Efficient Local Test Assertion Generation

Distilling CodeT5+ for Reduced Model Size and High Accuracy

Di Wu

Supervisors: Mitchell Olsthoorn, Annibale Panichella

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: Di Wu
Final project course: CSE3000 Research Project
Thesis committee: Mitchell Olsthoorn, Annibale Panichella, Petr Kellnhofer

ABSTRACT

Effective software testing relies on the quality and correctness of test assertions. Recent Large Language Models (LLMs), such as CodeT5+, have shown significant promise in automating assertion generation tasks; however, their substantial computational resource demands limit their practical deployment in common development environments like laptops or local IDEs. To address this challenge, this work explores knowledge distillation to derive smaller, more efficient student models from a larger, pre-trained CodeT5+ teacher. While knowledge distillation has been successfully applied to general code models, its specific application to creating lightweight, locally-deployable models for test assertion generation remains a recognized research gap. Using a dataset that includes assertion input-output pairs and teacher logits, we systematically investigate the impact of different distillation loss components—soft logits loss and hard target losses—on student performance. Our findings demonstrate the practical viability of this approach: a distilled 220M parameter student model can be nearly **3x faster** and consume over **40% less memory** than its 770M teacher, while retaining approximately **78%** of the original's assertion generation quality as measured by CodeBLEU. These results offer practical insights and a clear pathway for deploying efficient yet effective assertion-generation models suitable for local developer workflows.

Keywords: Knowledge Distillation, Assertion Generation, Large Language Models, Software Testing

1 INTRODUCTION

High-quality test assertions are fundamental to effective software testing, directly influencing fault detection and software correctness [15, 16]. However, manually crafting accurate and comprehensive assertions is often tedious and error-prone, motivating the need for automated assertion generation.

Recent advancements in Large Language Models (LLMs), particularly CodeT5+ [24] and other code-specific transformer models [3, 5, 12], have demonstrated significant promise in automating assertion generation tasks [22]. These models exhibit exceptional understanding of code semantics, enabling accurate and contextually relevant assertion generation [2, 4]. Despite their effectiveness, integrating these models into everyday development workflows remains challenging due to high computational demands, substantial memory usage, and slow inference speeds, especially in resource-constrained environments such as laptops or local IDEs [10].

Knowledge distillation offers a compelling solution by enabling the transfer of knowledge from powerful, computationally intensive *teacher* models into smaller, more efficient *student* models [9, 19, 26]. However, while knowledge distillation has been applied broadly, its specific application to the niche task of assertion generation—with the explicit goal of creating resource-efficient models for local development workflows—remains a gap in the literature that has received limited attention. In this work, we address this gap by utilizing a dataset generated by a pre-trained CodeT5+ teacher model, comprising input source code snippets, generated assertions, and associated logits (prediction probabilities). This dataset allows the student models to effectively learn the teacher's knowledge without the need for local deployment of the computationally demanding teacher model [7, 14].

We systematically distill knowledge into smaller student models, evaluating them across multiple performance metrics, including syntactic correctness, semantic similarity to teacher outputs (CodeBLEU [18]), exact match accuracy against ground-truth assertions, inference speed, and memory usage [21]. Preliminary findings suggest that distilled student models significantly enhance inference speed and reduce resource requirements while maintaining assertion generation performance close to that of the teacher [10, 19].

This paper makes the following contributions:

- Empirical evaluation of how varying distillation loss weights impact assertion generation accuracy, inference speed, and model efficiency.
- Empirical evaluation of how model size impacts inference speed, accuracy, and memory efficiency, offering insights into practical trade-offs for deployment.
- A comprehensive and publicly archived replication package that includes our entire toolchain—from distillation and evaluation scripts to the complete dataset and pre-computed teacher logits—to ensure full and transparent reproducibility of our findings.
- Practical recommendations for developers seeking to deploy effective, resource-efficient assertion-generation models in local development environments.

The remainder of this paper is structured as follows: Section 2 provides background information and related work on assertion generation and knowledge distillation. Section 3 describes our distillation methodology, and Section 4 outlines our experimental design. We present our empirical results in Section 5 and interpret these findings in Section 6. In Section 7, we discuss the threats to the validity of our study. Finally, Section 8 concludes the paper with a summary of our contributions and directions for future work, followed by our data availability statement.

