Context-Aware Codebase Simplification for Large Language Models

Hanyi Liu
University of California: Davis
hnyliu@ucdavis.edu

Vincent Ng
University of California: Davis
wshng@ucdavis.edu

Etienne Rousseau
University of California: Davis
erousseau@ucdavis.edu

Abstract

Software development has evolved into a landscape where human programmers now collaborate alongside their personal Large Language Model agents. As a result, when programmers are working on large codebases, equally large amounts of context must be passed to these models; however, all models hit an overhead in context where their performance significantly degrades. Therefore, codebase context must be simplified for these models to sufficiently understand the codebase while still maintaining high performance. LEANCODE proposes to accomplish this through context-aware token removal, buildling off of the work of DietCode and SlimCode to achieve SOTA performance in both efficiency and effectiveness.

1 Introduction

Nowadays, software programming has evolved into a field of collaboration between the human programmer and Large Language Models (LLM). Therefore, just like how the human programmer must be completely knowledgeable in the codebase, so must the model be aligned with the codebase's context. However, the context that must be passed to the LLM grows proportionally to the size of the codebase, and the computation required with self-attention also grows at $O(n^2)$–leading quickly to the limiting overhead of context rot. This becomes exceedingly problematic, especially in software programming, where clean code architectures and consistent patterns are critical for a maintainable codebase.

Therefore, LEANCODE is proposed as a solution which allows the LLM to consume and understand the contexts required for large codebases while maintaining consistent performance and avoiding context rot (Wang et al., 2025). Through context-aware token removal, LEANCODE is able to significantly cut down required context lengths while still presenting the LLM with the entire codebase, allowing for LLMs to sufficiently understand much larger codebases while also saving computational costs in training and inference.

2 Method

2.1 Preliminaries

We reproduce the LEANCODE framework (Wang et al., 2025) for context-aware code simplification. The key idea is that token importance should be derived from *task-specific* attention, not generic self-attention averages.

We study two tasks on the Java split of CodeSearchNet (Husain et al., 2019): *code search* (classification) and *code summarization* (seq2seq). We use CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021). For code search, both models use encoder features with a binary classification head. For summarization, CodeT5 uses its native encoder-decoder generation (beam size 4), while our current CodeBERT summarization implementation uses a Transformer decoder with greedy decoding.

Given code snippet $d_j = \{t_1, \ldots, t_{n_j}\}$ and simplification ratio r, we remove $X = r \cdot n_j$ tokens by solving:

$$\min \sum_{i=1}^{n_j} w_i x_i \quad \text{s.t.} \quad \sum_{i=1}^{n_j} x_i = X, \;\; x_i \in \{0, 1\} \tag{1}$$

where w_i is token importance. Tokens with lowest scores are removed first.

2.2 Techniques

2.2.1 Method

LEANCODE uses different attention signals by task. For *code search*, we use final-layer encoder attention from the classification token position: true `[CLS]` for CodeBERT, and a first-token proxy for CodeT5 (which has no explicit `[CLS]`). Scores are averaged across heads. For *code summarization*,

we use final-layer encoder-decoder cross-attention, average over heads, then max-pool over decoder time steps to obtain one score per source token.

We then build a category-local dictionary keyed by $(\text{token}, \text{category})$ and compute the mean score per pair (Eq. 3 in LEANCODE). At simplification time, tokens are ranked in ascending score and the bottom $r\%$ are removed. We evaluate $r \in \{10\%, 20\%, 30\%, 40\%, 50\%\}$.

2.2.2 Dataset

We use the CodeBERT extension of CodeSearchNet, accessed from HuggingFace (`code_search_net`), which provides paired `(code, docstring)` examples. Each record includes (at minimum) `func_code_string` and `func_documentation_string`, along with pre-tokenized fields and language metadata. Although CodeSearchNet covers six languages (Python, Java, JavaScript, PHP, Ruby, and Go), we follow LeanCode and restrict our experiments to **Java only**.

We evaluate two downstream tasks built on the same Java corpus. For **Code Search**, we use 908,886 training samples, 30,655 for validation, and 26,909 for testing. For **Code Summarization**, we use 164,923 training samples, 5,183 for validation, and 10,955 for testing.

For model input construction, we treat `func_code_string` as source code and `func_documentation_string` as the natural-language target/query. This setup supports both discriminative code-search training and sequence-to-sequence summarization under a consistent data schema.

