test-assertion synthesis.

In this thesis, we focus on distilling **CodeT5** into sub-1 GB students for automated test-assertion generation. Our main research question is:

To what extent can we distill CodeT5 into a compact student (under 1 GB) that retains at least 90% of the teacher's assertion-generation accuracy while reducing inference latency on CPU hardware?

We further decompose this into four sub-questions:

(1) **Feasibility of Distillation:** To what extent can a distilled model grasp the natural-language and code semantics involved in test-assertion synthesis?
(2) **Teacher vs. Student:** How much of the teacher's accuracy does the student retain, and at what memory cost?
(3) **Size–Performance Trade-off:** What is the smallest viable student architecture that still achieves "adequate" test-generation capabilities?
(4) **Device Compatibility:** How do distilled students perform across a range of developer machines, from low-end laptops to high-end workstations?

We evaluate our distillation pipeline on the Method2Test assertion-generation benchmark, comparing CodeT5 (the teacher) against the distilled student models with different configurations, and explore the impact of regularization techniques. For each model, we measure assertion accuracy and similarity, inference latency on CPU and GPU hardware, and memory footprint.

Our experiments show that the 230 MB student retains 81% of the teacher's assertion accuracy on exact matches and above 90% of the teacher's similarity while reducing RAM usage by 75% and having an average inference time of under 3 seconds on an Ultrabook Laptop.

Our contributions are:

- **A Distillation Pipeline** for transforming CodeT5 into lightweight students, complete with preprocessing, loss functions, and training recipes [15].
- **Quantitative Evaluation** of retained accuracy, similarity, and inference time across devices using the Method2Test dataset [27].
- **Pre-training vs. Custom Architectures and Regularization** comparing students initialized from CodeT5 pre-training against custom models trained from scratch, and studying the effect of different regularization techniques on assertion synthesis performance.

The remainder of this thesis is organized as follows. Section 2 presents background on assertion generation, SBST, and knowledge distillation. Section 3 details our distillation methodology. Section 4 describes study design, parameters, the experimental setup, and metrics. Section 5 reports results and discusses trade-offs. Section 6 examines threats to validity, Section 7 presents our approach to

responsible research, and Section 8 concludes with future research directions.

2 Background

Automated test-assertion generation sits at the intersection of software testing, deep learning for code, and model compression techniques. To ground our contribution, we first review Search-Based Software Testing (SBST) and the long-standing oracle problem (2.1). We then survey the evolution of assertion-generation approaches, from early RNN models to template-based classifiers and fine-tuned transformers, culminating in large LLM-driven techniques (2.2). Finally, we cover the principles of knowledge distillation (2.3), including its widespread application in NLP classification and emerging efforts to compress code-specialized models. This layered background highlights the gap we address: the absence of distilled code-LLMs tailored to test-assertion synthesis under tight resource constraints.

2.1 Search-Based Software Testing and the Oracle Problem

Search-Based Software Testing (SBST) employs metaheuristic techniques, using genetic algorithms, hill-climbing, or simulated annealing, to automatically generate test inputs that maximize coverage or reveal defects. However, SBST tools such as EvoSuite [4] treat the program under test as a black box and synthesize assertions based solely on observed executions. This "oracle problem", determining whether a test outcome truly reflects a fault, remains a major bottleneck, as generated assertions often require laborious manual refinement to capture semantic invariants [2].

A *test assertion* is a snippet of code, e.g., JUnit's `assertEquals(a,b)`, `assertTrue(condition)` or `assertNotNull(object)` that validates behavior at runtime and throws an error if violated [10].

To overcome the oracle problem, researchers have turned to deep-learning models that synthesize assertions directly from code [7].

2.2 Deep-Learning-Based Assertion Generation

Deep-learning methods have been applied to the assertion synthesis problem in three main phases: RNN-based sequence-to-sequence models (2.2.1), Transformer-based architectures (2.2.2), and LLM-driven oracle approaches (2.2.3).

2.2.1 Sequence-to-Sequence and RNN-Based Approaches. Initial work framed assertion generation as a translation task: given a test method and its focal code (the method under test plus its unit-test prefix), a Recurrent Neural Network (RNN) encoder-decoder predicts the assertion [26]. ATLAS abstracts identifiers into generic tokens to reduce vocabulary size, achieving exact matches for 31% of top-1 predictions and 50% in top-5 when trained on project-specific data [31]. ReAssert improved on this by employing precise code-to-test traceability and the Reformer architecture to boost exact match rates to 44% and 51% compile-time matches with the ground truth [32].

2.2.2 Transformer-Based and Pre-Trained Models. Transformer models offer superior capacity for handling code's large vocabularies. TOGA uses a pretrained CodeBERT classifier to select among template assertions [7], while pretrained BART, fine-tuned on code data, further improves generation quality [28]. More recently, AserT5 leverages CodeT5, a unified encoder-decoder pre-trained on code, to fine-tune directly for assertion insertion, achieving exact match rates up to 59.5% on benchmark datasets [18].

2.2.3 LLM-Based Oracle Generation. The latest approaches harness LLMs' in-context learning and retrieval augmentation. AugmenTest employs LLMs with Retrieval-augmented generation (RAG) techniques, which retrieve relevant code snippets or docs at runtime, to infer assertions from code documentation and comments, outperforming template methods with success rates up to 30% versus 8.2% for TOGA [11]. Meta's industrial TestGen-LLM tool deploys LLMs at scale to refine human-written tests, yielding 25% coverage gains and 73% adoption of its suggestions in production [1].

Despite their strong generative performance, these deep-learning approaches rely on pre-trained models with hundreds of millions to billions of parameters. Such scale vastly exceeds the compute and memory budgets of typical local development machines and edge devices.

