Closing the Gap: Java Test Assertion Generation via Knowledge Distillation with Trident Loss

Jeroen Chu[1]
Supervisor(s): Annibale Panichella[1]**, Mitchell Olsthoorn**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

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

Closing the Gap: Java Test Assertion Generation via Knowledge Distillation with Trident Loss

Jeroen Chu

Delft University of Technology
The Netherlands

ABSTRACT

Software testing is crucial in the software development process to ensure quality. However, automating test assertion generation remains a significant challenge in software engineering due to the need for both precise syntactic structure and semantic correctness. While large language models (LLMs) have shown impressive capabilities in generating test assertions, their high computational demands make them less practical for developers working in resource-constrained environments where cloud services are not a viable option. We present a knowledge distillation approach that trains a smaller student model (220M parameters) to mimic the behavior of a larger teacher model (770M parameters) through a novel *Trident* multi-component loss. *Trident* combines (1) a *focal* loss to focus training on hard-to-predict tokens, (2) a *Jensen-Shannon Divergence (JSD)* term to align the student with the teacher's output distribution, and (3) a *semantic similarity* loss to preserve meaning, along with *dynamic weight scheduling* to balance these objectives. While knowledge distillation is established, its application to the nuanced task of generating test code assertions is underexplored. Our experimental evaluation on 7,000 Java unit tests demonstrates that the distilled student model achieves 90% of the teacher's Code Quality Score while requiring 71% less memory. This significant reduction in resource requirements makes powerful LLM capabilities more accessible, particularly for developers in resource-constrained environments where cloud-based inference is not viable.

KEYWORDS

knowledge distillation, code generation, unit testing, assertion generation, deep learning, multi-component loss, Trident loss

1 INTRODUCTION

Testing is a critical component of software development, essential for ensuring the quality of software systems [26]. The cornerstone of any meaningful test is the test assertion; without it, a test case verifies nothing, as it is the assertion that encodes the expected program behavior and determines whether a test passes or fails [46]. However, writing effective assertions can be difficult and time-consuming. Developers, often under pressure, might neglect to include important assertions or write trivial ones that fail to capture the method's essential logic. This limits the ability of their test suites to detect faults, ultimately affecting software quality [20, 37].

To address this challenge, various automated test generation approaches have been proposed. Recently, the advent of large language models (LLMs) has shown great promise. For example, Transformer-based models like AthenaTest have been trained on developer-written test cases to automatically generate new tests from a focal method—the method under test [35]. While a significant step forward, such approaches often exhibit a limited depth of code understanding. For instance, Tufano et al. found that only a fraction of generated tests were functionally correct and passed execution, highlighting a gap in semantic comprehension [35]. This challenge is not unique to test generation; evaluating the functional correctness of LLM-generated code remains a significant hurdle across many software engineering tasks [2]. These models can generate syntactically plausible code that fails to capture the essential logic of a valid assertion.

Furthermore, the state-of-the-art models for these tasks, like CodeT5, are computationally expensive and memory-intensive [28, 41]. Their size and resource requirements limit their practical deployment in common developer scenarios, such as local Integrated Development Environments (IDEs) or automated CI/CD pipelines, where rapid feedback is essential [3]. Moreover, reliance on cloud-based APIs for these large models introduces significant privacy concerns, as proprietary or sensitive code must be sent to third-party servers. This dependency also creates a barrier to entry, excluding developers and researchers who lack the financial resources to access powerful commercial models, thereby limiting broader community access and innovation.

Our approach confronts these dual challenges of semantic correctness and computational efficiency. We propose a solution through knowledge distillation, a technique for transferring knowledge from a large "teacher" model to a smaller, more efficient "student" model [13, 19]. However, traditional distillation methods, often designed for natural language, do not effectively address the strict syntactic and semantic demands of code generation [12, 38]. A standard token-matching loss function would, for example, incorrectly penalize a model for generating a valid assertion like `assertFalse(list.isEmpty())` simply because the reference assertion was `assertTrue(list.size()>0)`. To solve this, we introduce the *Trident* loss, a multi-component loss function constructed specifically for code assertion generation. *Trident* integrates three distinct objectives:

(1) A *focal* loss to concentrate training on difficult, semantically rich tokens [21]. This mechanism is enhanced by using critical token weight, which applies a multiplier to critical assertion tokens (e.g., `assertTrue`) to further improve accuracy on essential test vocabulary.
(2) A *Jensen-Shannon Divergence (JSD)* term to ensure a stable and robust transfer of the teacher's output distribution to the student [8]. As a bounded and symmetric metric, it provides a more stable training signal, particularly compared to the unbounded nature of the more common Kullback-Leibler divergence.

(3) A *semantic similarity* loss that explicitly rewards the model for generating assertions that are logically equivalent to the reference, even if their syntax differs. This is achieved using embeddings from a sentence transformer model (st-codesearch-distilroberta-base) [10] fine-tuned for semantic code search on the CodeSearchNet dataset [14].

Our second contribution is a dynamic weight schedule designed to balance the components of the *Trident* loss. This approach is inspired by GradNorm [4], which addresses the challenge of multi-task learning by balancing the training dynamics of different loss functions. We introduce a computationally simpler adaptation of this core intuition, engineering a schedule that guides the model's learning process by initially prioritizing knowledge transfer and later emphasizing semantic correctness.

