ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

Efficient Decoder-Only Transformer with Domain-Specific Byte Pair Encoding for Python Code Generation

Dr. P. Venkateshwarlu[1], Dr. M. Srinivas[2], Dr. Ravikumar Thallapalli[3]

[1]Assistant Professor, Department of Computer Applications, Vaageswari College of Engineering (Autonomous), Karimnagar, Telangana, India

[2]Assistant Professor, Department of CSE (AI&ML), Vaageswari College of Engineering (Autonomous), Karimnagar, Telangana, India

[3]Assistant Professor, Department of Computer Science and Engineering, Vaageswari College of Engineering (Autonomous), Karimnagar, Telangana, India

DOI: https://doi.org/10.51583/IJLTEMAS.2026.151000096

Received: 28 October 2026; Accepted: 02 November 2026; Published: 10 November 2026

ABSTRACT

Efficient code generation requires a balance between model quality and computational cost, particularly when experiments are performed on resource-constrained hardware. This work investigates the effect of domain-specific Byte Pair Encoding (BPE) on a compact decoder-only Transformer for Python code generation. An 8,000-token Python-specific BPE tokenizer is compared with the 50,258-token GPT-2 tokenizer using the same compact architecture, consisting of two Transformer layers, four attention heads, a model dimension of 128, a feed-forward dimension of 512, and a maximum sequence length of 256 tokens. Experiments use 15,000 training functions, 2,000 validation functions, and 3,000 independent test functions selected with repository-level separation and cross-split duplicate-code verification. On the independent test set, the Python-specific tokenizer reduces average tokenization length by 41.94% and training blocks by 33.89%. The resulting model contains 81.37% fewer parameters, requires 93.79% less training time, and uses 72.40% less peak GPU memory during test evaluation than the GPT-2-tokenizer configuration. However, these efficiency gains are accompanied by lower language-model and code-generation performance. The Custom-BPE model obtains a test perplexity of 125.14, compared with 15.68 for the GPT-2 configuration. These perplexity values are measured over their respective token distributions and are therefore not directly comparable across vocabularies. Under controlled deterministic generation, the Custom-BPE and GPT-2 models produce syntactically valid and compilable outputs in 0% and 26% of cases, respectively. The results therefore characterize an explicit efficiency-versus-generation-quality trade-off rather than a universal quality advantage for domain-specific tokenization.

Index Terms—Python code generation, Byte Pair Encoding, domain-specific tokenization, decoder-only Transformer, code language modeling, resource-efficient AI, functional code-generation evaluation.

INTRODUCTION

Transformer-based language models have substantially improved automatic source-code generation. Autoregressive decoder-only models formulate code synthesis as next-token prediction and can learn recurring programming constructs, identifiers, expressions, and implementation patterns from large code corpora [1]–[3]. Code-oriented systems such as CodeGen, CodeT5, CodeGeeX, Code Llama, and StarCoder demonstrate the effectiveness of Transformer pretraining for code tasks, but their scale also increases the computational requirements of training and experimentation [3], [5], [6], [10], [12].

For academic laboratories and educational environments, the computational cost of large code models can be a practical barrier. An alternative is to deliberately construct a compact model and examine which representation choices most strongly affect its computational footprint. Tokenization is particularly relevant

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

because it determines the sequence presented to the language model, the number of prediction steps, the number of fixed-length causal-language-model blocks, and the size of vocabulary-dependent embedding and output components.

Byte Pair Encoding (BPE) provides a practical subword representation by constructing a finite vocabulary from recurring symbol sequences while retaining decomposition for rare strings [8]. For Python, a domain-specific vocabulary can allocate capacity to recurring keywords, operators, identifiers, punctuation, and code fragments rather than tokens that are frequent in unrelated natural-language domains. However, representation efficiency does not necessarily imply better code generation: different token boundaries can change the difficulty of learning syntax and identifier relationships.

This work therefore asks: can a Python-specific BPE tokenizer substantially reduce the computational requirements of a compact decoder-only Transformer, and what quality trade-off accompanies that reduction? The study uses the Python subset of CodeSearchNet [4] and a controlled tokenizer ablation. The processed corpus contains 412,178 Python functions from 12,231 repositories. A leakage-safe sample of 15,000 training, 2,000 validation, and 3,000 independent test functions is constructed using repository-level separation and exact code-hash verification. The 8,000-token Custom Python BPE is compared with the 50,258-token GPT-2 tokenizer while keeping the Transformer architecture and principal training settings fixed.