2 BACKGROUND AND RELATED WORK

2.1 The Role of Assertions in Software Testing

Test assertions are conditional statements that validate the correctness of software behavior during execution. They typically appear within unit tests to check specific outcomes or states against expected values. Effective assertions are crucial for quickly detecting software faults, ensuring that code modifications do not introduce regressions, and maintaining software quality over time. Writing accurate and comprehensive test assertions is, however, time-consuming and prone to human error, motivating the exploration of automated methods [16].

2.2 Automated Assertion Generation Techniques

Several paradigms have been explored to automate the generation of test assertions, each with distinct strengths and limitations.

Search-Based Approaches. One prominent line of work involves Search-Based Software Testing (SBST), which uses metaheuristic algorithms (e.g., genetic algorithms) to search for meaningful assertions. Tools like EvoSuite [6] mutate source code and guide the search towards assertions that fail on mutants but pass on the original code. While effective at finding specific types of faults, this approach is often computationally expensive and tends to generate assertions that are syntactically simple, frequently failing to capture complex semantic properties or produce human-readable expressions.

Early Deep Learning Approaches. Before the dominance of large-scale transformers, researchers applied earlier deep neural network (DNN) architectures, such as Recurrent Neural Networks (RNNs) and LSTMs, to the problem. These models learned to predict assertion statements by treating code as a sequence [8]. While a significant step beyond pattern-based methods, these models were often limited by their ability to capture long-range dependencies and the deep semantic context of the code, resulting in assertions that were plausible but not always accurate or relevant.

Large Language Models (LLMs). The recent advent of LLMs specialized for code has marked a significant leap in performance on automated code understanding and generation tasks. Models like CodeT5+ [24], Codex [3], and AlphaCode [12] leverage the transformer architecture to build a deep understanding of code semantics. This enables them to generate highly accurate and contextually relevant assertions that closely mimic those written by human developers [22]. However, this power comes at a cost. The immense size and computational requirements of these state-of-the-art models make them impractical for integration into local, interactive development workflows on standard hardware like laptops, creating a critical gap between their potential and practical deployment.

2.3 Knowledge Distillation for Code Models

Knowledge distillation offers a promising solution to bridge this deployment gap. It is a technique used to transfer knowledge from a large, computationally expensive model (the *teacher*) into a smaller, more efficient model (the *student*) [9]. This process involves training the student to mimic the teacher's behavior, often by using its prediction probabilities (soft logits) as a rich supervisory signal alongside traditional ground-truth labels. This allows the smaller student model to achieve comparable performance to its teacher while significantly reducing inference time and memory usage.

Relevant literature indicates the efficacy of knowledge distillation in various domains, including software engineering. Hinton et al. [9] introduced the core concepts, while more recent surveys by Yang et al. [26] have highlighted its effectiveness for code-related tasks, underscoring the importance of carefully balancing different loss components. Despite these foundational works, there remains a clear gap in the literature addressing the unique challenges of distilling state-of-the-art code models like CodeT5+ specifically for assertion generation. To our knowledge, no prior work has systematically evaluated the trade-offs between distillation loss configurations and practical deployment metrics (speed, memory) for this task. Our research aims to fill this niche, providing empirical evidence to guide the creation of efficient, locally-deployable assertion generation tools.

3 METHODOLOGY

This section provides a comprehensive description of the knowledge distillation pipeline used for Java test assertion generation, clearly outlining the key steps and decisions involved. A high-level overview of the pipeline is visualized in Figure 1.

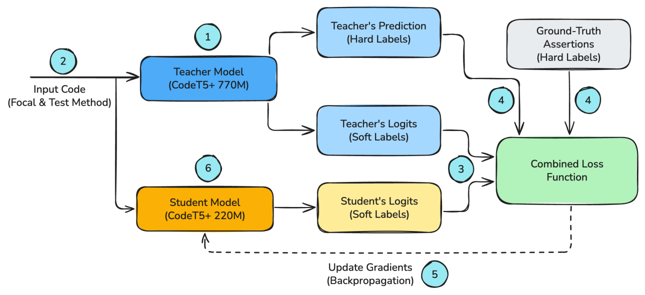

Figure 1: High-level overview of the knowledge distillation pipeline. The student learns from teacher logits and ground-truth labels. An additional AST-based syntactic penalty, not pictured, is also incorporated into the final loss function.

