Peek2: Regex-free Byte-level Byte-Pair Encoding Pretokenizer for LLM Inference on Edge Devices

Liu Zai and Iraklis Klampanos
University of Glasgow
University Avenue
Glasgow G12 8QQ

Abstract

Pretokenization is a crucial, sequential pass in Byte-level BPE tokenizers, yet little work has been done to optimize it for edge-side inference. Our proposed new implementation, Peek2, serves as a drop-in replacement for cl100k-like pretokenizers used in GPT-3, LLaMa-3, and Qwen-2.5. After breaking down and analyzing the logic of the original cl100k pretokenizer, we introduced a new pretokenization algorithm with linear time complexity and constant, trivial memory usage, suited for edge scenarios. Test results show that it increases microbenchmarking throughput by up to 2.48× and delivers a 1.14× improvement in overall throughput across the entire Byte-level BPE encoding process, depending on the dataset, while providing identical results as the baseline Regex-based tokenizer.

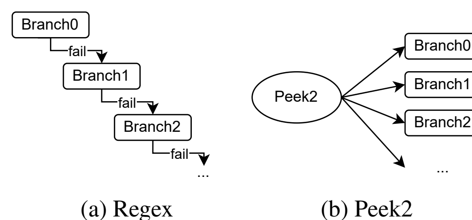
1 Introduction

Byte-level Byte-Pair Encoding (BPE) tokenizers (Sennrich et al., 2016) are widely used in Large Language Models (LLMs) to transform raw text into a sequence of tokens. A pretokenizer is used at the start of the tokenization process, segmenting the original text into shorter fragments to perform pair-merging individually. In practice, the pretokenizer is often implemented using Regular Expressions (Regex).

Recent advances in edge–cloud hybrid LLM inference systems have begun to challenge the long-standing dominance of server-centric architectures (Hao et al., 2024) (She et al., 2025). Improvements in model compression and runtimes now enable portions of LLM workloads to run directly on edge platforms such as desktop PCs, laptops, and embedded devices. We found that these systems perform BPE on low-cost edge devices, including pretokenization. For server-side training and inference, compiling and executing complex Regex is acceptable. However, on edge platforms, the Regex-based pretokenizer may introduce additional overhead that can affect throughput and cold-start time, due to low processing power, limited memory, and the lack of certain instruction sets.

As presented by Hao et al. (2024), the model workload can be routed to the cloud or SLM on the edge after tokenization. With prior work such as BlockBPE (You, 2025), the pair-merging process is also parallelizable, though porting it to edge systems remains necessary. From an overall perspective, the pretokenizing phase, which happens before the pair-merging process, becomes the final piece left sequential and unoptimized. This motivates us to seek an optimization for the widely adopted pretokenizer, cl100k (Brown et al., 2020).

While optimizing the inference pretokenizer, it is necessary to maintain bug-for-bug compatibility with the training pretokenizer. The segment boundaries inserted during the pretokenize phase are carried over to the pair-merging phase. If the pretokenizer is simplified without retraining, a disparity between the inference and training systems would arise, reduce the overall BPE compression rate, and negatively impact downstream performance (Schmidt et al., 2024).



(a) Regex (b) Peek2

Figure 1: Process diagram of different implementations of Cl100k

Existing pretokenizer optimizations are usually server-oriented (You, 2025). Aiming to improve the overall throughput of the tokenization process at the edge, we propose a reimplementation of the cl100k (OpenAI, 2025) pretokenizer, which we refer to as Peek2.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop), pages 123–128
July 2-7, 2026 ©2026 Association for Computational Linguistics

2 Background: Cl100k Analysis

Cl100k (OpenAI, 2025) is the pretokenizer introduced with GPT-3 (Brown et al., 2020). It is then widely adopted by other LLM tokenizers. LLaMa-3 (AI@Meta, 2024) and Qwen-2.5 (Team, 2024) are examples of such models. The pretokenizer segments the script by repeatedly applying the Cl100k regex pattern, shown in Appendix A, inserting a break after each match.