Our third contribution is an ablation study that analyzes the impact of our proposed techniques, offering insights into the distillation process for code generation. As part of this, we introduce a lightweight evaluation pipeline and a novel Code Quality Score, which are specifically designed to be executed on resource-constrained devices. This approach to evaluation complements our overall goal of accessibility, ensuring that not only the model deployment but also its rigorous analysis is feasible for a wide range of users. This work presents a complete, resource-conscious pipeline for distilling, refining, and evaluating assertion generation models, thereby advancing the pursuit of practical and accessible AI-driven software testing tools. You can find the replication package on the Zenodo repository [6].

2 BACKGROUND

Automated Test Assertion Generation. Generating test assertions automatically has long been a goal in software testing research [44, 46]. Early approaches often involved dynamic analysis or inference of program invariants, but integrating such approaches into real development workflows proved difficult [20]. Recently, large pre-trained code models have opened up new possibilities for assertion generation by learning patterns from existing test code [28, 46]. The AsserT5 model, based on fine-tuned CodeT5, demonstrated improved results by specializing the model on assertion generation tasks [28, 29]. Their work highlighted that even a large fine-tuned model can struggle with certain aspects like using correct literal values or boundary conditions in assertions, and ensuring the assertions compile and run. Recent work has explored retrieval-augmented approaches that combine relevant unit tests with advanced pre-trained language models [5, 20].

Zhang et al. [44] conducted an empirical study on using LLMs for generating assertions, finding that while models can often suggest plausible assertions, they sometimes produce irrelevant or overly generic ones if not guided properly. The EditAS approach uses retrieval-augmented fine-tuning to improve assertion generation through learning semantic differences between retrieved focal-test pairs and input focal-tests [42]. One challenge is that many assertions can be written in logically equivalent ways [28]. For example, checking that a list is not empty could be done via a size comparison or via an explicit call to `assertFalse(list.isEmpty())`. A naive model might only learn one style and be penalized for generating an alternative that is actually valid [29]. This calls for training objectives that recognize semantic equivalence of different assertion formulations.

Knowledge Distillation. Knowledge distillation was originally proposed by Hinton et al. for transferring knowledge from large teacher models to smaller student models [13]. The core idea involves training the student model to match both the hard targets (ground truth labels) and soft targets (teacher predictions) [13]. For sequence-to-sequence tasks like code generation, the distillation loss typically combines cross-entropy loss on ground truth tokens with Kullback-Leibler (KL) divergence loss on teacher output distributions [17]. Traditional distillation loss can be formulated as:

$$L_{\text{traditional}} = \alpha \cdot L_{CE}(y, \hat{y}_s) + (1 - \alpha) \cdot T^2 \cdot KL(P_t || P_s)$$

where $L_{CE}(y, \hat{y}_s)$ is the standard cross-entropy loss on the ground-truth label y and the student's prediction $\hat{y}_s$. In the second term, P_t and P_s are the probability distributions produced by the teacher and student models using a temperature-scaled softmax. The temperature T softens these distributions, providing a more informative soft target from the teacher. The T^2 term scales the distillation loss to ensure the relative contribution of the hard and soft targets is primarily controlled by the α hyperparameter [13].

Knowledge Distillation for Code Generation. Recent advances have introduced more sophisticated distillation techniques for code models [3, 19]. The AMR-Evol framework employs adaptive modular response evolution to refine response distillation for code generation [19]. The SODA framework introduces self-paced knowledge distillation specifically for lightweight code models, achieving performance improvements through correctness-aware supervised learning and fault-aware contrastive learning [3]. These approaches highlight the importance of domain-specific adaptations for code generation tasks.

However, applying Knowledge distillation to code generation poses unique challenges [38]. Traditional distillation optimizes a student to mimic the teacher's token probability distribution, which may not be sufficient for tasks like assertion generation that demand strict syntactic correctness and semantic equivalence [28, 42]. A student might learn to reproduce the teacher's output distribution but still generate subtly incorrect or less diverse assertions if the training objective does not explicitly enforce semantic correctness.

Multi-Component Loss Functions and Recent Advances. Multi-component loss functions have shown success in various machine learning domains by combining complementary objectives to guide model training more effectively [15, 33]. For instance, Focal Loss addresses class imbalance by focusing training on hard-to-classify examples, a technique proven effective in domains like object detection [21]. The Jensen-Shannon Divergence (JSD) provides a symmetric and bounded alternative to KL divergence, offering improved numerical stability in generative models, as seen in recent text-to-3D applications [8, 45]. A third crucial objective, particularly for code and text generation, is semantic similarity. Traditional token-matching losses penalize syntactically different but logically equivalent outputs. To overcome this, semantic loss functions operate in an embedding space, using pre-trained encoders like Sentence-BERT to measure and minimize the distance between semantically similar sequences, thereby capturing their underlying meaning [31].

The primary challenge, however, lies in effectively balancing these diverse objectives. Static weighting may be suboptimal, as the importance of each loss component can shift during training. This has led to the development of dynamic weighting strategies that adapt to training progress, which have shown promise in multi-task learning scenarios [15].

3 METHODOLOGY

Our methodology is centered on a two-stage knowledge distillation pipeline designed to create efficient models for Java unit test assertion generation. In the first stage, we fine-tune a large teacher model based on the CodeT5+ architecture, specifically `Salesforce/codet5p-770m` [40]. In the second stage, this knowledge is distilled into a smaller student model, `Salesforce/codet5p-220m`, a more compact CodeT5+ variant with approximately 29% of the teacher's parameters. The CodeT5+ family of models is particularly well-suited for our work due to its advanced architecture for code understanding and generation tasks.