3.1 Teacher Model

The teacher model employed is a `Salesforce/codet5p-770m`[24] fine-tuned specifically for Java assertion generation(**① in Figure 1**). We fine-tuned this model on the **Methods2Test** dataset[23], which consists of 780,944 pairs of Java methods (focal methods) and corresponding JUnit test cases collected from 91,385 Java open-source GitHub repositories. This comprehensive dataset is specifically designed to support automated test generation tasks by providing well-curated method-test pairs.

The fine-tuned teacher model generates two types of outputs for each input method-test pair:

- **Predicted assertions**, which are the high-quality generated test assertions.
- **Compressed token-level logits**, which represent the model's internal confidence distributions across vocabulary tokens.

These outputs serve as the supervision signals during the student model's distillation.

3.2 Distillation Procedure

The knowledge distillation process involves the following steps:

Data Preparation. All input code and assertion sequences are first tokenized and padded to fixed lengths. During each training step, the compressed logits provided by the teacher are decompressed on-the-fly, ensuring the rich probabilistic information is available for calculating the loss(②).

Composite Loss Function. The student model is optimized using a composite loss function integrating multiple components:

- **Soft-label distillation (KL Divergence)**[9]: The student's logits are trained to match the decompressed teacher logits, using a dynamically decaying temperature to gradually sharpen predictions(③).
- **Hard-label supervision (Cross-entropy loss)**: Losses computed separately against teacher-generated assertions and ground-truth assertions help stabilize the training process(④).

- **AST-based penalty**: Predictions are parsed with `javalang`, and syntactic validity violations incur an additional penalty, encouraging the student to produce syntactically valid Java code[1, 27].

The relative weights of these loss components are experimentally tuned to achieve optimal performance.

The total weighted loss from these components is then used to update the student model's parameters through **backpropagation (⑤)**, completing the learning cycle.

Optimization and Regularization. To improve training stability and model generalization, we employ:

- **Dropout**[20]: Randomly deactivates units during training to mitigate overfitting.
- **Weight decay (L2 regularization)**[13]: Encourages smaller parameter magnitudes, controlling model complexity.
- **Validation-driven learning rate scheduling**: We use the `ReduceLROnPlateau` scheduler to dynamically reduce the learning rate when validation performance stagnates.
- **Mixed-precision training**: Accelerates computation and reduces memory usage by using PyTorch's Automatic Mixed Precision (AMP).

3.3 Student Models

The main student model investigated is `Salesforce/codet5p-220m`, a compact variant of CodeT5+ optimized for efficient local deployment(⑥). We also consider two smaller variants—`CodeT5-base` and `CodeT5-small`—to examine trade-offs in performance and efficiency across varying model sizes. All models are initialized with pre-trained Hugging Face weights.

4 STUDY DESIGN

This section describes the experimental configurations and evaluation procedures used to assess the performance and practical deployment capabilities of the distilled student models.

4.1 Research Questions

We specifically address the following research questions:

(1) How does student model size affect assertion generation quality relative to the teacher model?
(2) How do different weighting strategies for teacher prediction loss, ground truth loss, and soft logit loss affect assertion generation performance?
(3) What practical trade-offs exist between inference speed, memory usage, and assertion quality for local deployment?

4.2 Experimental Configurations

Training is executed on Google Colab using NVIDIA A100 GPUs, while evaluation is conducted on Apple Silicon (M1 Pro) hardware to assess deployment in realistic scenarios. Our experiments are designed around two primary independent variables: **student model size** (220M vs. Base vs. Small) and the **distillation loss weighting scheme**. To analyze the latter, we define several named configurations, including a 'Gold(Ground Truth)-focused', 'Teacher-focused', and 'Balanced' approach, which are systematically tested while all other training hyperparameters (e.g., learning rate, epoch count, dropout rate) are held constant.