2.3 Knowledge Distillation

Knowledge distillation (KD) compresses large models into smaller ones while retaining task performance. We outline its theoretical basis (2.3.1), survey general LLM distillation techniques (2.3.2), and discuss specialized methods for code models (2.3.3).

2.3.1 Foundations of Distillation. Hinton et al. introduced distillation by training a compact student on "soft targets", the teacher's probability outputs, blended with true labels, demonstrating improved performance and regularization on tasks like speech recognition and vision benchmarks [6].

2.3.2 Distillation in NLP and LLM Compression. Surveys show that distillation is key to making LLMs deployable under resource constraints. Xu et al. did a comprehensive overview of the LLM distillation methods, from white-box to black-box approaches [33]. Yang et al. detailed KD methods, evaluations, and applications, emphasizing high compression rates with minimal accuracy loss [34].

2.3.3 Distillation for Code Models. Specialized code model compression includes Compressor, which applies genetic algorithms to search for compact architectures and uses KD to shrink CodeBERT into a 3 MB student with negligible drop in clone-detection and vulnerability-prediction performance [24]. MORPH extends this by integrating metamorphic testing and many-objective optimization to produce robust distilled models that balance accuracy, efficiency, and resilience to code changes [17].

Despite successful distillation of code models for classification and clone detection, no prior work has applied these techniques to test-assertion generation under tight resource budgets.

3 Methodology

This section details the approach we took to the distillation of the CodeT5 model. It describes the base dataset (3.1), training and evaluation preprocessing and datasets (3.2), the teacher model (3.3),

the student configurations (3.4), and the core distillation pipeline (3.5).

Our proposed distilled model is built upon the CodeT5 architecture using smaller variants. The model takes as input a test method with the assertions being replaced with a placeholder comment, together with as much of the *file under test* as the context window allows (the maximum number of input tokens), and is expected to output a list of assertions that would have been there.

3.1 Base Dataset: Methods2Test

Methods2Test is a large, supervised dataset pairing real-world Java tests with their corresponding "focal" methods in production code. It comprises 780,944 test-method pairs mined from a total of 91,385 Java open source projects hosted on GitHub with licenses permitting re-distribution [27]. To ensure proper mapping between the focal and test methods, Tufano et al. have implemented a set of heuristics such as: matching each test class to a production class of the same package after stripping “Test” prefixes/suffixes, then aligning method names by removing “test” affixes. If that fails, it intersects method-call sets between test and production classes, accepting only one-to-one matches; ambiguous cases are discarded.

We chose Methods2Test for its large scale, high-quality focal–test mappings, and real-world diversity, built using strict naming- and call-based heuristics that ensure each test truly exercises its intended method; it covers a broad spectrum of commonly used Java assertions in realistic developer tests, rather than synthetic or auto-generated cases; and it is publicly available and well-documented, enabling reproducible experiments and straightforward comparison against prior work.

3.2 Training and Evaluation Dataset

Due to the hardware constraints posed by the memory and computational power of the devices used for this project, we randomly sampled 10000 examples from the Methods2Test corpus. Rather than limiting context to the single focal method, we include the entire class under test, capturing helper methods, field declarations, and imports, to ensure the model has access to all semantic cues that might influence assertion generation.

We then split these 10000 examples into a 90%-10% training and validation partition. The validation set is unseen both during the training of the student model and the fine-tuning of the teacher model. This setup provides sufficient data for distillation while keeping training times tractable on a single A100 GPU, and even lower-end GPUs like the RTX3070.

Each example undergoes targeted augmentation to prepare inputs for both teacher and student models. First, we locate the assertions in the test (e.g., `assertEquals(...)`) and replace them with the comment marker `// ASSERTION PLACEHOLDER` so that the model learns to generate the missing check from context. We then run the masked test plus full class through the fine-tuned CodeT5 teacher, capturing both its **predicted assertion** and the **raw logits**, the unnormalized probability scores across the vocabulary at each token position, that serve as soft targets during distillation.

Since storing full-precision logits for every example would be prohibitively large, we apply a compression pipeline: first, quantizing 32-bit floats to 4-bit and then applying LZ4 entropy coding to the resulting byte stream [35]. This two-step compression reduces storage requirements by over 117 times on average while preserving the relative magnitude information critical for effective student training. This was needed as the teacher model is hosted on a separate machine; transferring its full-precision logits directly or storing them uncompressed would be impractical, so we compress the outputs at the source to eliminate both transfer and storage overhead.

3.3 Teacher Model

We use *CodeT5-base*, a moderate-sized encoder-decoder Transformer pre-trained on code, as the teacher. Its architecture, pre-training objectives, and token-level logits provide the rich semantic guidance needed for student distillation without overloading the hardware used for teacher inference.

CodeT5-base is a unified encoder-decoder Transformer that incorporates both code understanding and generation as text-to-text tasks. It closely follows the T5 architecture, stacking 12 self-attention layers in both the encoder and decoder, with hidden size 768, feed-forward inner dimension 3072, and 12 attention heads per layer, totaling approximately 220 million parameters and a 900MB memory footprint. [20].

During pre-training, CodeT5 learned from 8.35 M code functions across eight languages (including Java) using identifier-aware span-masking and denoising objectives, which teaches it to distinguish developer-assigned identifiers from other tokens and recover masked code snippets [30].

We chose the base variant over *CodeT5-large* (770 M parameters [13]) due to our workstation’s GPU memory and compute limits; *the base*’s 220 M parameters offer a practical trade-off between model capacity and resource usage. In contrast to autoregressive models, which generate extremely large logits vectors, making storage and transfer infeasible even after compression, *CodeT5-base*’s encoder-decoder outputs produce logits of manageable size for student devices. Moreover, we opted for the T5 family because its open-source checkpoints allow us to extract complete token-level logits for every output position, essential for full knowledge transfer, whereas closed-source models like OpenAI’s GPT API return only the top-k log-probabilities per token, making them unsuitable for effective knowledge transfer in the framework of distillation.

