Tokenization

Vocabulary Design and Its Impact on Language Model Behavior

February 2026 • Technical Report

Table of Contents

1. Introduction

Tokenization is the first and arguably most foundational step in the language model pipeline. Before any neural network computation occurs, raw text must be converted into a sequence of discrete tokens from a fixed vocabulary. The choice of tokenization algorithm, vocabulary size, and training data directly affects nearly every aspect of model performance: the effective context length in characters, multilingual capability, arithmetic and reasoning ability, code generation quality, and even the model's susceptibility to certain types of adversarial attacks.

Despite its fundamental importance, tokenization often receives less attention than model architecture or training methodology. This report examines the major tokenization approaches used in modern LLMs, the design decisions involved in building a tokenizer, and the downstream impacts of these choices on model behavior.

2. Tokenization Algorithms

2.1 Byte Pair Encoding (BPE)

Byte Pair Encoding is the most widely used tokenization algorithm in modern LLMs. BPE starts with a base vocabulary of individual characters (or bytes) and iteratively merges the most frequent adjacent pair of tokens into a new token, repeating until the desired vocabulary size is reached. The result is a vocabulary that includes common words as single tokens, frequent subwords as tokens, and rare words decomposed into multiple subword tokens. GPT-2, GPT-3, GPT-4, and Llama all use variants of BPE. The algorithm is deterministic during encoding (applying merges in priority order), fast, and produces a good balance between vocabulary efficiency and coverage.

2.2 SentencePiece and Unigram

SentencePiece is a language-independent tokenization framework that treats the input as a raw byte stream, eliminating the need for language-specific pre-processing or whitespace handling. It supports both BPE and the Unigram model. The Unigram algorithm starts with a large candidate vocabulary and iteratively prunes tokens that contribute least to the overall likelihood of the training corpus, using a unigram language model to evaluate each token's contribution. Unigram tokenization can produce slightly different segmentations for the same input depending on the probabilistic model, which can be used for data augmentation through subword regularization.

2.3 Byte-Level Tokenization

Byte-level tokenization operates on raw bytes (values 0 to 255) rather than Unicode characters, guaranteeing that any input can be encoded without unknown tokens. GPT-2 introduced byte-level BPE, which applies the BPE algorithm to byte sequences rather than character sequences. This approach handles any language, special characters, code, and binary-like content without vocabulary gaps. The trade-off is that non-ASCII

characters (common in languages like Chinese, Japanese, and Korean) require multiple bytes per character, reducing the effective context length for these languages compared to character-level approaches.

2.4 Byte Fallback and Hybrid Approaches

Some tokenizers, like those used in Llama 3 and Gemma, combine character or subword-level tokenization with byte-level fallback. Common text is tokenized efficiently as subwords, while rare characters or byte sequences that are not in the vocabulary fall back to individual byte tokens. This hybrid approach provides the vocabulary efficiency of subword tokenization for common text while maintaining the universal coverage of byte-level encoding.

3. Vocabulary Design Decisions

3.1 Vocabulary Size

Vocabulary size is a critical design parameter that involves several trade-offs. Larger vocabularies (100K to 200K tokens) encode text more compactly, requiring fewer tokens per document and effectively extending the context window. However, they require larger embedding matrices, increase the memory footprint, and may dilute training signal for rare tokens. Smaller vocabularies (32K to 50K tokens) are more memory-efficient but produce longer token sequences, reducing the effective context length. Modern models have converged on vocabulary sizes in the 32K to 128K range, with a trend toward larger vocabularies as model sizes increase.

3.2 Training Data Composition

The corpus used to train the tokenizer determines which tokens are created and their relative efficiency for different text types. A tokenizer trained primarily on English text will produce verbose tokenizations for other languages, requiring more tokens per word and consuming more context window. Code-heavy training data creates tokens for common programming patterns. Careful curation of tokenizer training data to balance languages, domains, and code ensures more equitable performance across use cases. Llama 3's tokenizer, for example, was trained on a more multilingual and code-inclusive corpus than its predecessor, significantly improving tokenization efficiency for non-English languages.