4.3 Implementation Tools and Evaluation Protocol

Evaluation is performed using a custom evaluation script executed on M1 Pro hardware. The script computes the following comprehensive set of metrics to thoroughly evaluate model performance:

Precision, Recall, and F1-score. [17]: Standard metrics for assertion correctness.

Exact Match Accuracy. : Measures the proportion of assertions that match ground truth exactly.

CodeBLEU. [18]: Assesses both semantic and syntactic quality of generated assertions.

Token-level Accuracy. : Fine-grained token matching metric to evaluate detailed correctness.

AST Validity. [27]: Measures the syntactic correctness of generated Java assertions.

Sequence Similarity. [11]: Measures the average character-level similarity using the Levenshtein ratio.

Inference Speed and Memory Usage. : Practical metrics evaluating resource efficiency on consumer-grade hardware.

Results are systematically logged and visualized using `matplotlib` and `seaborn` to facilitate clear performance analysis across all models and configurations.

5 RESULTS

This section presents the empirical results of our knowledge distillation experiments. We first provide a high-level overview of the performance of the various student models compared to the teacher. We then analyze these results in detail, structuring our discussion around each of our three research questions to evaluate the impact of model size, loss configurations, and the practical trade-offs for local deployment.

5.1 Overall Performance Comparison

To establish a performance baseline, we evaluated the fine-tuned teacher model against the three distilled student models. All student models in this comparison were trained using a balanced distillation configuration (0.4 ground-truth loss, 0.3 teacher prediction loss, and 0.3 soft logits loss). The following tables summarize the results, grouped by metric category.

Table 1: Core performance metrics (F1, Precision, Recall, and Exact Match Accuracy).

Model	F1	Prec.	Recall	Accuracy
Teacher (770M)	0.559	0.571	0.547	0.577
Student (220M)	0.298	0.310	0.287	0.323
Student (Base)	0.297	0.309	0.286	0.324
Student (Small)	0.220	0.230	0.211	0.243

Table 2: Code-specific and similarity metrics.

Model	CodeBLEU	Token Acc.	Seq. Sim.
Teacher (770M)	0.583	0.665	0.879
Student (220M)	0.453	0.430	0.782
Student (Base)	0.451	0.422	0.776
Student (Small)	0.403	0.345	0.731

Table 3: Structural metrics (AST Validity).

Model	AST Validity (%)
Teacher (770M)	84.6
Student (220M)	87.0
Student (Base)	88.1
Student (Small)	89.5

5.2 RQ1: Impact of Model Size on Quality

As shown in Tables 1 and 2, student model size has a clear and direct impact on assertion generation quality. There is a consistent trend where larger models yield higher performance across nearly all quality metrics. The 770M teacher model sets the performance benchmark with a Precision of 0.571, an F1-score of 0.559, and a CodeBLEU score of 0.583. The largest student, `Student (220M)`, retains a substantial portion of this quality, achieving a Precision of 0.310 and a CodeBLEU score of 0.453. As the model size decreases to `Student (Base)` and `Student (Small)`, all correctness and similarity metrics decline accordingly.

An interesting counter-trend appears in AST validity (Table 3), where the smallest models produce a higher percentage of syntactically valid assertions. The `Student (Small)` model achieves the highest validity at 89.5%, surpassing even the teacher model (84.6%). This suggests that smaller models, having less capacity, may be less prone to generating overly complex or syntactically adventurous outputs, thereby adhering more reliably to basic language structures.

5.3 RQ2: Impact of Distillation Loss Weights

To answer our second research question, we conducted two sets of experiments on the `Student (220M)` model. First, we analyze the overall effectiveness of our distillation process by comparing several strategies against a non-distilled baseline. Second, we investigate the optimal balance between the teacher's hard and soft signals when no ground-truth data is used.

5.3.1 The Effectiveness of Distillation vs. a Non-Distilled Baseline. Our first experiment compares three distinct distillation configurations against the off-the-shelf, non-distilled `Student (220M)` baseline. The results are presented across two figures: Figure 2 for core correctness metrics, and Figure 3 for code-specific quality metrics.