To improve the performance of the teacher model and to specialize it on the assertion generation task, we fine-tuned CodeT5-base on our masked test-class-assertion training set for 5 epochs, batch size of 8, using AdamW (learning rate 5×10^{-5}, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and weight decay = 0.01) with a linear learning-rate decay and no warm-up. Due to the memory constraints of both the fine-tuning workstation and the device used for distillation, the input context window was set to 1024, and the output context window (how many tokens the model can generate) was set to 512.

After fine-tuning, each masked example was fed through the teacher to produce the generated assertion text and logits, which were furthermore compressed using the steps detailed in Section 3.2.

3.4 Student Model

We distill two lightweight students derived from the same T5 encoder-decoder framework to probe size-quality trade-offs. **CodeT5-Small** (3.4.1), a pre-trained, open-source variant of CodeT5 that shares the teacher's architecture but at reduced scale, inheriting most of the original model's performance at a fraction of the size [30]. Followed by a **Custom Mini-Student** (3.4.2), an even smaller, custom-designed Transformer (fewer layers, narrower hidden size) initialized from scratch. Though it cannot leverage pre-trained weights, this micro-architecture lets us explore extreme compression points beyond standard checkpoints.

3.4.1 CodeT5-Small. CodeT5-Small is the officially released "small" checkpoint of CodeT5, an identifier-aware encoder–decoder Transformer pre-trained on mixed code and natural-language data. It features 6 layers in both encoder and decoder (versus 12 in the base), a hidden size of 512, feed-forward inner dimension of 2048, and 8 attention heads per layer, amounting to approximately 60 M parameters with a memory footprint of approximately 230 MB. It was pre-trained using the same techniques as the teacher model, CodeT5-Base, presented in Section 3.3 [30].

3.4.2 Custom Mini-Student. To examine even more aggressive compression, we introduce a new variant with a smaller architecture. It features 4 layers in both the encoder and decoder, a hidden size of 256, a feed-forward inner dimension of 1024, and 4 attention heads per layer, having a memory footprint of 60 MB. Because its dimensions diverge from the official CodeT5 checkpoints, we initialize this model with random weights and train it entirely via distillation losses. This enables us to probe both the extreme compression and the capabilities of distillation in training a model from scratch.

Moreover, both students share the teacher's context window: 1024 tokens for the input sequence and 512 tokens for the output sequence.

Both student variants retain full compatibility with CodeT5-base's tokenizer and tokenization logic, eliminating any vocabulary mismatches and ensuring that every token seen during teacher inference is interpretable by the students. By adhering to the same T5-style encoder–decoder framework across teachers and students, we leverage architectural patterns proven effective for code generation, enabling direct performance comparisons. Moreover, empirical benchmarks show that CodeT5 variants outperform other open-source code LLMs, such as PLBART and CodeBERT, on a broad range of generation and understanding tasks, validating our choice of a unified, well-tested architecture for test-assertion synthesis [3, 30].

3.5 Distillation Pipeline

The distillation pipeline trains a student model by feeding it the test methods and any necessary context, and comparing the resulting logits with those of the teacher and the ground truth to establish a unified loss. A visualization of the final pipeline is provided in Figure 1.

At its heart, our pipeline begins by taking each test method with its original assertion replaced by the special token *//ASSERTION*

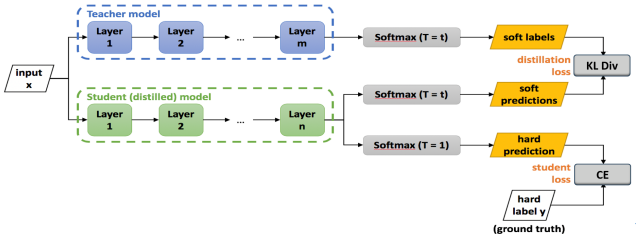

Figure 1: Overview of the distillation pipeline between teacher and student, where T is the temperature, and using the Kullback–Leibler (KL) divergence and the Cross-entropy (CE) as loss functions

PLACEHOLDER and concatenates it with the class under test. Considering the input sequence limit of 1024 tokens, the focal class might be truncated. If the input sequence still exceeds the token limit, the test method might be truncated as well. Afterwards, the Natural Language text is processed by CodeT5's Byte-Pair Encoding Tokenizer and is sent through the student, producing token-level logits, unnormalized scores over the vocabulary at each position, which capture the student's current predictions.

After first decompressing each of the teacher's stored logits back into floating-point score tensors, we quantify the student's error with a composite loss that measures both its divergence from the teacher's softened output distribution and its deviation from the true assertions, and backpropagate the sum to update the student. Concretely, letting z_T and z_S denote the teacher's and student's logits and y the one-hot ground truth, we compute:

$$\mathcal{L}_{\text{soft}} = T^2 \, D_{\text{KL}}\Big(\sigma\big(\tfrac{z_T}{T}\big) \,\Big\|\, \log\Big[\sigma\big(\tfrac{z_S}{T}\big)\Big]\Big), \quad \mathcal{L}_{\text{hard}} = -\sum_i y_i \, \log\big[\sigma(z_S)\big]_i$$

and blend them via

$$\mathcal{L} = \alpha \, \mathcal{L}_{\text{soft}} \; + \; (1-\alpha) \, \mathcal{L}_{\text{hard}}$$

where $\sigma(\cdot)$ denotes softmax (converting logits score into probability distribution), D_{KL} the Kullback–Leibler divergence (a measure of distributional discrepancy) [12], T the softening temperature that controls output entropy (higher T yields softer distributions), and α a weighting hyperparameter balancing the teacher's guidance against true labels [6]. We then backpropagate this combined loss to compute gradients and update the student's parameters using the AdamW optimizer.