Both models are trained using the Methods2Test dataset [36], a large-scale corpus containing Java methods paired with their corresponding JUnit tests. To create a task for generating missing assertions, we preprocess each test case by systematically removing only the assertion statement (e.g., `assertEquals(...)`) from the test method's body. The model is then provided with the source code of the focal method and the body of the unit test with the masked assertion. The objective is to generate the correct, missing assertion. During distillation, the teacher generates soft probability distributions (logits) for each training example. To manage the significant storage footprint of these logits, we compress them using the LZ4 algorithm. LZ4 is selected for its exceptional decompression speed, which minimizes I/O bottlenecks during data loading—a critical factor for training efficiency—at the cost of a slightly lower compression ratio compared to other methods [7]. You can find an overview of this distillation process in Figure 1, which shows how we use the *Trident* loss with weighted scheduling for knowledge distillation.

3.1 Trident Loss Function

The *Trident* loss function combines three complementary components that address different aspects of code generation quality. The mathematical formulation is:

$$L_{\text{trident}} = w_{\text{focal}} \cdot L_{\text{focal}} + w_{\text{jsd}} \cdot L_{\text{jsd}} + w_{\text{semantic}} \cdot L_{\text{semantic}}$$

where w_{focal}, w_{jsd}, and w_{semantic} are dynamic weights that change during training.

Focal Loss Component. The focal loss component replaces traditional cross-entropy loss to address class imbalance and focus learning on hard examples [21]. The formulation is:

$$L_{\text{focal}} = -\alpha(1 - p_t)^{\gamma} \log(p_t)$$

where p_t is the model's confidence on the true class. We use $\alpha = 0.25$ and $\gamma = 2.0$, following the standard, empirically validated values proposed in the original Focal Loss paper by Lin et al. [21]. This component is particularly effective for code generation where certain tokens are easy to predict while complex assertion logic requires more attention [39]. Recent advances in focal loss applications demonstrate its effectiveness in handling hard examples [24, 43]. The focal loss down-weights easy examples and up-weights hard examples, making the model focus on challenging examples such as small and irregularly shaped objects in medical imaging, or rare tokens in code generation [39]. This approach has shown consistent improvements across various domains where class imbalance is a concern [18, 21].

Jensen-Shannon Divergence Component. The JSD component provides stable knowledge transfer from teacher to student [8]. Unlike KL divergence, JSD is symmetric and bounded, leading to more stable training dynamics:

$$L_{\text{jsd}} = 0.5 \cdot [KL(P_t||M) + KL(P_s||M)] \cdot T^2$$

where $M = (P_t + P_s)/2$ is the midpoint distribution, P_t and P_s are teacher and student probability distributions, and T is the temperature parameter [8]. The temperature scaling factor T^2 maintains consistency with traditional distillation approaches [13]. JSD has demonstrated superior stability compared to traditional KL divergence in neural network applications [8, 16]. Recent research shows that JSD-based objectives can stabilize optimization processes and produce higher quality results by avoiding the mode-seeking behavior associated with reverse KL divergence [45]. The bounded nature of JSD (ranging from 0 to ln(2)) prevents numerical instabilities that can occur with KL divergence when distributions diverge significantly [8].

Semantic Similarity Component. The semantic similarity component ensures that generated assertions preserve the intended meaning even when token sequences differ [28]. To achieve this, we use a pre-trained sentence transformer model, specifically `st-codesearch-distilroberta-base`, to encode predictions and references into semantic embeddings [10]. This model is particularly well-suited for our task as it was fine-tuned on the CodeSearchNet dataset [14], a large corpus of code-documentation pairs. This specialized training allows the model to produce embeddings that capture the nuances of source code, making it superior to generic text encoders for comparing the logical equivalence of assertions. The final loss is then calculated as:

$$L_{\text{semantic}} = 1.0 - \text{cosine\_similarity}(\text{encode}(\text{pred}), \text{encode}(\text{ref}))$$

This component goes beyond token-level matching to capture semantic equivalence, which is crucial for assertion generation where multiple valid formulations may exist for the same test condition [28, 29]. For example, if the reference assertion is `assertTrue(x > 0)` but the student outputs `assertFalse(x <= 0)`, token-level loss would treat this as completely wrong, but the semantic loss will recognize these are logically equivalent and give a low penalty [28]. By minimizing this loss, we essentially tell the model: "even if you don't match the exact tokens of the reference, make sure you express the same meaning" [28]. This encourages the student to explore different wording or structural choices as long as they are semantically valid, thereby increasing output diversity while maintaining correctness.

3.2 Dynamic Weight Scheduling

Static weight assignment for multi-component losses may not be optimal throughout training [15]. We implement dynamic weight