The most striking result, visible in both figures, is the significant performance gap between the baseline model and all three distillation strategies. The baseline fails to generate any correct assertions, achieving an F1-score of 0.0, while all distillation configurations achieve F1-scores around 0.30. This confirms that general pre-training is insufficient for this specialized task and that the knowledge transfer process is essential for adapting the student model effectively. When comparing the three distillation configurations to each other, we observe remarkably similar performance. For example, the CodeBLEU scores for the Gold-focused (0.458), Balanced (0.453), and Teacher-focused (0.453) models are nearly identical. This suggests that when a strong soft-target signal from the teacher's logits is present (fixed at a weight of 0.3), the model is not highly sensitive to the specific source of the hard labels.

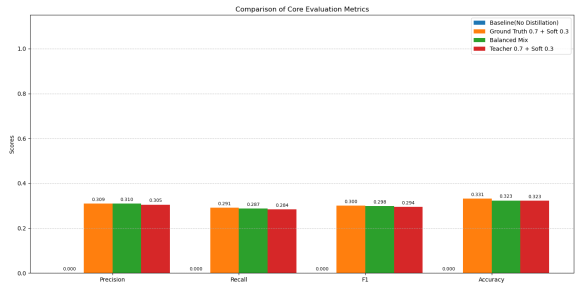

Figure 2: Comparison of core metrics for different distillation strategies against a non-distilled baseline.

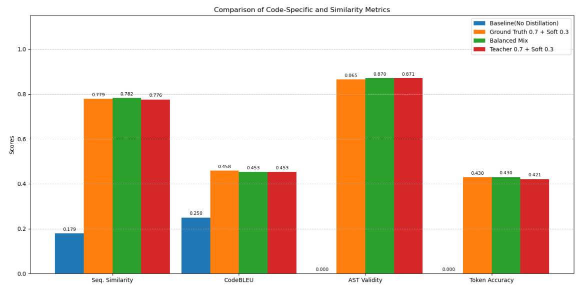

Figure 3: Comparison of code-specific metrics for different distillation strategies against a non-distilled baseline.

5.3.2 Balancing Hard vs. Soft Teacher Supervision. In the second experiment, we removed the ground-truth loss entirely to isolate the interplay between the two forms of teacher supervision: hard labels (from predictions) and soft labels (from logits). We evaluated four configurations by varying the weight between these two components, as shown in Figure 4 and Figure 5.

Contrary to the common expectation that a blend of signals would be optimal, our results reveal a different trend. The configuration relying exclusively on the teacher's hard predictions (**1.0 Teacher**) achieved the highest performance across most metrics, including an F1-score of 0.319 and a CodeBLEU score of 0.469. The second-best performer was the model trained purely on soft logits (**1.0 Soft**), with a CodeBLEU score of 0.455. Interestingly, the configurations that blended both signals produced slightly lower scores. This suggests that for this specific distillation task, a single, clear supervision signal—either imitating the teacher's final, fluent output (hard labels) or its pure underlying reasoning (soft labels)—was more effective than a combination that may have introduced conflicting optimization objectives for the student model.

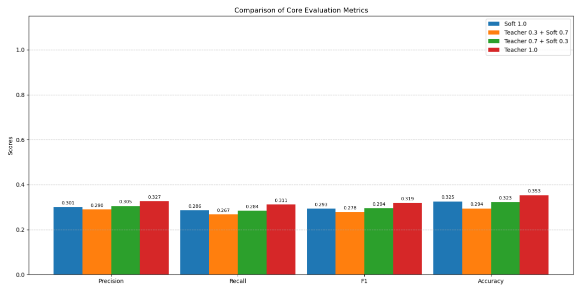

Figure 4: Effect on core metrics when varying the balance between the teacher's hard and soft losses.

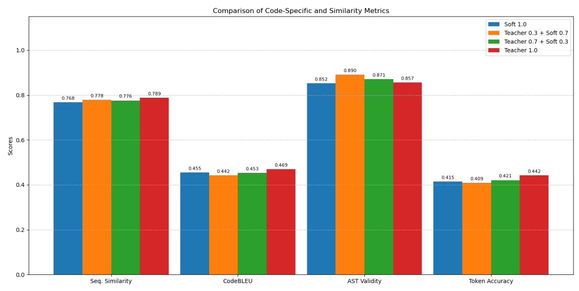

Figure 5: Effect on code-specific metrics when varying the balance between the teacher's hard and soft losses.