To improve convergence and prevent over-confidence or instability, common pitfalls in deep Transformer training, we augment the core pipeline with three refinements.

First, we apply a **learning-rate schedule** that linearly warms up the optimizer's step size from zero to the target rate over a fraction of training steps, then decays it linearly back toward zero over the remainder. Warmup eases the student into high-rate updates, mitigating early training blow-ups while linear decay gradually guides learning to fine-tune weights.

Second, we incorporate **gradient clipping**, enforcing a maximum norm on the backpropagated gradients at each update. By scaling gradients whose l_2 norm exceeds a threshold of 1.0, we prevent exploding gradients from destabilizing training, ensuring smoother convergence.

Finally, we include **weight decay**, a penalty on the l_2 norm of model parameters, to softly discourage large weight magnitudes and promote generalization. During each optimizer step, parameters are nudged toward zero by a small factor, introducing an implicit regularization effect that helps stabilize the student's learning dynamics.

4 Study Design

Building on our methodology, this section details exactly how we instantiated and measured our distillation experiments, including hardware/software environments, training procedures, hyperparameters, and evaluation protocols.

Following directly from the high-level research questions laid out in the Introduction, our study aims to answer the following research sub-questions:

RQ1 To what extent can a distilled model grasp the natural-language and code semantics involved in test-assertion synthesis? Which in practice asks whether a much smaller student network can still effectively minimize the combined distillation loss (KL-divergence plus cross-entropy) and thereby capture the teacher's semantic knowledge.

RQ2 How much of the teacher's accuracy does the student retain, and at what memory cost?

RQ3 What is the smallest viable student architecture that still achieves "adequate" test-generation capabilities? Can a much smaller custom architecture that is not pre-trained still provide acceptable performance?

RQ4 How do distilled students perform across a range of developer machines, from low-end laptops to high-end workstations? Is running such a model on a CPU feasible? What is the inference latency on such setups?

4.1 Training Procedure

We trained each student over the 9,000-example training split described in Section 3.2, using a batch size of 16. The number of epochs was tailored to the model's size and available resources Specifically, CodeT5-Small (pre-trained) was fine-tuned until its validation loss plateaued or began to rise, without regularization and scheduling, this occurred after 4 epochs, while with learning-rate warmup with linear decay, weight decay, and gradient clipping, it stabilized only after 15 epochs. In contrast, the Custom Mini-Student, which started from random weights, was trained for 62 epochs on Google Colab (the practical upper limit given session time constraints), although additional epochs would likely further improve its convergence. At the end of every epoch, we saved a checkpoint, comprising model weights, tokenizer files, and a small JSON metadata object, and immediately evaluated on the 1,000-example validation set using our full suite of generation and parsing metrics, allowing us to monitor overfitting and training evolution. No early-stopping was employed. Gradients were clipped to a maximum norm of 1.0 at each optimizer step, and weight decay (0.01) was applied to all non-LayerNorm, non-bias parameters. Each forward/backward step proceeded as follows (theoretical details explained in Section 3.5):

(1) Load a batch of tokenized inputs (input_ids and attention_mask) and their compressed teacher logits which are decompressed to float32 on-the-fly.
(2) Perform a forward pass through the student to obtain student_logits.
(3) Compute the combined distillation loss (soft KL + hard cross-entropy) using the teacher logits and ground-truth tokens.
(4) Backpropagate the loss, clip gradients, and update student parameters with AdamW.
(5) Step the learning-rate scheduler: we warmed up linearly over the first 10% of total training steps to the peak learning rate, then decayed linearly toward zero by the end of the last epoch.

By checkpointing and validating after every epoch, we ensured that any diverging validation loss or metric degradation could be detected early and attributed to specific training phases.

4.2 Hyperparameters and Scheduling

All experiments used the following hyperparameter configuration:

- **Optimizer:** AdamW with $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and weigth decay of 0.01.
- **Peak Learning Rate:** 10^{-4}, warmed up linearly over the first 10% of total training steps, then decayed linearly until 0.
- **Batch Size:** 16 (training), 12 (validation/test).
- **Gradient Clipping:** l_2 norm capped at 1.0.
- **Dropout:** 0.1 in all Transformer layers
- **Distillation Temperature (T):** 2.0
- **Soft/Hard Loss Weight (α):** 0.7/0.3

These choices reflect standard practices for T5-style models and were verified via preliminary runs to avoid unstable training.

4.3 Experimental Setup

To ensure consistent results, we fixed random seeds across Python's built-in RNG, NumPy, and PyTorch (all set to 42). Training and validation occurred on Google Colab using an NVIDIA A100 GPU (40 GB VRAM) with 83 GB of system RAM. Some evaluation experiments, such as CPU-based inference latency, were run on a personal laptop equipped with an NVIDIA RTX 3070 (laptop variant, 8 GB VRAM), 32 GB RAM, and an AMD Ryzen 7 6800H CPU at 3.20 GHz. To test performance in an extreme setting, an Ultrabook Laptop equipped with an Intel Core i7-1165G7 processor at 2.80 GHz and 32 GB of RAM was used as well.

All experiments were conducted under *Python 3.13.2*. Key Machine Learning libraries included *PyTorch 2.7.0* (installed from the CUDA 12.8 wheel index), *Transformers 4.51.3* (Hugging Face), and *NumPy 2.2.5*. For reproducibility, a *requirements.txt* file is included in the project repository, listing all package versions [15].