2.2.3 Splitting Tokens into Categories

To model token importance in context, we assign each source token a semantic category derived from its AST location rather than from surface text alone. Each Java method is parsed with `tree-sitter-java`, then traversed to map statement-level nodes (e.g., method signature, variable declaration, assignment, method call, return, loop, and exception-handling constructs) to predefined category labels. During traversal, category context is inherited from parent nodes unless a new statement boundary is encountered, which lets nested tokens keep the correct local role. Afterward, model subword tokens are aligned back to source spans and matched to the corresponding AST token/category so every token receives a context label. We then aggregate attention scores by $(\text{token}, \text{category})$ and compute category-local means, producing the lookup used at simplification time: lower-scored tokens in their local category are removed first for a given simplification ratio.

2.2.4 Model Finetuning

In order to extract accurate attention scores for our (token, category) pairs, we need models that have been trained on these two tasks, since attention varies and is task-specific. To accomplish this, we finetune the CodeBERT and CodeT5 model architectures to be suitable for code search and code summarization.

CodeBERT for Code Search: Code search at its core is a binary classification task. Given a text query and a code snippet, a proposed model needs to classify if the snippet matches the query or not. Therefore, we format this task's input by concatenating the code snippet after the text query, separate by the separator token, then pass this as input into our transformer models. For CodeBERT, since it is an encoder-only model, we simply append a two-output Linear layer to the last layer of the encoder, where one neuron represents 'No Match' and the other represents 'Match'.

CodeT5 for Code Search: Since CodeT5 follows an encoder-decoder architecture, we ignore the decoder half for code search and apply the same binary classifier to the end of CodeT5' encoder layer. However, for computational efficiency, we note that we have to make sure to not call the decoder when running CodeT5 for code search.

CodeBERT for Code Summarization: Code summarization is instead a autoregressive generation task, therefore requiring a decoder pair to be able to generate the output summary. The input in this case would be only the code snippet. For CodeBERT, however, since it is an encoder-only model, we therefore had to add our own decoder layer to CodeBERT to enable it for the task of code summarization. To do so, we utilized a 6-layer decoder architecture with 12 separate heads, which is then passed into cross-attention with the encoder's output embeddings, and finally we generate the new output token through a Linear layer. The output token is concatenated to the decoder's generated tokens until the model hits max length or outputs the [EOS] token. This generated output therefore represents our code summary.

CodeT5 for Code Summarization: Since CodeT5 is already an encoder-decoder model, no

further finetuning of the model architecture was necessary.

2.2.5 Model Training

We train the two models using the same distinct methods for code search and code summarization, since they both essentially have the same high-level model architectures for each task.

Training for Code Search: Due to code search being a binary classification problem, we utilized a cross entropy loss as the criterion for training CodeBERT and CodeT5 for code search. We utilize the AdamW optimizer for training, which is an Adam optimizer that utilizes decoupled weight decay. This same method is applied for training both CodeBERT and CodeT5 for code search.

Training for Code Summarization: Code summarization, on the other hand, is an autoregressive generation task. Therefore, we must conduct cross entropy over each generated token and the actual matching token pair, and hence we employ a token-level cross-entropy loss as the criterion, which we note is also the criterion used in the pretrained CodeT5 models. Similar to Code Search, this is optimizer through the AdamW optimizer, and we apply this training method to both CodeBERT and CodeT5 for code summarization.

However, we note that since code search and summarization are different tasks, we have to save separate model checkpoints for each model and task pair. Therefore, we end up with four final trained model checkpoints, an encoder-only (for search) and an encoder-decoder (for summarization) version for each of the two models.

2.2.6 Attention Extraction

Once fine-tuned checkpoints are available, we extract per-token attention scores over the full CodeSearchNet Java training split using a dedicated extraction pipeline. For each sample, the code snippet is tokenized without truncation; sequences exceeding 510 content tokens are split into overlapping windows of 510 tokens with a stride of 382 tokens (128-token overlap), processed independently, and stitched back together by mean-averaging scores in overlap regions so that every token in the original sequence receives exactly one final score regardless of sequence length.

For *code search* (Eq. 5), we extract the final encoder layer's attention matrix, average across all heads, and take the row corresponding to the `[CLS]` token, which represents the attention the classification token pays to every other source token. For CodeT5, which has no explicit `[CLS]` token, the first token of the encoder output serves as a proxy aggregation point, consistent with T5's encoder architecture.

For *code summarization* (Eq. 6), we extract the cross-attention matrix of the final decoder layer, average across all heads, and max-pool over all decoder time steps to produce one scalar score per source token. Ground-truth descriptions are used as decoder input via teacher forcing, matching the supervised fine-tuning setup.