In summary, our findings for RQ2 are twofold. First and foremost, the distillation process itself is critical, as any tested configuration dramatically outperforms the non-distilled baseline. Second, when relying solely on supervision from the teacher, our results indicate that a direct imitation of the teacher's final predictions (hard-label loss) provides the most effective training signal, a noteworthy finding that contrasts with some distillation literature.

5.4 RQ3: Trade-offs for Local Deployment

This research question assesses the practical balance between the quality losses observed in RQ1 and the efficiency gains necessary for local deployment. The data from all our tables, particularly the quality metrics in Table 1 and the performance benchmarks in Table 4, illustrates these trade-offs clearly.

The largest student model, `Student (220M)`, provides a compelling case for distillation. It is approximately **2.9 times faster** per method than the 770M teacher and reduces peak memory consumption by **≈41%** (from 4.4GB to 2.6GB). A similar trend is observed in the time per assertion, where the student is also nearly **2.9 times faster**, indicating that the models generate a comparable number of assertions per method. This significant efficiency gain comes with a measurable quality trade-off, as its CodeBLEU score drops from 0.583 to 0.453 (retaining ≈78% of the teacher's score).

Table 4: Inference speed and memory usage on Apple M1 Pro.

Model	Time / Method (ms)	Time / Assertion (ms)	Memory Usage (MB)
Teacher (770M)	8216.2	5310.1	4418.89
Student (220M)	2795.9	1835.1	2604.33
Student (Base)	2965.9	1948.7	2658.30
Student (Small)	1225.4	813.4	2402.34

The smallest model, `Student (Small)`, offers the most dramatic performance improvement, being **6.7 times faster** than the teacher and consuming nearly **46% less memory**. However, this efficiency comes at a steep price. As shown in Tables 1 and 2, the `Student (Small)` model consistently scores the lowest on all correctness and similarity metrics. These findings highlight that the `Student (220M)` model represents a strong compromise, offering a substantial reduction in resource requirements while retaining a high level of assertion generation quality, making it a viable candidate for deployment on consumer-grade hardware.

6 DISCUSSION

Our empirical evaluation confirms that knowledge distillation is a highly effective strategy for creating computationally efficient yet powerful models for Java test assertion generation. The results demonstrate a clear and practical trade-off between model size, inference performance, and generation quality. In this section, we interpret these findings in the context of our research questions and discuss their broader implications for practitioners.

6.1 Interpretation of Findings

The experimental results provide clear answers to our initial research questions.

For RQ1 (Impact of Model Size), our findings align with the established principle that model quality scales with size. The larger 770M teacher model consistently outperformed all student models on correctness and similarity metrics like F1-score and CodeBLEU. This confirms that larger parameter counts allow for a more nuanced understanding of code semantics. However, we observed a noteworthy counter-trend in AST validity, where smaller models produced a higher percentage of syntactically correct code. A possible explanation is that models with less capacity are less prone to generating overly complex or creative outputs, forcing them to adhere more strictly to the fundamental grammatical patterns present in the training data.

For RQ2 (Impact of Loss Weights), our experiments reveal two key insights. First, the most significant performance gain comes from the distillation process itself. All distillation strategies dramatically outperformed the non-distilled baseline, proving that general pre-training is insufficient and that knowledge transfer is essential for specializing the model for this task. Second, when relying solely on the teacher for supervision, our results indicate that direct imitation of the teacher's final predictions (hard-label loss) provides the most effective training signal, a noteworthy finding that contrasts with some distillation literature.

For RQ3 (Practical Trade-offs), the results highlight a clear "sweet spot" for local deployment. The `Student (220M)` model emerges as the most compelling candidate. It achieves a nearly **3x speedup** and a **41% reduction in memory usage** compared to the teacher model. While this efficiency comes at the cost of reduced quality (retaining 78% of the teacher's CodeBLEU score), it represents a powerful and practical compromise, making it a viable candidate for deployment on consumer-grade hardware.