4.4 Evaluation Metrics and Protocol

After each epoch's checkpoint, we generated assertions for the validation examples using beam search (beam size = 4, max length = 512). Each generated output was measured against the ground truth, and the following was recorded:

- **Exact-Match Accuracy.** Measures the percentage of generated assertions that exactly match the ground-truth string character-for-character, indicating strict reproduction of the developer-written check. From this benchmark, we further

compute standard classification metrics: precision, recall, and F1 score.

- **String Similarity (SequenceMatcher Ratio).** Uses Python's difflib.SequenceMatcher.ratio() to assign a score between 0 and 1 based on the longest common-subsequence heuristic (Ratcliff/Obershelp) while ignoring "junk" elements like whitespaces, capturing partial overlaps even when assertions are not identical [19].
- **CodeBLEU.** Extends BLEU by incorporating syntax (Abstract Syntax Tree) and semantic (data-flow) matching, yielding a composite score that better correlates with code quality; high CodeBLEU indicates that generated assertions preserve both structure and meaning beyond n-gram overlap [21].
- **CodeBERTScore.** Computes cosine similarity between contextual token embeddings from a CodeBERT encoder, reflecting deeper semantic alignment of code tokens even when surface forms differ, and has been shown to correlate well with human judgments on code generation tasks [37].
- **ROUGE-L F1.** Evaluates the length of the longest common subsequence between generated and reference assertions, balancing precision and recall of that subsequence to reward preserved token order and structure without requiring exact n-gram matches [14].
- **Inference Latency.** Evaluates the time it takes to generate assertions for a single test method. We have considered the four hardware configurations mentioned in section 4.3: A100 GPU, RTX 3070, AMD Ryzen 7 6800H CPU, and Intel Core i7-1165G7. We benchmark the latency by generating assertions for all the validation examples (1000 samples, batch size = 1, sequential execution) on each device. For each setup, we report the mean latency, standard deviation (with Delta Degrees of Freedom equal to 1), and 95% confidence interval, ensuring fair cross-device comparison of student inference speed.

We have evaluated the *CodeT5-base* teacher model using the same benchmarks, recording the assertion quality against the ground truth. To draw our conclusion, we have computed the relative performance between the teacher and student using these metrics and reported them in percentages of relative difference. In the evaluation, only the validation set of 1000 samples was used.

5 Evaluation

In this section, we present a systematic and data-driven evaluation of our distilled assertion models, organized around our four core research questions. We begin by examining whether a compact student can internalize the teacher's semantic knowledge (5.1), contrasting overfitting dynamics with and without regularization. Next, we quantify the trade-off between retained accuracy and deployment cost, measuring exact-match, CodeBLEU, CodeBert Score, and classification metrics alongside model size and inference latency (5.2). Building on that, we explore the lower bounds of viable student architectures, showing how far we can compress before performance degrades unacceptably (5.3). Finally, we demonstrate real-world feasibility by benchmarking inference speed across a spectrum of developer hardware, from high-end GPUs to CPU-only laptops, thereby answering whether these distilled models can run efficiently in everyday environments (5.4). For each Research Question, we provide the results as well as a discussion of their practical implications.

5.1 RQ1: Semantic Grasp via Distillation

5.1.1 Results. To evaluate whether a compact student can truly internalize the rich semantic knowledge of its teacher, we focus on our strongest performing configuration: CodeT5-Small equipped with full regularization as presented in section 3.5. This model was trained for 15 epochs, during which its training and validation loss steadily declined as shown in Figure 2.

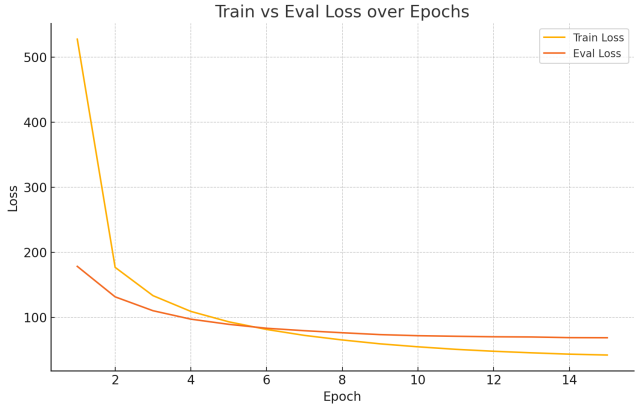

Figure 2: Training vs Validation losses over 15 epochs for the regularized CodeT5-Small student

For direct comparison, we trained an identical CodeT5-Small variant without any regularization or scheduling. The unregularized model achieved the highest performance in epoch 4, but then degrades; from epoch 4 to 5, its F1 score drops from 0.223 to 0.215, and the similarity ratio falls from 0.201 to 0.198. In contrast, the regularized model continues improving through 15 epochs without any decline in quality, as shown in Figure 3.