A practical challenge arose for CodeBERT summarization: because CodeBERT's decoder is a manually assembled `nn.TransformerDecoder` rather than a HuggingFace encoder-decoder model, cross-attention weights are not exposed through the standard `model()` forward call. We resolve this by calling the encoder and decoder separately — first running the encoder to obtain `last_hidden_state`, then passing it as `encoder_hidden_states` directly into the decoder, which correctly returns `cross_attentions` from each decoder layer. The final decoder layer's cross-attention is then averaged over heads and max-pooled over decoder steps, identical to the CodeT5 summarization path.

Following extraction, per-token scores are aggregated into a category-local score dictionary by grouping all scores for each $(\text{token}, \text{category})$ pair across the entire training set and computing their mean (Eq. 3). This produces a flat lookup score_dict[token][category] $\rightarrow \mathbb{R}$ used at simplification time.

2.2.7 Dataset Simplification

Given the per-token attention scores and their assigned AST categories, we construct a category-local score dictionary by grouping all scores for each $(\text{token}, \text{category})$ pair across the entire training set and computing their mean (Eq. 3). This produces a flat lookup score_dict[token][category] $\rightarrow \mathbb{R}$ used at simplification time. For tokens present in the score dictionary but not under the queried category, we fall back to the mean score across all categories for that token. For tokens absent from the dictionary entirely, we fall back to the global mean score across all token-category pairs.

To simplify a code snippet at ratio r, each subword token is first looked up in the score dictionary by its normalized surface form (subword markers `Ġ`, `##`, and `_` stripped, lowercased) and its AST

category. Special tokens (`[CLS]`, `[SEP]`) are excluded. `[SEP]`, `<s>`, `</s>`) are assigned a score of ∞ and are excluded from removal candidates. The number of tokens to remove is computed as $X = \lfloor r \cdot |\mathcal{T}_{\text{removable}}| \rfloor$, where $|\mathcal{T}_{\text{removable}}|$ is the count of non-special tokens. Tokens are then ranked in ascending order of importance score, and the bottom X tokens are removed by solving Eq. 1. Removal is performed at the character level: the source character spans corresponding to removed subword tokens are deleted from the original code string, and consecutive blank lines are collapsed to at most two newlines.

We evaluate simplification ratios $r \in \{10\%, 20\%, 30\%, 40\%, 50\%\}$, producing five simplified variants of the test set for each of the four model-task combinations, yielding twenty simplified datasets in total:

1. CodeBERT encoder-only (code search)
2. CodeBERT encoder-decoder (code summarization)
3. CodeT5 encoder-only (code search)
4. CodeT5 encoder-decoder (code summarization)

Each simplified dataset record preserves the original code, the simplified code, the natural-language description, and per-sample statistics (tokens removed, original token count), enabling downstream evaluation under identical conditions to the baseline.

3 Experiments

3.1 Evaluation Setup

To evaluate the effectiveness of our simplified datasets, we compare performance of both CodeBERT and CodeT5 for the two respective tasks with the different levels of simplified datasets as input against the baseline performance of the unsimplified dataset.

3.2 Metrics

Evaluating Code Search: To evaluate code search, we utilize the Mean Reciprocal Rank (MRR) metric, which is defined as:

$$\text{MRR} = \frac{1}{|N|} \sum_{i=1}^{N} \frac{1}{rank_i}$$

Where $N =$ *# of samples* and
$rank_i =$ *Rank of ground-truth code snippet*.

Therefore, a perfect prediction where the ground-truth snippet is ranked 1 would have a full score of 1 for that sample, while any incorrect lower ranking would have a inversely proportional lower score. This is then averaged across all samples to obtain an overall metric.

Evaluating Code Summarization: To evaluate code summarization, we utilize the Bilingual Evaluation Understudy metric with a maximum 4-gram order (BLEU-4), defined as:

$$\text{BLEU-4} = BP \cdot \exp\left(\frac{1}{4} \sum_{n=1}^{4} \ln p_n\right)$$

Where BP is the Brevity penalty and p_n is the n-gram precision for an order of 4. The Brevity penalty is defined as:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

Where c is the length of the generated summary and r is the actual summary length. This Brevity penalty is utilized here to penalize model responses that are too short but matches the predicted n-grams, hence why there is no penalty for predicted responses that are longer than the actual summary length.