6.2 Implications for Practitioners

Our findings have direct implications for developers and teams looking to integrate AI-powered code generation into local development environments. The primary takeaway is that it is not necessary to rely on large, cloud-hosted models for effective assertion generation. A well-distilled medium-sized model, like our 220M student, can provide a significant productivity boost directly within an IDE without prohibitive resource requirements. This demonstrates that distillation is a viable pathway to creating developer tools that are both powerful and practical. Furthermore, the high syntactic validity of smaller models suggests they can be trusted to produce usable code snippets that require minimal correction.

7 THREATS TO VALIDITY

This section outlines the primary threats to the validity of our findings and the scope of our study.

A key threat to **external validity** is the study's focus on a single domain (Java assertion generation) and a single teacher architecture (CodeT5+). The generalizability of our findings regarding optimal distillation weights and performance trade-offs may be limited when applied to other programming languages, code generation tasks, or different model architectures.

Regarding **internal validity**, our study relies on a pre-generated dataset with compressed teacher logits. The truncation of logits to the top-4 probabilities constitutes an information loss that may have impacted the fidelity of the soft-target supervision signal compared to an experiment with the full, uncompressed distributions.

The **construct validity** of our evaluation is constrained by two main factors. First, since the training and validation sets were provided to us, there is a potential risk of semantic overlap between them, which we mitigated through automated duplicate checks. Second, the task of assertion generation is inherently ambiguous, as multiple correct assertions can exist for a given test. This challenges metrics like Exact Match, which we addressed by reporting a diverse suite of metrics, including CodeBLEU, to form a more holistic assessment.

Finally, we acknowledge the limitations in scope. Our investigation is focused on a specific set of student models and distillation weight configurations. While our results indicate clear trends, a broader hyperparameter sweep or the inclusion of different model architectures could yield different results, presenting important avenues for future research.

8 CONCLUSION AND FUTURE WORK

The increasing size and resource demands of state-of-the-art language models for code present a significant barrier to their practical use in local development environments. In this paper, we addressed this challenge by demonstrating that knowledge distillation is a highly effective technique for compressing a large CodeT5+ model for the specific task of Java test assertion generation.

Our key contribution is the empirical validation of a distillation pipeline that produces smaller, more efficient student models. We found that a distilled 220M parameter model can be nearly **3x faster** and consume over **40% less memory** than its 770M parameter teacher, while retaining a substantial portion ($\approx$78%) of its generative quality as measured by CodeBLEU. Our analysis of distillation loss components provides actionable insights, showing that while any distillation strategy dramatically outperforms a non-distilled baseline, a balanced approach including ground-truth labels offers robust performance. These findings provide a clear pathway for creating powerful and practical code generation tools that can run efficiently on developers' local machines.

For future work, several promising directions exist. More advanced distillation techniques, such as matching intermediate layer representations from the teacher, could be explored to further close the quality gap. Additionally, reinforcement learning strategies that use compiler feedback or test execution results as a reward signal could be integrated to optimize for functional correctness directly. Finally, extending and evaluating this distillation approach across a broader range of programming languages and code generation tasks would be a valuable next step in assessing its generalizability.

DATA AVAILABILITY

The complete replication package for this study is publicly and permanently archived on Zenodo [25]. The package contains all scripts used for model distillation, evaluation, and performance benchmarking. Furthermore, it includes the full dataset of method-test pairs, along with the pre-computed assertions and compressed logits from the teacher model, ensuring full reproducibility of our experimental results.

ACKNOWLEDGMENTS

The author acknowledges the use of ChatGPT (OpenAI's GPT-4.5 model) as a writing assistant for drafting and improving the clarity of the manuscript. The author thanks Professor Mitchell Olsthoorn and Professor Annibale Panichella for their invaluable guidance and support.

A EXPERIMENTAL DETAILS

This section provides supplementary details regarding the hyperparameters, software environment, and qualitative examples from our experiments to ensure full reproducibility.

A.1 Training Hyperparameters

Unless otherwise specified for a particular experiment, the distillation training for all student models was conducted using the constant hyperparameter values listed in Table 5.

A.2 Software Environment

All experiments were conducted using Python 3.11. The key libraries and their versions used in our training and evaluation pipelines are listed in Table 6.