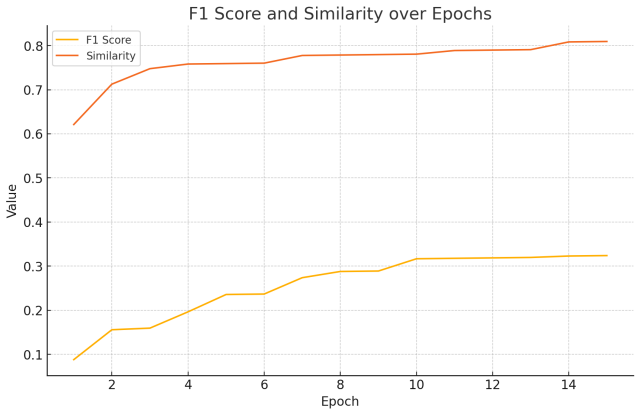

Figure 3: F1 and similarity scores over 15 epochs for the regularized CodeT5-Small student

5.1.2 Discussion. Our distilled student internalizes the semantics of assertion generation: training and validation losses, F1 scores, and similarity metrics all improve steadily through epoch 15, showing that the model is genuinely learning to map test-class contexts to precise, human-readable assertions rather than memorizing patterns. This is further emphasized if we consider an example of generated output from the unseen validation dataset: `assertEquals("Class @Order doesn't annotated with @Order", e.getMessage());` which is a semantically correct Java assertion. By contrast, the unregularized variant overfits by epoch 5, as its F1 and similarity both decline, demonstrating that without learning rate warmup and decay, gradient clipping, and weight decay, the student cannot sustain semantic generalization without directly memorizing inputs.

Conclusion RQ1

The best distilled student is capable of grasping the natural-language and code semantics involved in test-assertion generation if proper regularization techniques are employed, having a training loss of 42.1 and a validation loss of 68.7 after 15 epochs.

5.2 RQ2: Accuracy Retention

5.2.1 Results. Experiments show that our best student (CodeT5-Small with full regularization) retained over 80% of the teacher's exact-match accuracy, while semantic-aware metrics (CodeBLEU, CodeBERTScore, ROUGE-L, SequenceMatcher similarity) remained above 90% as shown in Figure 4. Moreover, this student requires only 230 MB of memory, 75% less than the 900 MB footprint of CodeT5-Base.

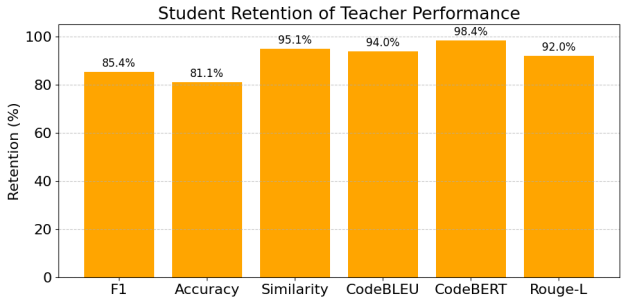

Figure 4: Relative performance of the student against the teacher in: F1, Accuracy, Similarity, CodeBLEU, CodeBERT Score and Rouge-L

5.2.2 Discussion. High retention on semantic metrics indicates that, even when the student's literal output differs from the ground truth, it preserves the underlying meaning of assertions. By contrast, the exact-match metric can be unduly harsh: for example, replacing `Assert.assertTrue(results.isEmpty());` with `Assert.assertEquals(results.size(), 0);` is equally valid but counts as a mismatch. The student's > 90% retention on CodeBLEU and CodeBERTScore shows that it generates semantically equivalent assertions the majority of the time, confirming that it has captured the teacher's deep code understanding despite drastic compression.

Conclusion RQ2

The best distilled student retains more than 90% of the teacher's similarity on the unseen ground truth data while providing a 75% reduction in size.

5.3 RQ3: Viability of Ultra-Compact Architectures

5.3.1 Results. To probe the limits of compression, we trained a Custom Mini-Student from scratch for 62 epochs, reaching a validation loss of 394, an F1 score of 0.09, and a SequenceMatcher similarity of 0.45. Compared to our best CodeT5-Small student (F1 = 0.32, similarity = 0.81), this custom model's F1 is roughly 3.6× lower and its similarity about 2× lower.

5.3.2 Discussion. This experiment reveals practical limitations: distilling a randomly initialized Transformer to any degree of proficiency demands extensive computational resources. Even on Google Colab's A100 GPU, 62 epochs represented our maximum feasible training time, far short of the 150 epochs used to pre-train CodeT5 itself on large-scale corpora with 16 A100 GPUs [30]. In this setting, achieving near-teacher performance from scratch is simply infeasible.

Moreover, the outputs of this mini-student often fail to form valid Java assertions, e.g., `assertEquals("The/foo",key",key);`, which is syntactically invalid and semantically nonsensical. Such failures indicate that without the rich, identifier-aware pre-training of CodeT5, the student cannot grasp the fundamental patterns of test assertions.

Conclusion RQ3

Training a custom mini-student from scratch yielded a performance drop of roughly 3.6× in F1 score and 2× in SequenceMatcher similarity compared to our CodeT5-Small student, demonstrating that, without extensive pre-training, achieving adequate assertion synthesis performance is impractical under our compute constraints.

5.4 RQ4: Cross-Device Feasibility

5.4.1 Results. Our CodeT5-small student model ran stably on all four hardware configurations for over thirty minutes of continuous assertion generation without any crashes or memory errors. Inference latency was measured per test method (i.e., a full forward pass producing all relevant assertions). Table 1 summarizes the mean, standard deviation, and 95% confidence intervals for these measurements across configurations.

5.4.2 Discussion. The fact that our medium-sized student ran reliably for over thirty minutes of continuous assertion generation on every hardware configuration, from high-end A100 GPUs to modest

Table 1: Inference Latency Statistics by Configuration

Configuration	Mean (s)	SD (s)	95% CI (s)
GPU A100	0.6943	0.9270	±0.0575
GPU RTX 3070	1.7204	2.3926	±0.1483
CPU i7-1165G7	1.9086	2.1475	±0.1331
CPU 6800H	2.6557	3.2182	±0.1995

laptop CPUs, demonstrates its broad compatibility and stability in real-world developer environments. Even on the slowest device, the Ryzen 6800H CPU, the mean inference latency remained on the order of two to three seconds per test method, a delay that is perfectly acceptable for on-device assertion generation during active development. This sub-second to low-second response time enables interactive workflows, such as on-the-fly test augmentation in an IDE or quick CI sanity checks on remote builds, without requiring specialized hardware or cloud access.