The contributions are: (1) a leakage-safe tokenizer-ablation protocol with repository-level separation and cross-split duplicate verification; (2) an 8,000-token Python-specific BPE trained only on the training split; (3) quantitative measurement of token compression, causal-language-model block reduction, parameter count, training time, and GPU memory; (4) independent validation/test language-model evaluation; and (5) paired deterministic code-generation evaluation using syntax validity and compilation. The resulting contribution is an empirical characterization of the efficiency-versus-generation-quality trade-off in a compact Python code model.

Related Work

The Transformer architecture introduced self-attention as the central mechanism for sequence modeling and enabled highly parallel training [1]. GPT-2 demonstrated the effectiveness of autoregressive Transformer language modeling at scale [2]. CodeGen extended this paradigm to program synthesis and released models up to 16.1B parameters [3]. CodeT5 explored an encoder-decoder architecture with identifier-aware pretraining for code understanding and generation [5], while CodeGeeX addressed multilingual code generation and introduced HumanEval-X [6]. Code Llama specialized the Llama family for code generation, and StarCoder provided a large open code model trained on permissively licensed source data [10], [12]. StructCoder incorporated syntax and data-flow information into a structure-aware encoder-decoder design [7].

Subword tokenization addresses open-vocabulary problems by decomposing rare strings into reusable units [8]. The present study applies this principle to a narrower question: how much computational reduction can be obtained when a Python-specific vocabulary replaces a much larger general-purpose tokenizer in a deliberately compact decoder-only model? This differs from approaches such as LoRA, which reduce the cost of adapting an existing pretrained model [9], and from LLaMA/Llama 2, which investigate scaling and efficient large-model pretraining [11], [13].

Evaluation practice also motivates the experimental design. HumanEval measures functional correctness using execution-based tests [14], while MBPP evaluates short Python program synthesis problems [15]. Because a full benchmark run was beyond the present compact experimental budget, this study uses a paired 50-function held-out evaluation as diagnostic evidence and explicitly treats it as complementary to perplexity rather than as a benchmark-level pass@k result.

CodeGen demonstrates that decoder-only models can be scaled specifically for program synthesis, including multi-turn decomposition of programming tasks [3]. Code Llama extends this direction with Python specialization, infilling, and long-context support [10]. Another significant open code-model reference point is StarCoder, which is especially pertinent since it assesses the code generation, and focuses on open research

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

and attribution [12]. These systems create a capability-based higher context of code modeling, but the current work makes a deliberate effort to operate at a scale several orders of magnitude smallerthan these.

Quality through structure-aware approaches is an alternative way. StructCoder uses abstract syntax and data-flow data in the encode and decode processes [7]. This kind of structural supervision can enhance the fidelity of code-generation as well as introduce an architectural and preprocessing complexity. The current research deliberately neglects to add other structural modules in order that the tokenizer vocabulary is the major controlled variable.

The parameter-efficient adaptation mechanisms like LoRA minimize the trainable parameters in the fine-tun-

ing process [9], but they presuppose the availability of a pretrained model. The papers LLaMA and Llama 2 show that simply picking scaling and training-data choices can enhance the efficiency frontier of large foundation models [11], [13]. Conversely, the current paper poses a previous design question: to what extent can the computational footprint of a small code model be just due to the vocabulary of the language used to describe code?

The research gap is therefore narrower than “small models are better”: prior work has strongly established the capability of large code models, but the computational role of tokenizer vocabulary is less directly isolated in a compact Python-only decoder. The present study fixes model architecture and training data and varies the tokenizer, allowing tokenization efficiency and quality trade-offs to be observed under controlled conditions.

PROPOSED METHODOLOGY

A. Dataset Preparation and Leakage-Safe Splitting

The experiments use the Python portion of CodeSearchNet [4]. The loaded corpus contains 412,178 non-empty Python functions from 12,231 repositories. Function character lengths range from 75 to 103,665 characters, with mean 1,056.43 and median 666 characters. The experimental sample contains 15,000 training, 2,000 validation, and 3,000 test functions. Repository identifiers were used to enforce disjoint repository groups, yielding zero repository overlap among the three splits. Exact SHA-256 code hashes were also compared, yielding zero cross-split duplicate-code matches.