3.3 Special Tokens

Special tokens serve as control signals that structure the model's input and behavior. Common special tokens include beginning-of-sequence and end-of-sequence markers, separator tokens for multi-segment inputs, tool-use and function-calling delimiters, and role markers in chat formats (system, user, assistant). The design of special tokens affects how the model handles conversation structure, tool integration, and generation

termination. Poor special token design can lead to subtle bugs where the model fails to recognize conversational boundaries or terminates generation prematurely.

4. Impact on Model Behavior

4.1 Multilingual Performance

Tokenization quality varies dramatically across languages and directly impacts multilingual model performance. For languages well-represented in the tokenizer training data, common words are encoded as single tokens, providing efficient compression. For underrepresented languages, the same semantic content may require 3 to 10 times more tokens, consuming proportionally more context window and computation. This tokenization inequality creates a fundamental disparity in effective model capability across languages, even when the underlying model weights are shared. Addressing this disparity through balanced tokenizer training is one of the most impactful steps toward equitable multilingual models.

4.2 Arithmetic and Numerical Reasoning

How numbers are tokenized significantly affects a model's ability to perform arithmetic. If numbers are split into arbitrary chunks (e.g., "12345" becoming "123" and "45"), the model must learn digit alignment across token boundaries, which is much harder than operating on individual digits. Some tokenizers encode each digit as a separate token, simplifying arithmetic but increasing sequence length for large numbers. Others create tokens for common number patterns. Research has shown that tokenizer design accounts for a meaningful fraction of the variance in LLM arithmetic performance.

4.3 Code Generation

Code has different statistical properties than natural language: significant whitespace (indentation), frequent special characters, highly repetitive patterns (variable names, syntax structures), and sensitivity to exact character sequences. Tokenizers trained with substantial code in their training corpus develop tokens for common code patterns like indentation levels, programming keywords, and frequent API calls. This improves both the efficiency (fewer tokens per line of code) and the quality (the model can reason about code in larger, semantically meaningful chunks) of code generation.

5. Tokenizer Comparison

Tokenizer	Model Family	Vocab Size	Algorithm	Notable Features
cl100k_base	GPT-4, GPT-3.5	100,256	Byte-level BPE	Good multilingual, code-aware
o200k_base	GPT-4o	200,000	Byte-level BPE	Improved multilingual

				efficiency
SentencePiece BPE	Llama 2	32,000	BPE + byte fallback	Compact vocabulary
SentencePiece BPE	Llama 3	128,256	BPE + byte fallback	4x larger, better multilingual
SentencePiece BPE	Gemma	256,000	BPE + byte fallback	Very large vocabulary
tiktoken-compatible	Mistral/Mixtral	32,768	Byte-level BPE	Efficient for European languages

6. Tokenization Pitfalls and Edge Cases

Tokenization introduces several subtle failure modes that practitioners should be aware of. Token boundary effects can cause models to behave differently depending on how a word is segmented, leading to inconsistencies in string processing tasks. Prompt injection attacks sometimes exploit tokenization by using unusual Unicode characters or whitespace that tokenizes differently than expected. Trailing whitespace, invisible characters, and Unicode normalization differences can change tokenization in surprising ways. Token counting for context window management must use the actual tokenizer rather than word count approximations, as the ratio of tokens to words varies significantly across languages and content types.

7. Future Directions

Several research directions aim to improve or move beyond current tokenization approaches. Token-free models that operate directly on bytes or characters eliminate tokenization artifacts entirely, though they face challenges with sequence length and computational efficiency. Dynamic vocabularies that adapt to the input domain could improve efficiency for specialized applications. Improved multilingual tokenization through balanced training and larger vocabularies continues to reduce cross-language performance disparities. The Megabyte architecture and similar approaches that process bytes at a coarse level and then refine at a fine level offer a middle ground between byte-level and subword approaches.

8. Conclusion

Tokenization is a deceptively important component of the LLM pipeline whose design decisions propagate through every aspect of model behavior. From vocabulary size and algorithm choice to training data composition and special token design, each decision creates trade-offs that affect model performance across languages, domains, and tasks. As the field moves toward larger vocabularies, better multilingual coverage, and potentially tokenizer-free architectures, understanding these trade-offs remains

essential for anyone building or deploying language models. The often-overlooked tokenizer deserves careful attention proportional to its outsized impact on model capability.