The pretokenizer Regex is formed in an or-of-branches hierarchy. The text is greedily matched against the leftmost branch, and if the match fails, the next branch is tested, as shown in Figure 1a.

2.1 Left Snapping

For convenience, we introduce `Left Snapping`. In this paper, the `|` symbol is used to symbolize a break inserted in text, e.g.

`Lorem| ipsum| dolor| sit| amet|.`

Splits the original script into 6 individual parts. Note that the space is combined with the next word to form a segment.

2.2 Functionality of Each Branch

Branch0 handles contractions. It matches common contractions in common Latin scripts.

Branch1 handles words. Matches any group of letters, usually forming a word in Latin scripts, or a subsentence in East Asian scripts. It `Left Snaps` one character not of Unicode Letter Class, Unicode Number Class, ASCII CR, or ASCII LF.

Branch2 handles numbers. The numbers are grouped into sets of 3 digits.

Branch3 handles punctuations and other characters. Anything that does not match the Unicode Letter Class, Unicode Number Class, or ASCII Whitespaces is grouped together. This cluster will also `Left Snap` one space, and `Right Snap` any count of line folds.

Branch4 handles whitespaces. Whitespaces are grouped together, with the following exceptions: Will break after the last one of a cluster of line folds. Will break before the last whitespace, enabling the next word to left snap that whitespace if it is there.

3 Our proposed pretokenizer: Peek2

We present an optimized pretokenizer implementation, Peek2, primarily focused on improving the branch decision process, as shown in Figure 1b.

3.1 Categorizing the Peeked Character

In the original Regex, there are multiple match elements: some detect the Unicode scalar value, while others classify whether it belongs to a predefined Unicode Class. By applying a process similar to alphabet compression, we get an exhaustive list of categories, which provides just-enough granularity for pretokenization:

- Category0: All other scalars
- Category1: ASCII Space
- Category2: ASCII Single Quote
- Category3: ASCII CR or LF
- Category4: Unicode Letter Class
- Category5: Unicode Whitespace Class
- Category6: Unicode Number Class

To reduce duplicated lookups, we define PeekCategorize in Figure 2, which takes a Unicode scalar value and returns its category. The smaller subsets take precedence, i.e., Category1 will be returned instead of Category5 for the ASCII space `' '` character, and a match group targeting the Unicode Whitespace Class should accept Category1, Category3, and Category5.

```
procedure PEEKCATEGORIZE(scalar)
    if scalar = Space then return 1
    else if scalar = SingleQuote then return 2
    else if scalar = Cr|Lf then return 3
    else
        category ← UNICODECLASSOF(scalar)
        if category = Letter then return 4
        else if category = Whitespace then return 5
        else if category = Number then return 6
        end if
    end if
    return 0
end procedure
```

Figure 2: PeekCategorize, classifying the peeked Unicode scalar value

3.2 Branch Decision

Because only one character can be snapped at a time, the second character's category instantly determines the logic path: it either triggers a `Snap` for the first character or acts as a continuation of the first character's segment.

Based on PeekCategorize (Figure 2), the two branch-deciding temporal steps now have 7×7 input states. This makes it applicable for replacement with a one-time table lookup, as opposed to the original Unicode Scalar input space, which is prohibitively large ($1e5^2$ level). This replacement is inspired by Hopcroft's DFA Minimization Algorithm (Hopcroft, 1971), which iteratively splits

state partitions until no distinguishable states remain. Other temporal steps can be replaced by specific branch functions that trivially look for the next segment stop without any failover.

We have already labelled branches in the original Regex implementation with unique integer indices, from Branch0 to Branch6. For each branch, we translate the first two match sets into the corresponding list of PeekCategorize-returned binary tuples of integers. Each character pair (tuple) is mapped to a specific rule ID (branch index) in Table 1. To maintain the original Regex priority, cells are filled in order; once a high-priority rule (a smaller index) claims a cell, it cannot be overwritten by a later one.