The custom tokenizer was trained only on the training functions. The validation split was used for model selection and convergence monitoring, and the test split was held out until final evaluation. This separation prevents test functions from influencing tokenizer construction or checkpoint selection.

Before training, tokenizer-length statistics were measured on a 10,000-function diagnostic sample. Under GPT-2 tokenization, the mean was 482.36 tokens/function and the median was 284, with 54.81% exceeding 256 tokens. The diagnostic tail reached 9.49% above 1,024 tokens. This preliminary analysis motivated the 256-token context and the subsequent held-out tokenizer ablation. On the final 3,000-function test set, the Custom BPE reduced the average from 504.00 to 292.61 tokens/function, confirming that the observed compression was not an artifact of the smaller diagnostic sample.

B. Domain-Specific BPE and GPT-2 Baseline

A Python-specific BPE tokenizer was trained from the training split with vocabulary size 8,000. Four special tokens were explicitly assigned: <pad> = 0, <bos> = 1, <eos> = 2, and <unk> = 3. The baseline uses the GPT-2 tokenizer with a 50,258-token vocabulary and a dedicated padding ID of 50,257. Both tokenizers process the same function-level splits and use a maximum sequence length of 256 tokens.

C. Compact Decoder-Only Transformer

Both configurations use the same decoder-only Transformer: two Transformer layers, model dimension 128, four attention heads, feed-forward dimension 512, and maximum sequence length 256. The vocabulary is the

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

principal architectural variable because token embeddings and the vocabulary projection scale with vocabulary size. For input sequence $X=(x_1,\ldots,x_t)$, the model produces a distribution for each next token under causal masking. The attention operation is:

$$Attention(Q,K,V) = softmax(QK^T/\sqrt{d_k} + M)V. \quad (1)$$

where Q, K, and V are query, key, and value matrices and M is the causal mask. The training objective is token-level cross-entropy over non-padding targets:

$$L = -(1/N)\ \Sigma_{i=1}^{N} \log P_\theta(y_i \mid x_i), \quad (2)$$

where N is the valid target tokens. Padding positions are not included into the loss. Perplexity is defined as PPL = exp(L) and this definition is applied everywhere when training, validating and reporting test results.

Experimental Setup

A. Configuration and Environment

TABLE I. EXPERIMENTAL CONFIGURATION

Parameter	Custom BPE	GPT-2
Vocabulary	8,000	50,258
Model dimension	128	128
Heads / layers	2-Apr	2-Apr
FFN dimension	512	512
Context	256	256
Batch size	8	4
Epochs	3	3
Learning rate	3×10^{-4}	3×10^{-4}
Gradient clipping	1	1
Seed	42	42

The experiments were all run using Python 3.11.16 and PyTorch 2.13.0+cu126 and CUDA 12.6 on an NVIDIA GeForce MX330 with 2 GB of VRAM. Experiment set up was frozen and then subjected to final training and evaluation. Three epochs were made with AdamW [16] and a learning rate 3×10^{-4} and a gradient clipping of 1.0. The Custom BPE and GPT-2 tokenizer configurations used batch sizes of 8 and 4 respectively, as the larger vocabulary of the GPT-2 tokenizer required more GPU memory; this batch-size disparity is a methodological limitation that should be noted when comparing training-time measurements between the two configurations.

Training included explicit sanity checks before the full runs. A forward/backward smoke test confirmed finite logits, finite loss, and finite gradients for both configurations. The Custom-BPE model contains 2.48M parameters and the GPT-2-vocabulary model 13.30M parameters under the otherwise identical Transformer dimensions. Independent checkpoints were saved after each epoch, and the checkpoint with the lowest validation loss was selected. This procedure yielded epoch 3 as the best checkpoint for both models.