Therefore, a response that perfectly matches the actual response would have a BLEU-4 score of 1.0, since it would have full precision for all n-gram precisions, while a response that doesn't match at all would have a BLEU-4 of 0.0.

3.3 Results

Tables 1 and 2 present our reproduction results for code search (MRR) and code summarization (BLEU-4), alongside the original LEANCODE results reported by Wang et al. (2025). We evaluate each model at simplification ratios of 10%–50% on a randomly sampled subset of 1,000 test examples from the CodeSearchNet Java test split.

Our baseline MRR and BLEU-4 scores are lower than those reported by Wang et al. (2025). We attribute this gap to three factors. First, our models were fine-tuned for only one epoch due to compute constraints, whereas the original work trained to convergence on two NVIDIA A100 GPUs. Second, we evaluate on a random subset of 1,000 test

Ratio	CodeBERT		CodeT5	
	Ours	Original	Ours	Original
Base	0.613	0.726	0.634	0.747
10%	0.620 (+1.14%)	0.728 (+0.27%)	0.632 (−0.32%)	0.743 (−0.53%)
20%	0.602 (−1.79%)	0.719 (−0.96%)	0.625 (−1.42%)	0.736 (−1.47%)
30%	0.594 (−3.10%)	0.716 (−1.37%)	0.617 (−2.68%)	0.724 (−3.07%)
40%	0.576 (−6.04%)	0.697 (−3.99%)	0.609 (−3.94%)	0.714 (−4.41%)
50%	0.552 (−9.95%)	0.688 (−5.23%)	0.601 (−5.21%)	0.706 (−5.48%)

Table 1: Code Search (MRR). Percentages show change from baseline. "Original" refers to LEANCODE (Wang et al., 2025).

Ratio	CodeBERT		CodeT5	
	Ours	Original	Ours	Original
Base	16.74	18.25	18.94	20.55
10%	16.59 (−0.90%)	18.08 (−0.93%)	18.66 (−1.48%)	20.32 (−1.11%)
20%	16.32 (−2.51%)	17.73 (−2.84%)	18.47 (−2.48%)	20.18 (−1.80%)
30%	15.94 (−4.78%)	17.23 (−5.58%)	18.34 (−3.17%)	19.82 (−3.55%)
40%	15.57 (−6.99%)	16.71 (−8.43%)	17.82 (−5.91%)	19.27 (−6.22%)
50%	15.13 (−9.62%)	16.24 (−11.01%)	17.21 (−9.13%)	18.46 (−10.17%)

Table 2: Code Summarization (BLEU-4). Percentages show change from baseline. "Original" refers to LEANCODE (Wang et al., 2025).

samples rather than the full test split, introducing sampling variance. Third, during reproduction we identified and fixed several implementation issues in the checkpoint-loading pipeline, including missing tokenizer artifacts and a Python truthiness bug in the decoder's start-token selection (`bos_token_id=0` evaluates as `False`, causing initialization with `<pad>` instead of `<s>`).

Despite lower absolute scores, the *relative trends* remain consistent with the original findings. For code search, MRR shows the same qualitative pattern where performance slightly improves at low removal rates (e.g., 10%) before degrading, supporting the claim that removing low-importance tokens can improve model focus. For code summarization, BLEU-4 degradation closely tracks the original results, confirming that LEANCODE's context-aware simplification preserves task-relevant information even under substantial token removal.

3.4 Comparison With Original Paper

Our reproduction confirms the central finding of Wang et al. (2025): attention-based token removal preserves downstream performance significantly better than random removal, with both MRR and BLEU-4 degrading gracefully as the simplification ratio increases. While we restrict our evaluation to CodeBERT and CodeT5 due to compute and API constraints (excluding larger proprietary models such as GPT-4o), the consistency of trends across both tasks supports the generalizability of the LEANCODE framework across model architectures.

4 Related Work

Pre-trained transformer models have driven significant advances in code intelligence tasks. CodeBERT (Feng et al., 2020) extended the RoBERTa architecture (Liu et al., 2019) to bimodal pre-training on natural language and code, establishing strong baselines for code search and documentation tasks. CodeT5 (Wang et al., 2021) built on the T5 encoder-decoder framework with identifier-aware pre-training, achieving state-of-the-art performance on code summarization.

The use of attention weights as a proxy for token importance has been widely studied in NLP. Clark et al. (2019) found that certain BERT attention heads reliably attend to syntactically meaningful positions, while Jain and Wallace (2019) and Wiegreffe and Pinter (2019) debated whether attention constitutes a faithful explanation signal. In the code domain, Wan et al. (2022) showed that CodeBERT attends to structurally important tokens such as method signatures and return statements, supporting attention-based token importance estimation.