						Cat1	
Cat0	0	1	2	3	4	5	6
0	3	3	3	3	1	3	3
1	3	4	3	4	1	4	4
2	3	3	3	3	0	3	3
3	4	4	4	4	4	4	4
4	1	1	1	1	1	1	1
5	4	4	4	4	1	4	4
6	2	2	2	2	2	2	2

Table 1: Branch Decision Lookup Table

The result in Table 1 maps identically to the previously discussed branch-fallback logic. Pretokenization is achieved in Figure 3 by recursively peeking at the next two scalars and using this table to call the appropriate function to segregate and remove the next segment from the remaining string.

```
procedure PRETOKENIZE(string)
    Cat0 ← PEEKCATEGORIZE(string[0])
    Cat1 ← PEEKCATEGORIZE(string[1])
    string ← LOOKUPANDCALLBRANCH(Cat0, Cat1, string)
    PRETOKENIZE(string)
end procedure
```

Figure 3: Pretokenize, the core pretokenizer algorithm

3.3 Handling of Special Cases

There is an exception of fallbacks happening later: during the handling of Branch0, if the subsequent pattern does not match any common contractions, the single quote should fallback to Category0 and pair with the next letter to fallback to Branch1. We can simply let Branch0 handler function chain invoke Branch1 handler function to handle this exception.

3.4 Demonstration by Example

Given an example input string presegmented:

`Color| :| Red`

Now we demonstrate and compare an Nondeterministic Finite Automaton (NFA)-based Regex engine and our Peek2 algorithm for cutting out the secondary segment `" :"`.

The Regex engine will match Branch0 to Branch6 one by one. While other branches fail at the first character, Branch1 and Branch3 share the `Left Snapping` behavior, so after identifying the first space, the next scalar is first tested if it is of the Unicode Letter class as required by Branch1, which it is not. A stack-like structure is used for failback, and the two characters will be tested again when Branch3 is reached.

Our proposed Peek2 algorithm will first run PeekCategorize on the space and the colon simultaneously. The space is categorized as Category1, and the colon will be Category0. After looking up Table 1, the correct Branch3 is selected.

After the branch decision, the branch function trivially replicates the remaining matching logic performed by the Regex engine until the next mismatch, then the branch selection comes again. Since the branch decision is always identical, the bug-for-bug compatibility is guaranteed.

3.5 Time and Space Complexity

As in the previous example, normally each character is visited exactly once in branch decisions. On rare occasions, such as the late fallback situation described earlier, a small portion of characters might be visited 2 times. So, the time complexity of Peek2 is $O(n)$, where n is the length of the input sequence. The space complexity of Peek2 is strictly $O(1)$ as the lookup table has a fixed size.

For comparison, as the Regex expression gets complex with multiple match groups, it is found (Bille and Thorup, 2010) that the time complexity of NFA-based Regex engines (Thompson, 1968) is $O(n \times k)$ or even $O(n \times m)$, where k is the number of groups and m is the length of the pattern. Deterministic Finite Automaton (DFA)-based Regex engines can achieve linear time complexity (Cox, 2007), but require a complex compiling stage, which often results in high memory usage.

4 Test Results

We implemented the Peek2 pretokenizer with Safe Rust, then integrated this approach into the open-source Hugging Face tokenizers (Moi and Patry, 2025) library on top of version `0.22.3-dev.0`.

The tests are carried out over four datasets: en, cn, code, and math, respectively, collected from The FineWeb Datasets (Penedo et al., 2024), Fineweb-Edu-Chinese (Yu et al., 2025), codesearchnet (Husain et al., 2019), and opc-fineweb-math (Huang et al., 2024).

All of our test cases involve a 10-second warm-up period. The target task is then run over the target dataset repeatedly for 30 seconds, collecting the duration for each sample and the overall throughput. The tests are run on an Intel Core i5-13600KF CPU.

4.1 Microbenchmarking

First, we perform a microbenchmark of the pretokenization stage. No downstream tasks are included in the microbenchmarking.