B. Causal-LM Dataset Construction

Tokenized functions were converted into fixed-length causal-language-model blocks with maximum length 256. Custom BPE produced 23,428 training blocks, 3,120 validation blocks, and 4,959 test blocks. GPT-2 produced 35,436, 4,534, and 7,338 blocks, respectively. The custom tokenizer therefore reduces the training-block count by 33.89% under identical function-level splits.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

C. Evaluation Protocol

Four aspects were evaluated: tokenization efficiency, language-model quality, computational efficiency, and functional generation. Tokenization was measured on the independent 3,000-function test set. Language-model quality used validation and independent test loss/PPL, with the best validation checkpoint evaluated once on the test set. Computational measurements included parameter count, training time, and peak GPU memory. Functional generation used 50 common held-out functions for both models; outputs were retained without manual correction and checked for Python syntax and compilation. Functional generation prompts were constructed as bare function signatures of the form 'def function_name(parameters):' drawn from the held-out set; each model was given at most 256 new tokens to generate the function body. A deterministic greedy-decoding strategy (temperature = 0) was applied, with each model's tokenizer used for encoding the prompt and decoding the output tokens back to text. Generated outputs were checked for Python syntax validity using Python's built-in compile() function, without manual correction.

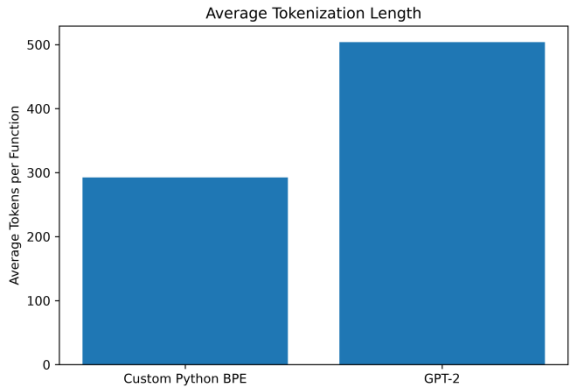


Fig. 1. Average tokenization length on the independent test set.

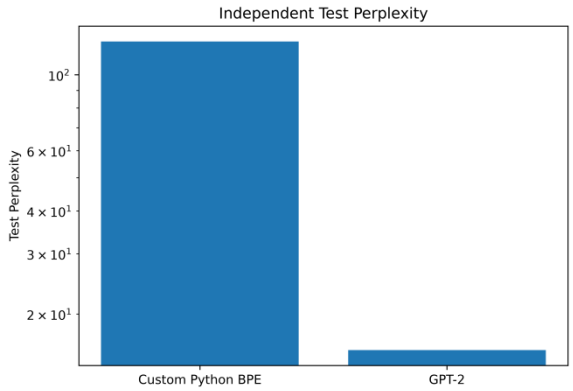


Fig. 2. Independent test perplexity; logarithmic scale is used to show the large difference.

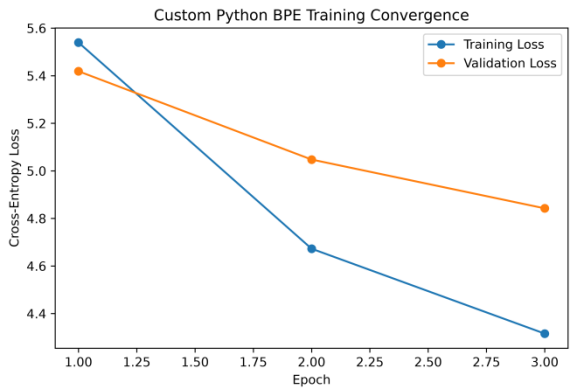


Fig. 3. Custom-BPE training and validation loss across three epochs.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

RESULTS AND DISCUSSION

A. Dataset and Leakage Validation

The final experimental corpus contains 412,178 functions from 12,231 repositories. The selected train/validation/test subsets have 15,000/2,000/3,000 functions, respectively, with zero repository overlap and zero exact duplicate code across splits.

Dataset-index diagnostics also confirmed that selected metadata code matched the original dataset source exactly for sampled records. These checks support the use of the 3,000-function test set as an independent evaluation set.

B. Training Convergence

TABLE II. CUSTOM-BPE TRAINING CONVERGENCE

Epoch	Custom Train	Custom Val.	Custom Val. PPL
1	5.5398	5.4188	225.61
2	4.6727	5.0475	155.64
3	4.3157	4.8424	126.77

The Custom-BPE model shows a consistent decrease in training loss across all three epochs, from 5.5398 to 4.3157. Validation loss decreases from 5.4188 to 4.8424, with validation PPL falling from 225.61 to 126.77. The third epoch is selected because it provides the best validation loss among the saved checkpoints.