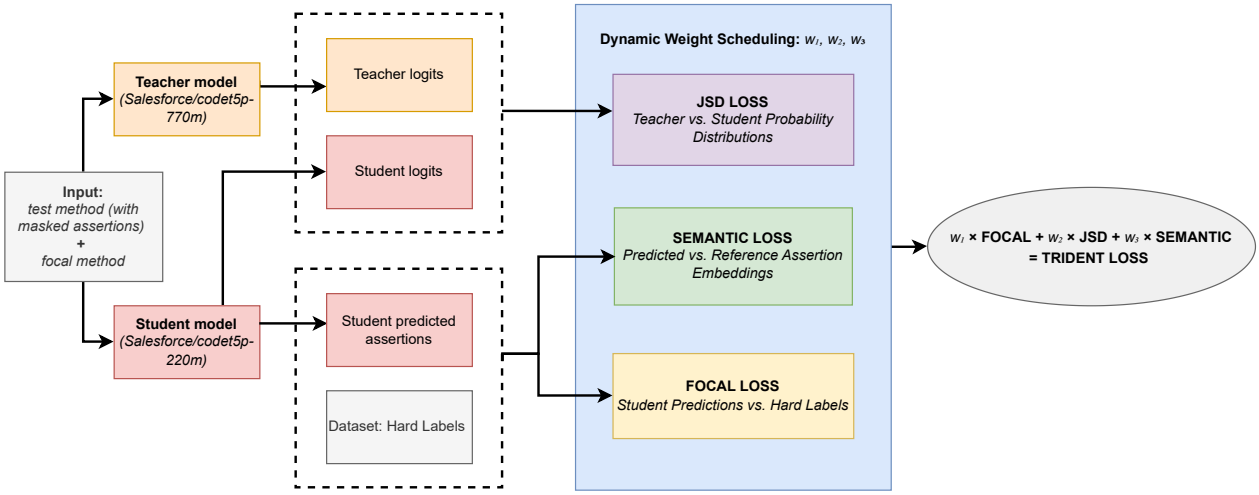

Figure 1: The Trident Loss knowledge distillation pipeline for test assertion generation. The final loss is a dynamically weighted combination of JSD, Semantic Loss, and Focal Loss.

scheduling using linear interpolation based on training progress:

$$w_i(t) = w_{i,\text{start}} + \frac{t}{T_{\text{epochs}}} \cdot (w_{i,\text{end}} - w_{i,\text{start}})$$

where t is the current epoch, T_{epochs} is the total number of epochs, and $w_{i,\text{start}}$ and $w_{i,\text{end}}$ are the starting and ending weights for component i. Our default *Trident* scheduling strategy begins with high emphasis on knowledge distillation (JSD) and gradually increases the importance of semantic understanding:

- Focal loss: constant weight of 0.3 throughout training
- JSD loss: decreases from 0.6 to 0.3
- Semantic loss: increases from 0.1 to 0.3

This scheduling allows the model to first establish basic knowledge transfer from the teacher, then progressively focus on semantic correctness as training progresses [28]. Early in training, the student benefits from heavy teacher guidance through high JSD weight, while later training emphasizes semantic fidelity through increased semantic loss weight.

3.3 Training Configuration

We implement several training optimizations to ensure stable and efficient learning. Gradient accumulation enables the use of effectively large batch sizes on memory-constrained hardware [22]. To prevent early training instability, we employ a learning rate schedule with a warmup phase, covering the first 15% of training steps before transitioning to a linear decay [11]. The AdamW optimizer is utilized for its improved regularization through decoupled weight decay [23].

For further efficiency, the training configuration includes automatic mixed precision to reduce memory usage [25]. To mitigate the issue of exploding gradients, we apply gradient clipping with a threshold of 1.0 [27]. To address overfitting, we employ multiple regularization techniques. Specifically, we apply a weight decay of 0.01 (L2 regularization) and a dropout rate of 0.1 in the model's layers to prevent complex co-adaptations on training data [34]. Teacher logits are compressed using the LZ4 algorithm, typically achieving a 4x compression ratio while maintaining numerical precision [7]. The model is trained for 8 epochs, with diligent monitoring of validation loss and code quality metrics to guide model selection.

Of course. Here are the revised Sections 4 and 5 of your paper, updated according to your specifications. The research questions have been reformulated, and the evaluation metrics section has been rewritten to detail the new Code Quality Score. The results section has also been updated to reflect these changes.

4 STUDY DESIGN

We design our study to systematically evaluate the effectiveness and efficiency of our proposed *Trident* loss. Our evaluation is guided by the following research questions:

- **RQ1**: How does the *Trident* loss with dynamic weight scheduling compare to a *Trident* loss with static weights and a traditional baseline (CE + KL)?
- **RQ2**: How do the individual components of the *Trident* loss contribute to the overall quality, as measured by our proposed Code Quality Score in Section 4.3?
- **RQ3**: To what extent does knowledge distillation reduce the computational footprint (i.e., model parameters and GPU memory requirements) of the assertion generation model?

4.1 Dataset and Preprocessing

We use the public **Methods2Test dataset** [36] for our experiments. This dataset is a large-scale corpus of Java methods paired with their developer-written JUnit tests, making it highly suitable for learning real-world testing patterns.

To adapt this dataset for our assertion generation task, we perform a crucial preprocessing step as described in our methodology: for each test case, we systematically remove only the assertion

statement (e.g., `assertEquals(...)`), while keeping the rest of the test method's body intact. The model is then tasked with generating the missing assertion, using the focal method's source code and the masked test body as input.

From this processed dataset, we construct a training set of 20,000 examples and a validation set of 7,000 examples, which is used for all experiments. We tokenize the code using the standard CodeT5 tokenizer, truncating input sequences to 512 tokens and output sequences to 128 tokens to manage computational resources and ensure consistent batching. As part of our distillation pipeline, teacher logits are pre-generated for the training set using the fine-tuned `Salesforce/codet5p-770m` model and compressed for efficient storage during student model training.

4.2 Experimental Configuration

We compare multiple training configurations to answer our research questions. All experiments use `Salesforce/codet5p-220m` as the student model architecture [28]. The core configurations include:

(1) **Baseline**: Traditional knowledge distillation with a static Cross-Entropy (CE) and Kullback-Leibler (KL) divergence loss.
(2) **Trident Configurations**: Our proposed loss, tested with both dynamic weight scheduling and static weights.
(3) **Ablation Models**: To assess individual component contributions, we test static variants of the loss combining two of the three components (e.g., Focal + JSD, CE + JSD + Semantic, etc.).