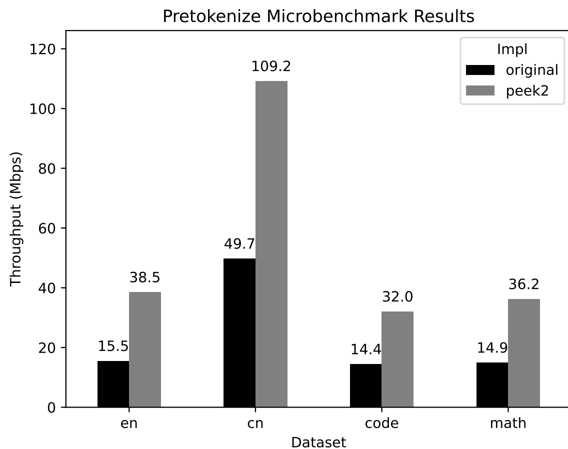


Figure 4: Throughput comparison of the pretokenization stage (errors $< 1\%$)

The throughput has increased sharply, as shown in Figure 4. All datasets have more than doubled their throughput, with the en dataset achieving the largest gain, up to 2.48×. The cn dataset shows the least gain (2.20×), as East Asian scripts have only sub-sentence splitting during this stage, unlike other datasets, which undergo word-level tokenization.

Across all four datasets, the pretokenizer yields the same results as the Regex-based splitter. This validates the bug-for-bug compatibility we aimed for in design.

4.2 End-to-End Benchmarking

Next, we tested and benchmarked the Peek2 pretokenizer versus original Regex pretokenizer across a range of tasks of the complete LLaMa-3 BPE pipeline. The datasets are split into batches of 1000 and distributed to multiple threads to simulate real-world scenarios of long context handling and BPE training.

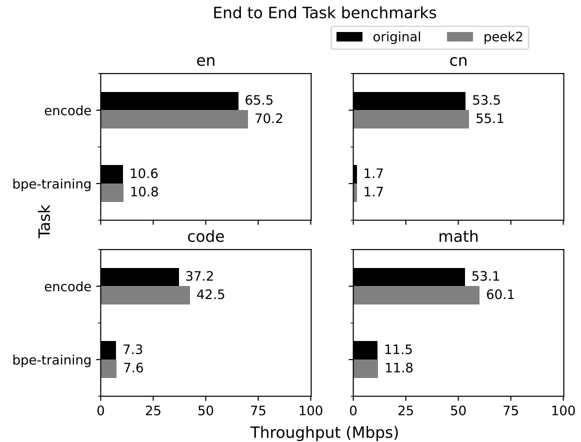


Figure 5: Throughput comparison of different end-to-end tasks (errors $< 1\%$)

All encoding tasks improved their throughput by 1.03× to 1.14×, as shown in Figure 5. However, across all BPE training tasks, gains are much smaller than those in encoding, as downstream tasks dominate the process. code and math have the most significant improvements in their encoding tasks. This is expected, as code and math-heavy data rely more on pretokenization, as there are fewer subword units requiring downstream pair merging.

Impl Dataset	Pretokenize Time (%) original	 peek2
en	11.1	6.9
cn	5.4	3.0
code	22.8	14.0
math	19.8	12.7

Table 2: Pretokenization time consumption proportion, for encoding tasks, comparing the original pretokenization with Peek2

Because the encode task process shares the same input with microbenchmarks, we calculate the time proportion of the pretokenization process relative to the full end-to-end pipeline, as presented in Table 2. While the cn dataset is insensitive to pretokenizer optimizations, in other datasets, especially code, pretokenizer performance plays a significant role.

5 Conclusion

We present Peek2, an edge-optimized, Regex-free implementation of the cl100k pretokenizer, de-

signed around a branch decision table that replaces the branch failback logic in Regex with a one-time lookup.

Test results show that while maintaining identical output, higher throughput is achieved on all tokenizer-related tasks. This work can serve as a drop-in replacement for GPT-3 (Brown et al., 2020), LLaMa-3 (AI@Meta, 2024), and Qwen-2.5 (Team, 2024) pretokenizers for edge or edge-cloud hybrid LLM inference systems, for better tokenization throughput.