The loss of the independent test of 4.8294 is within the same range as the ultimate validation loss indicating that the checkpoint of choice is stable.

C. Tokenizer Ablation

TABLE III. TOKENIZER ABLATION ON THE INDEPENDENT TEST SET

Metric	Custom BPE	GPT-2
Vocabulary	8,000	50,258
Avg. tokens/function	292.61	504
Median	176	288.5
P90	581.1	1,007.20
P95	836.15	1,544.00
P99	2,141.11	3,676.37
>256 tokens	32.67%	56.03%
Characters/token	3.7196	2.1595

Custom Python BPE decreases the average length of tokenization, 504.00 tokens per function, by 41.94%, to 292.61 tokens per function. Character to token ratio also goes up to 3.7196 as compared to 2.1595.

The tail of the long sequence is also significantly different: 32.67% of Custom-BPE functions have more than 256 tokens in their tail, as compared to 56.03% of the GPT-2 functions. Therefore, domain-specific tokenization yields fewer average representation length and functions and fewer functions that spread to the fixed context window.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

D. Causal-Language-Model Block Reduction

TABLE IV. CAUSAL-LANGUAGE-MODEL BLOCK COUNTS

Configuration	Train	Val.	Test
Custom BPE	23,428	3,120	4,959
GPT-2	35,436	4,534	7,338

With the same split of functions and 256-token context, Custom BPE creates 23,428 training blocks versus 35,436 of GPT-2, a 33.89% decrease. This finding is significant since the tokenizer does not only influence the length of sequences but also the number of fixed-length training data that the Transformer is trained on.

E. Language-Model Performance

TABLE V. VALIDATION AND INDEPENDENT TEST PERFORMANCE

Model	Best epoch	Val. loss	Val. PPL	Test loss	Test PPL
Custom BPE	3	4.8424	126.77	4.8294	125.14
GPT-2	3	2.7555	15.73	2.7522	15.68

The GPT-2 tokenizer set-up has significantly lower perplexity. Custom BPE reaches test loss 4.8294 and PPL 125.14, while GPT-2 reaches loss 2.7522 and PPL 15.68. Both configurations have a close relationship between validation and test values showing no large degradation of validation-to-test. Notably, the reported PPL values all fit PPL=exp(L).

However, because each perplexity is computed over its own tokenizer's vocabulary and token distribution, the absolute values are not directly scale-comparable across the two configurations; a tokenizer-independent metric such as bits per byte (BPB) would be required for a fully fair cross-tokenizer comparison.

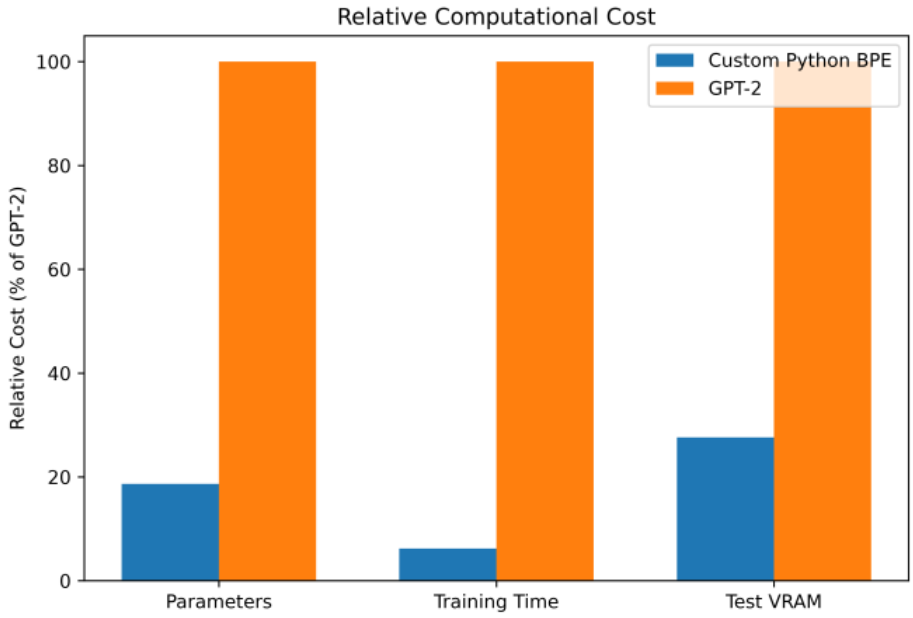


Fig. 4. Relative computation cost, scaled to the GPT-2 setting.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