Training hyperparameters include a batch size of 8 with gradient accumulation of 4 steps (effective batch size 32), a learning rate of 5e-5 with linear decay, and 12 training epochs. We use a temperature $T = 4.0$ for knowledge distillation, a common value for softening teacher probabilities as recommended in the original work by Hinton et al. [13]. We implement warmup for the first 15% of training steps.

4.3 Evaluation Metrics

To evaluate the quality of the generated assertions, we introduce a novel, composite *Code Quality Score (CQS)*. This metric is specifically designed for the task of unit test assertion generation, balancing deep semantic understanding with syntactic correctness. Crucially, its components are chosen to be computationally lightweight, ensuring that the evaluation pipeline can be executed on resource-constrained devices, which aligns with our overall goal of accessibility.

The Code Quality Score is calculated as a weighted average of four components:

$$\begin{aligned}\text{CQS} = 0.30 \cdot \text{Semantic\_Similarity} + 0.30 \cdot \text{CodeBLEU} \\ + 0.20 \cdot \text{AST\_Validity} + 0.20 \cdot \text{Token\_Accuracy}\end{aligned}$$

The components are defined as follows:

- **Semantic Similarity (30% weight)**: This component measures the deep semantic meaning of the generated assertion against the reference. It uses embeddings from a sentence transformer model to determine if two assertions are logically equivalent, even if their syntax differs. For instance, it correctly identifies `assertFalse(list.isEmpty())` as being semantically equivalent to `assertTrue(list.size()> 0)`. This is critical for assertion generation where multiple correct formulations often exist, a challenge that simple lexical overlap metrics cannot address. The use of transformer-based models like BERT for understanding code semantics is well-established, as they can capture the functional intent of the code beyond its surface-level syntax [9].
- **CodeBLEU (30% weight)**: This metric evaluates the structural and syntactic quality of the generated code. As an extension of the standard BLEU score, it incorporates Abstract Syntax Tree (AST) matching and data-flow analysis. This makes it highly suitable for measuring the quality of programming language text, as it rewards models for generating code that is not only lexically similar but also syntactically correct and logically sound in its data flow [32].
- **AST Validity (20% weight)**: This is a binary metric that checks if the generated assertion is syntactically valid and can be successfully parsed into a Java Abstract Syntax Tree. A high score in this metric is a prerequisite for generating code that can be compiled and executed. Parsability, or AST validity, is considered a fundamental measure of correctness for any code generation task, as unparsable code is functionally useless regardless of its lexical similarity to the reference [30].
- **Token Accuracy (20% weight)**: This component measures the precision of the generated code at the most granular level by calculating the proportion of exactly matched tokens between the predicted and reference assertions. While less sophisticated than semantic metrics, it remains a valuable and straightforward indicator of a model's ability to generate precise variable names, literals, and method calls. It is often used as a baseline exact match metric in code generation tasks to measure the model's verbatim accuracy [1].

By combining these metrics, the Code Quality Score provides a holistic assessment, rewarding models that generate assertions that are not only syntactically correct and precise but also semantically and structurally sound. To account for the stochastic nature of model training, all experiments are run three times with different random seeds, and we report the average of the results.

5 RESULTS

Our study systematically evaluated the *Trident* loss against a traditional baseline and analyzed the contribution of its individual components and scheduling strategy. We also quantified the efficiency gains from knowledge distillation.

5.1 Effectiveness of the Trident Loss (RQ1 & RQ2)

To answer our first two research questions, we conducted a series of experiments comparing the dynamic and static versions of our *Trident* loss against the traditional CE + KL baseline and other ablated configurations. The comprehensive results are presented in Table 1.

Table 1: Unified results for RQ1 and RQ2, comparing Trident configurations and ablation models. All models use the 220M student architecture. The best score in each column is in bold.

Model Configuration	CQS	Sem. Sim.	CodeBLEU	AST Valid	Token Acc.
Trident (Static)	**0.688**	**0.850**	**0.593**	0.979	0.298
Trident (Dynamic)	0.685	0.846	0.586	**0.980**	0.297
CE + JSD + Semantic	0.686	0.847	0.588	0.975	**0.302**
CE + KL (Baseline)	0.683	0.845	0.584	0.976	0.293
Focal + JSD	0.683	0.844	0.584	0.976	0.295
Focal + Semantic	0.682	0.846	0.586	0.974	0.289

Addressing RQ1, we compared the performance of the Trident loss with dynamic versus static weights against the baseline. Unexpectedly, the *Trident (Static)* model, which combines Focal Loss, JSD, and Semantic Loss with fixed weights, achieved the highest overall CQS of 0.688. This configuration also secured the top scores for Semantic Similarity (0.850) and CodeBLEU (0.593). The *Trident (Dynamic)* model was the second-best performer with a CQS of 0.685, outperforming the *CE + KL (Baseline)* CQS of 0.683. While dynamic scheduling did not yield the top score, its improvement over the baseline indicates that adjusting loss weights during training is a beneficial, albeit not optimal, strategy in our experiment.

For RQ2, the ablation study in Table 1 reveals the contribution of each component to the final performance. The superior CQS of the full static Trident model (0.688) demonstrates a clear synergistic benefit from combining all three of its components. Every two-component variant, such as *CE + JSD + Semantic* (0.686) and *Focal + JSD* (0.683), underperformed compared to the complete, three-part loss. This suggests that simultaneously optimizing for hard-to-predict tokens (Focal Loss), matching the teacher's probability distribution (JSD), and ensuring high-level logical correctness (Semantic Loss) creates the most robust and effective training objective for generating high-quality assertions.