Table 5: Constant hyperparameters used for all distillation training runs.

Hyperparameter	Value
Optimizer	AdamW
Learning Rate	3e-5
Max Epochs	10
Batch Size	4
Gradient Accumulation Steps	1
Weight Decay	0.01
Dropout Rate	0.1
LR Scheduler	ReduceLROnPlateau
LR Scheduler Patience	1 epoch
Initial Temperature (KL Loss)	4.0

Table 6: Key software library versions used in the experimental environment.

Library	Version
CodeBLEU	0.7.1
Javalang	0.13.0
NumPy	2.0.2
PyTorch	2.6.0+cu124
SacreBLEU	2.5.1
Transformers	4.52.4

A.3 Example Model Outputs

To provide a qualitative understanding of the models' performance, Table 7 shows a sample output from the teacher and the distilled student model for a given input focal method.

REFERENCES

[1] Wasi Ahmad, Neel Sundaresan, Michiel Bacchiani, et al. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Alexey Dosovitskiy, Egor Tarassov, , et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[4] Colin B Clement, David Drain, Neel Sundaresan, et al. 2020. PyMT5: multi-mode translation of natural language and Python code with transformers. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Lin Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[6] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium on the foundations of software engineering*. 415–418.

[7] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision* 129 (2021), 1789–1819.

[8] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 631–642.

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

[10] 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. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 4163–4174.

[11] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 10 (1966), 707–710.

[12] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Bellemare, Hieu Le, Johannes Huesing, Johnny Nham, et al. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097.

[13] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[14] Subhabrata Mukherjee, Arindam Mitra, Ahmed Khamissi, Kushal Agrawal, Hamed Hassan, and Ahmed H Awadallah. 2023. Orca: Progressive learning from complex explanation traces of GPT-4. *arXiv preprint arXiv:2306.02707* (2023).

[15] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[16] Mauro Pezzè and Michal Young. 2008. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons.

[17] David Martin Powers. 2020. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness and correlation. *arXiv preprint arXiv:2010.16061* (2020).

[18] Shuo Ren, Daya Guo, Shuai Lu, Lin Shou, Sijie Tang, Daxin Jiang, and Ming Zhou. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. In *Proceedings of the 28th International Conference on Computational Linguistics*.

[19] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *Proceedings of the 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing*.

[20] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[21] Emma Strubell, Ananya Ganesh, and Andrew McCallum McCallum. 2019. Energy and Policy Considerations for Modern Deep Learning Research. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 3645–3650.

[22] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Mathew White, and Denys Poshyvanyk. 2020. Unit Test Case Generation with Transformers and Focal Context. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 515–526.

[23] Michele Tufano, Cody Watson, Georgii Panteley, Mike Poremba, and Denys Poshyvanyk. 2022. Methods2Test: a dataset of focal methods mapped to test cases. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 15–26.

[24] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint arXiv:2305.07922* (2023).

[25] Di Wu. 2025. *SyrupDali/assertion-test-distillation: Paper Submission Release: Linked to Zenodo*. https://doi.org/10.5281/zenodo.15716118

[26] Cheng Yang, Jia Li, Yao Fu, Yifei Zhang, Letian Zhang, Yixin Li, Haidong Hu, and Ting Gui. 2024. A Survey on Knowledge Distillation for Large Language Models. *arXiv preprint arXiv:2403.03362* (2024).

[27] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1134–1145.

Table 7: Example of generated assertions from the Teacher and various distilled Student models.

Component	**Code**
Focal Method	public static void commentOrUncommentSQL(JTextArea scriptPanel) { commentOrUncomment(scriptPanel, SQL_COMMENT_CHARACTER); }
Ground Truth	`assertEquals(TEXT, scriptPanel.getText());`
Teacher (CodeT5+ 770M) Output	`assertEquals(TEXT, scriptPanel.getText());`
Student (CodeT5+ 220M) Output	`assertEquals(TEXT, scriptPanel.getText());`
Student (CodeT5 Base) Output	`Assert.assertEquals(TEXT.length, scriptPanel.getText().length);`
Student (CodeT5 Small) Output	`assertEquals(226, scriptPanel.getText().length());`