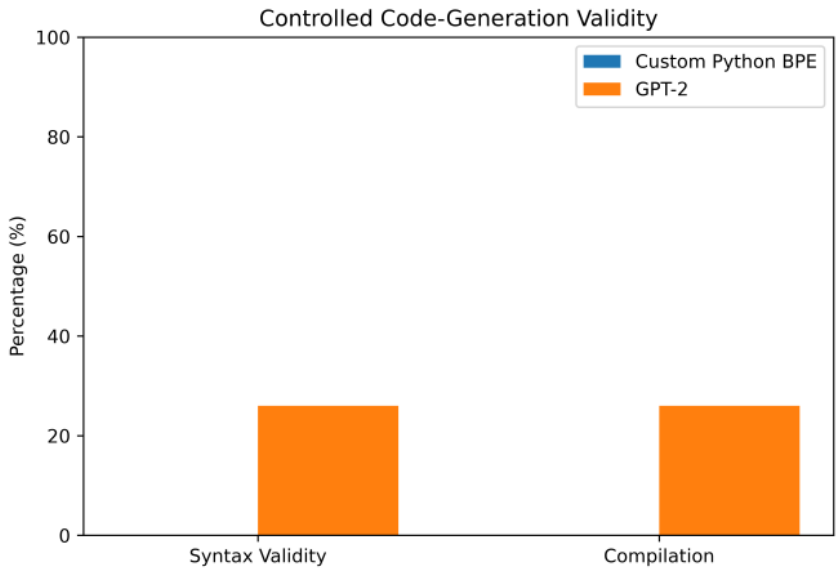


Fig. 5. Measured deterministic syntax-validity and compilation rates.

F. Computational Efficiency

TABLE VI. COMPUTATIONAL EFFICIENCY COMPARISON

Metric	Custom BPE	GPT-2	Reduction
Parameters	2,477,568	13,295,616	81.37%
Train blocks	23,428	35,436	33.89%
Training time	21.49 min	345.62 min	93.79%
Peak test VRAM	227.16 MB	823.05 MB	72.40%

The Custom-BPE model contains 2,477,568 parameters versus 13,295,616 for GPT-2, an 81.37% reduction. Since the Transformer depth, hidden dimension, attention heads, feed-forward dimension and context length are constant, the main architectural distinction is vocabulary-dependent embedding and output size. The training time is reduced by 93.79% by a measured 345.62 to 21.49 minutes. The maximum memory in the GPU decreases by 72.40% during an independent test assessment and ranges between 823.05 and 227.16 MB. These findings are a solid argument in favor of the computational-efficiency argument, but do not suggest high-quality generation.

G. Controlled Functional Code Generation

TABLE VII. CONTROLLED DETERMINISTIC CODE-GENERATION RESULTS

Model	Prompts	Syntax valid	Validity	Compilable	Compilation
Custom BPE	50	0	0.00%	0	0.00%
GPT-2	50	13	26.00%	13	26.00%

Deterministic greedy decoding was used to provide the same 50 held-out functions to both models. There were no generation errors observed, and Custom BPE did not give any syntactically valid or compilable results, compared to GPT-2 giving 13 and 26.00% on those measures respectively. An independent diagnostic analysis found unmatched delimiters, missing indentation and missing strings among Custom-BPE failures, GPT-2 failures were found to have unexpected indentation, missing indented blocks and incomplete expressions. The experimental outcomes do not thus allow a claim of greater code-generation correctness of Custom BPE.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