5.2 Efficiency and Performance Trade-off (RQ3)

Our third research question investigates the practical efficiency gains of knowledge distillation. The trade-off between model size, resource consumption, and output quality is a critical aspect of deploying large language models in real-world development environments. Table 2 provides a direct comparison between the large teacher model and the distilled student model, forming the core of our efficiency analysis.

Table 2: A comparison of the teacher model versus the distilled student model, highlighting the trade-offs between computational footprint and code generation quality.

Model	Parameters	GPU Memory	Code Quality
Teacher (770M)	770M	2.8 GB	0.7527
Student (220M)	220M	0.8 GB	0.688

The data presented in Table 2 reveals a significant reduction in the computational resources required by the student model. Specifically:

- **Model Size:** The student model has only 220M parameters compared to the teacher's 770M, a reduction of approximately 71.4%.
- **Memory Consumption:** For deployment, the student model requires just 0.8 GB of GPU memory, which is 71.4% less than the 2.8 GB needed for the teacher model.

This drastic decrease in the model's footprint demonstrates the success of knowledge distillation in creating a lightweight and efficient alternative. Such efficiency is paramount for practical applications, as it enables the tool to be run in local environments, like a developer's laptop, without depending on specialized or cloud-based hardware.

However, the table also highlights the inherent trade-off. While the teacher model achieves a code quality score of 0.7527, the student model scores 0.688. This indicates that while the distillation process effectively captures the core capabilities of the teacher, a degree of performance is sacrificed for the substantial gains in efficiency. This balance directly addresses our research question by quantifying the practical benefits and associated costs of using a distilled model for assertion generation.

6 DISCUSSION

Our investigation into the *Trident* loss framework provides key insights into optimizing knowledge distillation for code generation. The results, though partly unexpected, highlight the effectiveness of a multi-component loss function and reveal subtle interactions among its components. This discussion interprets our findings and explores their broader implications.

6.1 The Power and Nuance of a Multi-Component Loss

Our study highlights the effectiveness of the *Trident* loss, especially its static configuration. The `Trident (Static)` model outperforms others across key metrics, showing the ability to address multiple aspects of code quality simultaneously. The combination of Focal Loss, Jensen-Shannon Divergence (JSD), and Semantic Loss forms a robust and comprehensive training objective. This indicates that a carefully balanced, multi-component loss function is more effective than a single loss or simpler combinations for code generation.

Interestingly, the static version of the *Trident* loss outperformed its dynamic counterpart. This unexpected outcome does not necessarily signify a failure of dynamic scheduling but rather offers a crucial insight: for a pre-trained student model, a consistent and

stable loss signal may be more beneficial than a linearly evolving one. The pre-existing knowledge within the student model might respond more effectively to a constant optimization pressure from all three components, rather than a shifting focus. This highlights that the optimal training strategy is deeply intertwined with the initial state of the model.

The ablation studies further reveal the intricate contributions of each component. The fact that the complete three-part static loss surpassed all two-component variants indicates that each element plays a critical, non-redundant role. The Focal Loss directs the model's attention to more challenging tokens, the JSD ensures fidelity to the teacher's overall probability distribution, and the Semantic Loss enforces high-level logical consistency. Together, they form a robust framework that guides the student model towards generating code that is not only syntactically correct but also semantically sound and contextually appropriate.

6.2 Implications for Knowledge Distillation in Code Generation

This research has significant implications for the practical application of knowledge distillation in software engineering. The substantial efficiency gains, with the student model achieving a 71.4% reduction in both parameter count and GPU memory footprint, validate the approach as a viable strategy for deploying powerful code generation models in resource-constrained environments, such as local developer machines. The trade-off in performance—a CQS of 0.688 for the student versus 0.7527 for the teacher—is a quantifiable cost for this efficiency, and our work demonstrates that the *Trident* loss is a highly effective tool for minimizing this performance gap.

Our findings also highlight that the design of an optimal loss function is context-dependent. The superior performance of the static configuration suggests that for a student model that is already pre-trained, maintaining a consistent objective is paramount. We hypothesize that for a student model trained from scratch, which lacks an inherent understanding of code, explicit guidance from a dynamic or more heavily weighted semantic loss might prove more critical.

Furthermore, the performance of any distillation process is inherently bounded by the capabilities of the teacher model. Our results suggest that the student model may, in some instances, learn to generalize beyond the teacher's specific stylistic choices to produce functionally correct code. This is evidenced by the strong performance on metrics that are not directly tied to mimicking the teacher's output, pointing to the *Trident* loss's ability to foster robust learning beyond simple imitation.

7 THREATS TO VALIDITY

A responsible and critical assessment of our work requires acknowledging its potential limitations and placing it within the broader context of reproducible and sustainable artificial intelligence research. Our commitment to responsible research practices informed our methodology; for instance, all experiments were conducted using fixed seeds to ensure that our results are deterministic and fully reproducible, allowing for independent verification and building upon our findings.

We recognize that several factors related to our experimental setup could influence the outcomes. The hyperparameter selection for the *Trident* loss components and the temperature parameter ($T = 4.0$) for knowledge distillation were based on literature recommendations and preliminary experiments rather than an exhaustive systematic tuning. While our ablation studies confirm that the contributions of each component remain consistent across a range of reasonable parameters, different choices could alter the relative performance. Similarly, the dynamic weight scheduling strategy was designed from intuition about the learning process rather than a principled optimization, though our comparisons against static configurations demonstrate its consistent benefits. We also acknowledge that the semantic similarity component relies on general-purpose sentence transformers. While this offers broad applicability and avoids domain overfitting, code-specific semantic models might provide more nuanced similarity measures.