Incorporating AST structure into code models has also shown consistent gains. Guo et al. (2021) augmented CodeBERT with data flow graphs derived from ASTs, while Zhang et al. (2019) proposed AST-based neural networks for code summarization. Our work uses tree-sitter (Brunsfeld et al., 2018) to assign AST node categories to subword tokens, enabling the category-local attention aggregation central to LEANCODE.

5 Future Steps

While LeanCode does achieve state of the art performance as compared to methods like DietCode and SlimCode, it still remains a static approach to code simplification. Furthermore, LeanCode's method at trying to address this through category assignment also adds human bias of rule-based implementation. Therefore, while some contextualized understanding is gained, context is still unique for each code snippet that is passed into any model, hence LeanCode misses the unique context of each code snippet.

To address the context uniqueness of all code snippets, we must approach code simplification in a dynamic method that can capture code context

and dynamically assign importance score to each token before inference. A perfect dynamic method would satisfy the following:

1. Tokenized code snippet as input and outputs importance score for each individual token.
2. The compute required for this importance calculation must be much smaller than passing the code snippet through a multi-layer transformer model.
3. The input code snippet would be passed through this method's model first to calculate a simplified version, then this would be passed into the actual LLM.

The dynamic method would then be some form of a neural network model. To train this potential model, we build off of LeanCode's assumption that the attention score of the last transformer layer is most indicative of a token's importance. With this in mind, we then essentially have free training data. For all code snippets, we can pass it through our finetuned transformer models, extract their attention scores, and use the (code snippet, attention scores) pair as training data for our dynamic model.

We initially propose the following potential model architectures, in increasing levels of complexity:

1. **Linear Network**: Takes in current token and information such as location and category as input. Pass through a multi-layer architecture with one neuron as output for score prediction.
2. **Convolutional Neural Network**: Takes in current code snippet. Pass through a 1x1 CNN with kernel size influencing how much relative context each token sees, and outputs array of importance scores for each token.
3. **Recurrent Neural Network**: Takes in current code snippet. Pass through a bi-directional, many-to-many RNN, and outputs array of importance scores for each token in snippet.

We also note that we will need to evaluate these models on newer state of the art coding LLMs. While CodeBERT and CodeT5 were good baselines to measure against, they are too old nowadays to gain relevant improvements off of. Therefore, we would need to evaluate through newer SOTA models such as Kimi-Dev, Qwen3-Coder, and DeepSeek-V3.

While we do not have time this quarter to implement these steps, they are future plans that we do plan to bring into real results.

6 Conclusion

LEANCODE addresses a fundamental bottleneck in LLM-assisted software development: as codebases grow, the context passed to models grows with them, eventually degrading model performance through context rot. By leveraging task-specific attention signals from fine-tuned models to identify and remove low-importance tokens in an AST-aware manner, LEANCODE demonstrates that significant reductions in context length (up to 50%) can be achieved while preserving the majority of downstream task performance. Our reproduction validates the central claims of Wang et al. (2025) on the CodeSearchNet Java benchmark across both code search and code summarization tasks using CodeBERT and CodeT5. Despite operating under tighter compute constraints and resolving several non-trivial implementation challenges along the way, our results align with the original paper's qualitative findings: attention-guided simplification consistently outperforms the degradation that would be expected from naive token removal, and the category-local aggregation of attention scores provides a robust signal for token importance across diverse code constructs. Seeing these results suggests that context-aware simplification is a promising and practical direction for enabling LLMs to operate effectively over large codebases without sacrificing task fidelity.

References

Max Brunsfeld et al. 2018. Tree-sitter: An incremental parsing system for programming tools. https://tree-sitter.github.io.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. 2019. What does BERT look at? an analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *Proceedings of the 9th International Conference on Learning Representations*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. In *arXiv preprint arXiv:1909.09436*.

Sarthak Jain and Byron C. Wallace. 2019. Attention is not explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 3543–3556.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*.

Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1377–1389.

Yan Wang, Ling Ding, Tien N. Nguyen, Shaohua Wang, and Yanan Zheng. 2025. LEANCODE: Understanding models better for code simplification of pretrained large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1551–1567. Association for Computational Linguistics.

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

Sarah Wiegreffe and Yuval Pinter. 2019. Attention is not not explanation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, pages 11–20.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*, pages 783–794.