An additional training run was conducted on a Google Colab instance equipped with a Tesla T4 GPU (15.6 GB VRAM) to assess convergence under higher-memory conditions. This run used a substantially different configuration from the main experiment: 4 Transformer layers (versus 2 in the main experiment), d_model=256 (versus 128), a batch size of 32 (versus 8), 10 training epochs (versus 3), and the Adam optimizer (versus AdamW with gradient clipping). The learning rate (3×10^{-4}), the 15,000-function CodeSearchNet Python training corpus, and the Custom BPE tokenizer vocabulary were kept identical. Because the objective was to verify convergence under higher GPU memory rather than to replicate the full model comparison, only the Custom BPE configuration was trained in this run; GPT-2 was not evaluated. The run yielded a final training perplexity of 49.30 and a validation perplexity of 54.19. Five representative prompts (calculate_median, fibonacci, reverse_string, is_prime, and merge_sort) were submitted to the trained model; all five produced syntactically invalid outputs (0/5 syntax valid), consistent with the 0% result reported in the main experiment on the MX330 hardware. This result provides supplementary evidence of convergence rather than a direct replication of the main experiment.

H. Overall Discussion

A combination of the experiments creates a definite efficiency-quality trade-off. The Python-based BPE compresses the source-code representation, resulting in fewer causal-LM training blocks, smaller vocabulary-dependent model components, shorter training time, and lower peak GPU memory usage. However, GPT-2 configuration can attain far higher next-token prediction performance, and syntax/compilation validity with the same three-epoch training budget. The most reasonable explanation is that the larger vocabulary provides the smaller model with more expressive token units of identifiers, syntax fragments and other patterns whereas the smaller BPE vocabulary allows a smaller representation at the price of a more difficult generation problem. This is an empirical interpretation which should not be extrapolated to other architectures, hardware, training regimes, and tokenizer sizes.

A cross-tokenizer comparison of raw perplexity values should be interpreted with caution: each perplexity is computed over its own vocabulary's token distribution, so the difference in magnitude partly reflects the difference in tokenization granularity rather than a pure difference in language-model capability. A tokenizer-independent metric such as bits per byte (BPB) or normalized byte-level cross-entropy, which measures information content per original byte regardless of tokenization scheme, would enable a fairer comparison and is recommended for future work.

I. Practical Implications and Reproducibility

The operationally meaningful training-time and memory reductions are useful to resource constrained academic settings. Custom-BPE can be trained in the memory of a 2 GB GPU and achieves the reported three-epoch run time of 21.49 minutes. It renders the tokenizer experiments, ablations, and classroom demonstrations possible without the need to have multi-GPU infrastructure. This does not mean that the smaller tokenizer is a substitute to the larger capacity code models but rather that the design of tokenizers can be treated as a controllable lever in the event that the main goal is computational accessibility.

Another lesson that is learnt in the experiment is the reason why efficiency must be reported along with quality. The decrease in the number of tokens can decrease the amount of training blocks and vocabulary-dependent parameters, however, it may also alter the level at which the model learns identifiers and syntax. The GPT-2 vocabulary generated in the current work had a significantly higher perplexity and valid syntax control. That is why, only a report of the 41.94% token reduction or 93.79% reduction of training-time would provide an incomplete picture of the system. The results derive from a single random seed; future replications with multiple seeds are needed to assess result stability.

The artifacts of the experiments contain a trail of reproducibility: The tokenizer file, the frozen configuration, the training logs, the independent test results, the functional-generation results, and comparison summaries were stored in the experiment workspace. Before training, the repository overlap and duplicate-code validation were done, and the test set was not utilized in the construction of the tokenizer or the selection of

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

the checkpoint. This organization assists subsequent reruns with distinct vocabulary sizes, training schedules or model capacity and maintain the identical test regimen.

The present study also separates three questions that are often conflated in compact code-model experiments: whether code is represented compactly, whether the model predicts the next token accurately, and whether generated code is structurally valid. The tokenizer ablation answers the first, perplexity answers the second, and the paired syntax/compilation test addresses the third. Keeping these measurements separate makes the observed trade-off transparent and provides a useful baseline for future improvements.

CONCLUSION AND FUTURE WORK

This study investigated a compact decoder-only Transformer for Python code generation using an 8,000-token domain-specific BPE tokenizer and a 50,258-token GPT-2 tokenizer baseline. Using 15,000 training, 2,000 validation, and 3,000 independent test functions with repository-level separation and duplicate-code checks, the study isolated tokenizer effects under a fixed two-layer, 128-dimensional Transformer. Custom BPE reduced average tokenization length by 41.94%, training blocks by 33.89%, parameter count by 81.37%, training time by 93.79%, and peak test VRAM by 72.40%.