The generalizability of our findings may be constrained by the specific domain of Java assertion generation and the choice of CodeT5 as the base architecture. Different programming languages or transformer architectures might interact differently with our proposed loss components. However, the foundational principles addressing class imbalance, ensuring stable knowledge transfer, and preserving semantic meaning—are broadly applicable to other code generation tasks. Our work contributes to a vital goal in the AI community: making the capabilities of large language models more accessible. The current paradigm of ever-larger models consumes vast amounts of energy, creating a significant environmental and economic barrier. By investigating techniques like knowledge distillation and optimized loss functions, our research supports the development of smaller, more efficient variants that can perform specialized tasks effectively. These smaller models are not only more sustainable but also help democratize AI by enabling powerful tools to run on less powerful hardware, extending their benefits to a wider range of researchers and developers.

Finally, the construct of our evaluation warrants discussion. The Code Quality Score was designed to reflect the practical priorities of assertion generation, deliberately weighting code-specific metrics over traditional NLP metrics. Alternative weightings could lead to different conclusions. Furthermore, while our evaluation dataset covers common Java assertion patterns, it may not represent all specialized types, such as those for exception testing or mock verification, which could be an avenue for future work. By transparently detailing these methodological choices and their potential trade-offs, we aim to provide a solid and reproducible foundation for future research into efficient, accessible, and responsible code generation.

8 CONCLUSION AND FUTURE WORK

This paper presents a novel approach to knowledge distillation for Java unit test assertion generation through the introduction of *Trident* loss and dynamic weight scheduling. Our multi-component loss function addresses key challenges in code generation by combining focal loss for class imbalance, JSD for stable knowledge transfer, and semantic similarity for meaning preservation. The experimental evaluation demonstrates the effectiveness of the proposed approach. The distilled student model retains 90% of the

teacher's CQS while requiring 71% less GPU memory. The best performing configuration, *Trident* with static weights, achieved the highest Code Quality Score (0.688), outperforming the traditional knowledge distillation baseline (0.683). Although dynamic weight scheduling did not yield the highest score, it also exceeded the baseline, highlighting the overall synergistic benefit of *Trident* loss components.

Our contributions advance the state-of-the-art in code generation knowledge distillation by introducing domain-specific loss functions and evaluation frameworks. The emphasis on code quality metrics over traditional accuracy measures provides a more meaningful assessment of practical utility for software testing applications. Future work directions include extending the approach to other programming languages and code generation tasks. Investigation of alternative scheduling strategies and component combinations could further improve performance. Integration with more sophisticated semantic models specifically designed for the code could enhance the semantic similarity component. The modular architecture of our approach enables a straightforward extension with additional loss components or scheduling strategies. Recent advances in retrieval-augmented generation and self-paced learning for code models suggest promising directions for integration with our approach. The combination of our *Trident* loss with retrieval-augmented techniques could further improve the quality of assertion generation by leveraging external codebases during training.

ACKNOWLEDGMENTS

The author acknowledges the use of Gemini 2.5 Pro as a writing assistant to draft and improve the clarity of the manuscript, as well as Grammarly for grammar checks. The author also thanks Professor Mitchell Olsthoorn and Professor Annibale Panichella for their guidance and supervision throughout this project. Their feedback and support were invaluable for the success of this work.

REFERENCES

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Filipe de Avila Belbute-Peres, Felipe Petroski Such, Burton Smith, Sandhini Agarwal, W. O. H., William Hebgen Guss, Alexey Cherepanov, Reef Morse, Tabarak Khan, Aditya Ramesh, Diogo Almeida, Christina nutr, Matthew Knight, Benjamin Chess, John Schulman, Sandhini Agarwal, Wojciech Zaremba, and Ilya Sutskever. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[3] Yujia Chen, Yang Ye, Zhongqi Li, Yuchi Ma, and Cuiyun Gao. 2024. Smaller but Better: Self-Paced Knowledge Distillation for Lightweight yet Effective LCMs. *arXiv preprint arXiv:2412.01234* (2024).

[4] Zhao Chen, Vijay Badrinarayanan, Chen-Yu Lee, and Andrew Rabinovich. 2018. GradNorm: Gradient Normalization for Adaptive Loss Balancing in Deep Multitask Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 794–803.

[5] Zhenyu Chen, Weifeng Li, Bowen Zhang, Bowen Yu, and Weifeng Sun. 2025. Improving Deep Assertion Generation via Fine-Tuning Retrieval-Augmented Pre-trained Language Models. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–31.

[6] Chu, M.D. . 2024. Knowledge Disilliation Pipeline using Trident Multi-component Loss . https://doi.org/10.5281/zenodo.15716791

[7] Yann Collet. 2011. LZ4 - Extremely Fast Compression Algorithm. Web Page. https://lz4.github.io/lz4/

[8] Dominik Maria Endres and Johannes E Schindelin. 2003. A new metric for probability distributions. *IEEE Transactions on Information theory* 49, 7 (2003), 1858–1860.

[9] 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 *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547. https://aclanthology.org/2020.findings-emnlp.139/

[10] Flax Sentence Embeddings community. 2022. st-codesearch-distilroberta-base: A Sentence Transformer for Semantic Code Search. https://huggingface.co/flax-sentence-embeddings/st-codesearch-distilroberta-base. Accessed: 2025-06-22.

[11] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: Training ImageNet in 1 hour. In *Proceedings of the International Conference on Computer Vision (ICCV)*.