Conclusion RQ4

It is feasible to run the 230 MB distilled student on a variety of hardware, from low-end Ultrabook Laptop CPUs to high-end GPUs, while keeping the mean inference delay under 3 seconds.

6 Threats to validity

In this section, we discuss potential threats to the internal (6.1), construct (6.2), external (6.3), and conclusion (6.4) validity of our study.

6.1 Internal validity

Data leakage. Although we reserved a held-out 10% validation split that was unseen during both teacher fine-tuning and student distillation, hyperparameter tuning on this same split (e.g., choosing the number of epochs and regularization) may still have introduced subtle information bleed into the final models.

Teacher error propagation. Any systematic mistakes the fine-tuned CodeT5-Base teacher makes are transferred to the student via soft targets, potentially biasing the student toward teacher-specific errors rather than true ground-truth semantics.

Input truncation. Limiting the combined test-method + class context to 1024 tokens can truncate relevant code or test fixtures, reducing the information available to both teacher and student and possibly harming assertion quality in edge cases

6.2 Construct validity

Metric bias. While exact-match accuracy is a strict measure, it penalizes semantically equivalent assertions (e.g., `Assert.assertTrue(results.isEmpty());` vs `Assert.assertEquals(results.size(), 0);`). We complement it with CodeBLEU, CodeBERTScore, ROUGE-L, and SequenceMatcher, but these heuristics may still diverge from human judgments of assertion adequacy.

6.3 External validity

Dataset scope. Our experiments use only the Java/JUnit Methods2Test corpus. Results may not generalize to other languages, testing frameworks (e.g., pytest), or assertion styles in industrial codebases.

Model family. We focus on distilling CodeT5 variants; other architectures (e.g., GPT-style LLMs or different encoder–decoder backbones) may exhibit different size–performance trade-offs.

6.4 Conclusion validity

Sample size and statistical power. We subsampled 10,000 examples for tractability (9,000 train, 1,000 validation). While sufficient for preliminary evaluation, larger or more diverse samples could alter the performance and generalizability of our model, revealing additional patterns.

Hardware variability. Inference-latency measurements were taken on specific CPU/GPU setups; actual performance may vary on other hardware (e.g., different cache sizes, interconnects, or quantization support).

7 Responsible research

Our work builds on openly licensed resources to ensure ethical and legal clarity. The Methods2Test dataset is public and free to use under the MIT License. Moreover, the data gathered by Methods2Test was assembled from 91 K Java projects publicly hosted on GitHub, with licenses permitting redistribution, ensuring that all test-method pairs derive from open repositories and thus uphold ethical, transparent data usage [27]. CodeT5's architecture and checkpoints are available under the BSD-3 License as well as its training data, which was ethically sourced from public GitHub repositories under permissive licenses (e.g., MIT, Apache-2.0, BSD-3-Clause) to ensure transparent and lawful use [30]. This open foundation avoids proprietary barriers and makes our process transparent.

We keep everything reproducible by tracking each step, from collecting and cleaning the input code to training the models, and sharing our scripts, original and processed data used in training and validation, as well as the metadata, training parameters, random seeds, software versions, and hardware setups. Anyone can follow our recipe and verify the results without hidden shortcuts [15].

We're also mindful of energy use. Big models can consume huge amounts of power, so our distillation approach shrinks model size and cuts inference costs, reducing electricity use compared to running larger models on servers [25].

In terms of AI usage, we have leveraged *ChatGPT o4-mini-high* as an auxiliary research assistant to accelerate exploratory literature searches, generate LaTeX scaffolding, suggest code templates, and structure manuscript content, while enforcing strict manual verification and rewriting to uphold scientific rigor. In the literature discovery phase, we prompted the model with queries like "Provide a list of recent peer-reviewed articles on test-assertion synthesis

and code distillation", which quickly surfaced candidate papers that we then manually filtered, discarding non-peer-reviewed materials and verifying each reference to see if they existed, as these models are commonly known to hallucinate references and to also check its relevancy to our work. For LaTeX code generation, we asked ChatGPT to produce initial table and schema templates (e.g., "Generate LaTeX code for a box that will contain text with a title"), then hand-tuned these structures to match our style and thesis guidelines. When drafting code, the AI provided informative snippets, such as a general structure for training a model, but we implemented and reviewed all final code ourselves, using the suggestions strictly as a knowledge resource rather than directly copying any generated text. During manuscript composition, ChatGPT helped outline sections, refine phrasing, and surface synonym alternatives, for instance, "Suggest a structured outline for a 'Study Design' section" and "Propose synonyms for 'we focus on'", yet no AI-generated sentences were inserted verbatim; every phrase was hand-edited and validated for accuracy.

8 Conclusion and future work

Our work presents a practical knowledge-distillation pipeline that compresses CodeT5-Base into sub-1 GB student models specifically tailored for test-assertion generation. The best 230 MB student retains over 80% of the teacher's exact-match accuracy and more than 90% of semantic similarity, while reducing RAM usage by 75% and achieving inference times under three seconds on a consumer-grade CPU. We showed that combining response-based soft-label distillation with hard-label fine-tuning, along with learning-rate scheduling, gradient clipping, and weight decay, is key to preventing overfitting and sustaining semantic generalization across 15 epochs of training. Cross-device benchmarks, from A100 and RTX 3070 GPUs to Ultrabook CPUs, demonstrated stable operation and sub-3-second per-test latency even on modest hardware. These results confirm that a compact student under 1 GB can achieve near-teacher assertion quality while meeting stringent memory and latency constraints, enabling on-device IDE integration and low-overhead CI workflows.