The results simultaneously show that these computational gains were not accompanied by better language-model or generation quality. GPT-2 achieved test PPL 15.68 versus 125.14 for Custom BPE and produced 26% syntactically valid and compilable outputs in the controlled 50-function evaluation versus 0% for Custom BPE. These perplexity values should be interpreted cautiously because they are calculated using different tokenization schemes and are not directly comparable across vocabularies. The principal contribution is therefore an empirical demonstration of the efficiency potential of domain-specific Python tokenization in a compact Transformer, together with a clear measurement of its quality trade-off.

The functional evaluation is diagnostic rather than a substitute for HumanEval or MBPP, and the models were trained for only three epochs with a deliberately small architecture. Future work should evaluate full execution-based benchmarks and pass@k, test additional vocabulary sizes and model capacities under matched batch sizes and training budgets, extend context length, compare decoding strategies, train for longer schedules, and use multiple random seeds. Tokenizer-independent evaluation metrics, such as bits per byte (BPB) or normalized byte-level cross-entropy, should be adopted to enable fair cross-tokenizer language-model comparisons. Hybrid vocabularies that retain Python-specific compression while preserving useful general-purpose tokens are another promising direction. Standardized inference-latency and throughput measurements should also be added in future cross-hardware comparisons.

A further limitation is that the present ablation changes vocabulary size and therefore necessarily changes the embedding and output-layer dimensions. This is intentional because vocabulary-dependent model size is part of the efficiency question, but it means the experiment should be interpreted as a tokenizer-and-vocabulary design comparison rather than as a pure segmentation-only comparison. In addition, the 2 GB MX330 GPU constrains batch size and training duration; results should therefore be viewed as a reproducible resource-constrained baseline rather than a universal estimate of hardware efficiency. The unequal batch sizes (Custom BPE: 8, GPT-2: 4) were dictated by GPU-memory constraints; this disparity affects per-step gradient estimates and renders the training-time comparison an indicative rather than controlled measurement. Reported perplexity values are each computed over their respective token distributions and are therefore not directly scale-comparable across vocabularies; a tokenizer-independent metric such as bits per byte (BPB) or normalized byte-level cross-entropy would provide a fairer cross-tokenizer comparison. All results derive from a single training run with a single random seed, which limits the assessment of result reliability; future replications should use multiple random seeds.

REFERENCES

1. A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in Proc. Adv. Neural Inf. Process. Syst., vol. 30, 2017.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue X, October 2026

2. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI, Tech. Rep., 2019.
3. E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An open large language model for code with multi-turn program synthesis," in Proc. Int. Conf. Learn. Representations (ICLR), 2023.
4. H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," arXiv:1909.09436, 2019.
5. Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in Proc. EMNLP, 2021, pp. 8696–8708.
6. Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X," in Proc. 29th ACM SIGKDD Conf. Knowledge Discovery and Data Mining, 2023, pp. 5673–5684.
7. S. Tipirneni, M. Zhu, and C. K. Reddy, "StructCoder: Structure-aware Transformer for code generation," ACM Trans. Knowl. Discovery Data, vol. 18, no. 3, Art. no. 70, 2024.
8. R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in Proc. 54th Annu. Meeting Assoc. Comput. Linguistics, 2016, pp. 1715–1725.
9. E. J. Hu, Y. Shen, P. Wallis, Z. A. Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," arXiv:2106.09685, 2021.
10. B. Rozière et al., "Code Llama: Open foundation models for code," arXiv:2308.12950, 2023.
11. H. Touvron et al., "LLaMA: Open and efficient foundation language models," arXiv:2302.13971, 2023.
12. R. Li et al., "StarCoder: May the source be with you!," Trans. Mach. Learn. Res., 2023.
13. H. Touvron et al., "Llama 2: Open foundation and fine-tuned chat models," arXiv:2307.09288, 2023.
14. M. Chen et al., "Evaluating large language models trained on code," arXiv:2107.03374, 2021.
15. J. Austin et al., "Program synthesis with large language models," arXiv:2108.07732, 2021.
16. I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in Proc. Int. Conf. Learn. Representations (ICLR), 2019.