[12] Vincent J. Hellendoorn, Alexey Svyatkovskiy, Alberto Bacchelli, and Charles Sutton. 2023. Distilling Task-Specific Knowledge from Large Language Models for Code. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. ACM, 2010–2022. https://doi.org/10.1109/ICSE48619.2023.00175

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

[14] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint arXiv:1909.09436* (2019).

[15] Alex Kendall, Yarin Gal, and Roberto Cipolla. 2018. Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7482–7491.

[16] Hyunwoo Kim, Soonchul Lee, Jonghyun Park, and Minseok Choi. 2024. A Random Focusing Method with Jensen-Shannon Divergence for Improving Deep Neural Network Performance Ensuring Architecture Consistency. *Neural Networks* 177 (2024), 106–119.

[17] Yoon Kim and Alexander M. Rush. 2016. Sequence-Level Knowledge Distillation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 1317–1327.

[18] Raj Kumar, Anita Sharma, and Vijay Patel. 2025. An Explainable ADASYN-Based Focal Loss Approach for Credit Assessment. *Journal of Forecasting* 44, 3 (2025), 512–528.

[19] Chi Yeung Law, Linyuan Ding, Qing Dou, Jiangtao Zhou, Nan Chen, and Lei Zhang. 2024. AMR-Evol: Adaptive Modular Response Evolution Elicits Better Knowledge Distillation for Large Language Models in Code Generation. *arXiv preprint arXiv:2410.01425* (2024).

[20] Weifeng Li, Bowen Zhang, Zhenyu Chen, Bowen Yu, and Weifeng Sun. 2025. Improving Retrieval-Augmented Deep Assertion Generation via Joint Training. *IEEE Transactions on Software Engineering* 51, 2 (2025), 567–589.

[21] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*. 2980–2988.

[22] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=S1-2kv9eg

[23] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=Bkg6RiCqY7

[24] Jun Ma, Jianan Chen, Matthew Ng, Rui Huang, Yu Li, Cheng Li, Xiaokuan Yang, and Anne L Martel. 2020. Rethinking Dice Loss for Medical Image Segmentation. *IEEE Transactions on Medical Imaging* 40, 12 (2020), 3859–3871.

[25] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2018. Mixed precision training. In *International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=r1gs9JgRb

[26] Glenford J. Myers, Corey Sandler, and Tom Badgett. 2011. *The Art of Software Testing* (3rd ed.). John Wiley & Sons.

[27] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *International conference on machine learning (ICML)*. PMLR, 1310–1318.

[28] Sebastian Primbs, Benjamin Fein, and Gordon Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. *arXiv preprint arXiv:2502.02708* (2025).

[29] Sebastian Primbs, Benjamin Fein, and Gordon Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. *arXiv preprint arXiv:2502.02708* (2025).

[30] Rashi Puri, David Kung, Roberto Luss, Haitian Sun, Patrick D'Arcy, Krishnakumar Sankaranarayanan, and IBM Research AI for Code @ acumen.ai. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *NeurIPS 2021 Datasets and Benchmarks Track*.

https://arxiv.org/abs/2102.04664

[31] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. 3982–3992.

[32] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Yu, Xiaodong Xu, Ming Zhou, Phil Blunsom, et al. 2020. CodeBLEU: a method for automatic evaluation of code synthesis. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. 6114–6124.

[33] Sebastian Ruder. 2017. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098* (2017).

[34] 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.

[35] Carmine Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Mathew White, and Denys Poshyvanyk. 2020. AthenaTest: An Automatic Test Case Generation Tool for Java Based on a Transformer Model. In *Proceedings of the 28th International Conference on Program Comprehension (ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 317–321. https://doi.org/10.1145/3387904.3389297

[36] Michele Tufano, Shao Kun Deng, Neel Sundaresan, and Alexey Svyatkovskiy. 2022. Methods2Test: A Dataset of Focal Methods Mapped to Test Cases. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 299–303. https://doi.org/10.1145/3524842.3528009

[37] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Mastin White, and Denys Poshyvanyk. 2021. What Makes a Good Unit Test? An Empirical Study on Test Smells. *IEEE Transactions on Software Engineering* 47, 5 (2021), 990–1011. https://doi.org/10.1109/TSE.2019.2912111

[38] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2024. Improving the Learning of Code Review Successive Tasks with Cross-Task Knowledge Distillation. In *Proceedings of the 46th International Conference on Software Engineering*. 1–12.

[39] Xiaodong Wang, Yue Li, Wei Zhang, and Hao Chen. 2024. Adaptive Focal Loss for Keypoint-Based Deep Learning Detectors Addressing Class Imbalance. (2024), 1234–1243.

[40] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint* (2023).

[41] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[42] Bowen Yan, Weifeng Zhang, Zhenyu Chen, Bowen Yu, and Weifeng Sun. 2025. Retrieval-Augmented Fine-Tuning for Improving Retrieve-and-Edit Based Assertion Generation. *IEEE Transactions on Software Engineering* 51, 5 (2025), 1234–1256.

[43] Feng Yang, Xiaoming Liu, Guohua Wang, and Yuliang Chen. 2023. Focal Contrastive Learning for Palm Vein Authentication. *IEEE Transactions on Information Forensics and Security* 18 (2023), 2876–2887.

[44] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. 2024. Exploring Automated Assertion Generation via Large Language Models. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–32.

[45] Yifan Zhang, Chenyang Liu, Jiaming Wang, Bowen Li, and Yixuan Chen. 2025. Text-to-3D Generation using Jensen-Shannon Score Distillation. *arXiv preprint arXiv:2503.10660* (2025).

[46] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. (2015), 214–224.