The future directions in this research could broaden and deepen our approach by first scaling up the training corpus, not only sampling the full Methods2Test dataset with its 780 K+ Java–JUnit test–method pairs but also incorporating analogous corpora in other programming languages to validate assertion generalizability beyond Java. Concurrently, we envision an online distillation paradigm in which teacher and student co-reside on the same hardware, removing the compression layer and potentially improving precision through direct soft-label transfer. To assess real-world impact, human-centered studies will combine qualitative evaluations of assertion quality with reinforcement-learning loops driven by developer feedback. On the infrastructure side, access to more powerful or specialized training hardware will enable us to train custom student architectures from scratch, pushing the boundaries of extreme compression without relying on pre-trained checkpoints. Finally, a systematic exploration of hyperparameters and student configurations, via grid and log-spaced sweeps or even evolutionary algorithms, will help identify the optimal trade-offs between size, speed, and accuracy.

Ultimately, these efforts will pave the way for a versatile, resource-efficient assertion-generation framework that seamlessly integrates into developers' workflows and elevates software quality.

Acknowledgments

To the Software Engineering Research Group at TU Delft, especially Professor Annibale Panichella and Professor Mitchell Olsthoorn.

References

[1] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. arXiv:2402.09171 [cs.SE] https://arxiv.org/abs/2402.09171

[2] Maurício Aniche. 2022. *6.4 Search-based software testing*. GitBook. Retrieved 2025-04-28 from https://mordeky.github.io/SQAT/chapters/intelligent-testing/sbst.html

[3] 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

[4] 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

[5] Gregory Gay. 2024. Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter. In *Search-Based Software Engineering*, Paolo Arcaini, Tao Yue, and Erik M. Fredericks (Eds.). Springer Nature Switzerland, Cham, 140–146.

[6] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. arxi v:1503.02531 [stat.ML] https://arxiv.org/abs/1503.02531

[7] Soneya Binta Hossain, Antonio Filieri, Matthew B. Dwyer, Sebastian Elbaum, and Willem Visser. 2023. Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 120–132. doi:10.1145/3611643.3616265

[8] 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)*. 61–64. doi:10.1109/SBFT59156.2023.00025

[9] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. arXiv:1909.10351 [cs.CL] https://arxiv.org/abs/1909.10351

[10] JUnit Team. 2020. *Class Assert*. Retrieved 2025-04-28 from https://junit.org/junit4/javadoc/4.13/org/junit/Assert.html

[11] Shaker Mahmud Khandaker, Fitsum Kifetew, Davide Prandi, and Angelo Susi. 2025. AugmenTest: Enhancing Tests with LLM-Driven Oracles. arXiv:2501.17461 [cs.SE] https://arxiv.org/abs/2501.17461

[12] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79 – 86. doi:10.1214/aoms/1177729694

[13] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. arXiv:2207.01780 [cs.LG] https://arxiv.org/abs/2207.01780

[14] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81. https://aclanthology.org/W04-1013/

[15] Andrei Vlad Nicula. 2025. *DistilT5: Distilled CodeT5 for Assertion Generation*. doi:10.5281/zenodo.15707159

[16] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[17] A. Panichella. 2025. Metamorphic-Based Many-Objective Distillation of LLMs for Code-related Tasks. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*. IEEE / ACM.

[18] Severin Primbs, Benedikt Fein, and Gordon Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. doi:10.48550/arXiv.2502.02708

[19] Python Software Foundation. 2023. *difflib — Helpers for computing deltas*. Python Documentation Team. https://docs.python.org/3/library/difflib.html#module-difflib Python 3.10 documentation.

[20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG] https://arxiv.org/abs/1910.10683

[21] 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

[22] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv:1910.01108 [cs.CL] https://arxiv.org/abs/1910.01108

[23] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[24] Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang, and David Lo. 2022. Compressing Pre-trained Models of Code into 3 MB. arXiv:2208.07120 [cs.SE] https://arxiv.org/abs/2208.07120

[25] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. arXiv:1906.02243 [cs.CL] https://arxiv.org/abs/1906.02243

[26] Weifeng Sun, Hongyan Li, Meng Yan, Yan Lei, and Hongyu Zhang. 2023. Revisiting and Improving Retrieval-Augmented Deep Assertion Generation. arXiv:2309.10264 [cs.SE] https://arxiv.org/abs/2309.10264

[27] 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 (MSR '22)*. ACM. doi:10.1145/3524842.3528009

[28] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. 2022. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test (AST '22)*. ACM. doi:10.1145/3524481.3527220

[29] Yuqing Wang, Mika Mäntylä, Serge Demeyer, Kristian Wiklund, Sigrid Eldh, and Tatu Kairi. 2020. Software Test Automation Maturity: A Survey of the State of the Practice. 27–38. doi:10.5220/0009766800270038

[30] 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. arXiv:2109.00859 [cs.CL] https://arxiv.org/abs/2109.00859

[31] 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* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. doi:10.1145/3377811.3380429

[32] Robert White and Jens Krinke. 2020. ReAssert: Deep Learning for Assert Generation. arXiv:2011.09784 [cs.SE] https://arxiv.org/abs/2011.09784

[33] Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. 2024. A Survey on Knowledge Distillation of Large Language Models. arXiv:2402.13116 [cs.CL] https://arxiv.org/abs/2402.13116

[34] 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.* (Oct. 2024). doi:10.1145/3699518 Just Accepted.

[35] Collet Yann and LZ4 Contributors. 2011. LZ4: Extremely Fast Compression Algorithm (GitHub repository). https://github.com/lz4/lz4. Accessed: 2025-05-26.

[36] 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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 214–224. doi:10.1145/2786805.2786858

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