Limitations

Peek2 is solely based on the currently widely adopted cl100k-like pretokenizers. Future research might migrate away to other pretokenizers for better performance of the BPE and the LLM. However, this optimization paradigm might be reused for future research.

Currently, the experiments are limited to desktop CPUs. Although we expect the results to be similar, if not better, future work could add comparisons of laptops and embedded devices to cover more platforms for edge inference.

Peek2 is a CPU algorithm. Future work could port it to TPUs and APUs, if available on edge processors, to improve throughput and utilization.

Peek2 was designed to be bug-for-bug compliant with the Regex implementation. However, some of the bugs are almost certainly introduced by Regex's ambiguity. Notably:

`'D|oes| it| work|?'| She| asked.`

Notice how the `'D` is split out due to misinterpretation as a contraction. Evaluation of fixing such bugs might involve retraining or post-training the BPE, or even the model itself, which is out of the scope of this paper.

References

AI@Meta. 2024. Llama 3 model card.

Philip Bille and Mikkel Thorup. 2010. Regular expression matching with multi-strings and intervals. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, page 1297–1308, USA. Society for Industrial and Applied Mathematics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Russ Cox. 2007. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...).

Zixu Hao, Huiqiang Jiang, Shiqi Jiang, Ju Ren, and Ting Cao. 2024. Hybrid slm and llm for edge-cloud collaborative inference. In *Proceedings of the Workshop on Edge and Mobile Foundation Models*, EdgeFM '24, page 36–41, New York, NY, USA. Association for Computing Machinery.

John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, J. H. Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2024. Opencoder: The open cookbook for top-tier code large language models.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *ArXiv*.

Anthony Moi and Nicolas Patry. 2025. HuggingFace's Tokenizers.

OpenAI. 2025. tiktoken.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. 2024. The fineweb datasets: Decanting the web for the finest text data at scale. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Craig W Schmidt, Varshini Reddy, Haoran Zhang, Alec Alameddine, Omri Uzan, Yuval Pinter, and Chris Tanner. 2024. Tokenization is more than compression. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 678–702, Miami, Florida, USA. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Jianshu She, Wenhao Zheng, Zhengzhong Liu, Hongyi Wang, Eric P. Xing, Huaxiu Yao, and Qirong Ho. 2025. Token level routing inference system for edge devices. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 159–166, Vienna, Austria. Association for Computational Linguistics.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422.

Amos You. 2025. Blockbpe: Parallel bpe tokenization. *Preprint*, arXiv:2507.11941.

Yijiong Yu, Ziyun Dai, Zekun Wang, Wei Wang, Ran Chen, and Ji Pei. 2025. Opencsg chinese corpus: A series of high-quality chinese datasets for llm training. *Preprint*, arXiv:2501.08197.

Appendix

A Cl100k Regex

A.1 Complete Regex

```
'(?i:[sdmt]|ll|ve|re)|
[^\r\n\p{L}\p{N}]?+\p{L}++|
\p{N}{1,3}+|
 ?[^\s\p{L}\p{N}]++[\r\n]*+|
\s++$|\s*[\r\n]|\s+(?!\S)|\s~
```

The linefolds are added for the ease of interpretation.

A.2 Verbose Branches

To improve the clarity of this complex Regex, we also provide each Branch in Python's VERBOSE Regex format, with additional whitespaces added for visual guidance. The original whitespaces are escaped with \.

A.2.1 Branch 1

```
'
(?:
    [sdmt]
    | ll
    | ve
    | re
)
```

A.2.2 Branch 2

```
[^\r\n\p{L}\p{N}]?+
\p{L}++
```

A.2.3 Branch 3

```
\p{N}{1,3}+
```

A.2.4 Branch 4

```
\ ?
[^\s\p{L}\p{N}]++
[\r\n]*+
```

A.2.5 Branch 5

```
  \s++$
| \s*[\r\n]
| \s+(?!\S)
| \s~
```

B Code and Data Availability

B.1 GitHub Link

https://github.com/omegacoleman/tokenizers_peek2

B.2 Archival DOI

https://doi.org/10.5281/zenodo.18459917