



Tokenisation

LLM folder | How text becomes token ids: BPE, WordPiece, Unigram and SentencePiece, byte-level BPE, vocabulary size, tokens per language, special tokens, chat templates and the quirks they cause

#LLM #Tokenizer #BPE #ChatTemplate

Why tokens, in plain words

Tokenisation

In one sentence
●○○○

The model cannot read letters: a tokenizer cuts text into frequent chunks and gives each chunk a number.

Practitioner
●●○○

Common words are one token, rare words are split into pieces, any byte can still be encoded.

- English: about 4 characters or 0.75 words per token (OpenAI rule of thumb)
- you pay, wait and fill the context per token

Expert
●●●○

A sub-word vocabulary (BPE, WordPiece, Unigram) is learned once on a corpus sample, before pretraining, and frozen with the model.

- changing the tokenizer means retraining or re-aligning the embeddings
- vocabulary size sets the embedding and output matrix sizes

Under the hood
●●●●

Token boundaries leak into behaviour: spelling, counting letters, arithmetic, leading spaces, other languages and prompt endings all depend on them.

- the same word with and without a leading space is two different tokens
- special tokens and chat templates must match training exactly

Analogy

Like a shorthand stenographer: frequent words get one squiggle, rare names are spelled out syllable by syllable.

Tokenisation at four depths. Level 1 is the whole idea; level 4 is what bites in production

Prerequisites

Know what an integer id and a lookup table are, and that the model outputs one probability per vocabulary entry (sheet 00). No maths beyond counting is needed here.

The algorithms

	BPE	Byte-level BPE	WordPiece	Unigram (SentencePiece)
Base units	characters	256 bytes	characters	large seed set
Training rule	merge most frequent pair	merge most frequent pair	merge pair with best likelihood gain	prune least useful pieces
Unknown characters	unknown token unless byte fallback	never: every byte exists	[UNK] token	byte fallback optional
Marks word pieces	end-of-word or space marker	space is part of the token (Ġ)	## on continuations	? marks a preceding space
Typical users	original GPT, Llama 2 (SentencePiece BPE)	GPT-2 to GPT-4o, Llama 3, Qwen	BERT, DistilBERT, ELECTRA	T5, ALBERT, XLNet, mT5

Four tokenizer families. Byte-level BPE is the default for decoder LLMs in 2026; Unigram via SentencePiece is common in multilingual and Google models

Pre-tokenisation — A regex split before BPE (on spaces, punctuation, digit groups) so merges never cross word boundaries; GPT-4's tokenizer splits digits into groups of at most 3.

Subword regularisation — Sampling different valid segmentations of the same text during training (Unigram, BPE-dropout) to make the model robust to splits.

Tadata | tadata.fr | contact@tadata.codes

PROBLEM

Corpus: low ×5, lower ×2, newest ×6, widest ×3. Learn 7 merges, then tokenize the unseen word 'lowest'.

GIVEN

- Start: every word split into characters
- Count each adjacent pair, weighted by word frequency
- Merge the top pair everywhere, recount, repeat

1

Count pairs

e s = 6+3 = 9, s t = 9, w e = 8, l o = 7, o w = 7

Why: newest and widest both contain 'est'

2

Merges 1-2

e s → es (9); es t → est (9)

Why: 'est' now a token in newest and widest

3

Merges 3-4

l o → lo (7); lo w → low (7)

Why: low ×5 plus lower ×2

4

Merges 5-7

n e → ne; ne w → new; new est → newest (6 each)

Why: the frequent word becomes one token

5

Encode 'lowest'

l o w e s t → l o w e s t → l o w e s t → l o w e s t → low est

Why: apply learned merges in their learned order

ANSWER

'lowest' = [low, est]: 2 tokens, although the word never appeared in the corpus

CHECK

✓ 'newest' = 1 token, 'widest' = [w, i, d, est] = 4 tokens: frequent words get short encodings

Learning BPE merges by hand on Sennrich's toy corpus. Pair counts are summed over word frequencies; ties go to the pair seen first

Text: "lowest newest widest"

start: characters

l o w e s t n e w e s t w i d e s t

1. e + s → es ×3

2. es + t → est ×3

3. l + o → lo ×1

4. lo + w → low ×1

5. n + e → ne ×1

6. ne + w → new ×1

7. new + est → newest ×1

Tokens and ids

low est newest w i d est

12 10 15 8 2 0 10

20 characters · 3 words · 7 tokens · 2.86 characters per token · 2.33 tokens per word

Characters per token = characters ÷ tokens (cost and context limits count tokens, not words)

where characters = every letter, digit, punctuation mark and space of the text; merge a + b → ab = every adjacent pair a, b inside a word becomes one new token, rules applied in their learned order; id = position in the vocabulary: sorted characters, then one id per merge

The same seven merges applied by the renderer: characters, token chips and counts for an unseen word

Vocabulary size: the trade-off		
Tokenizer / model	Algorithm	Vocabulary
BERT-base uncased	WordPiece	30,522
Llama 2, Mistral 7B	SentencePiece BPE + byte fallback	32,000
GPT-2, GPT-3	byte-level BPE	50,257
GPT-4 (cl100k_base)	byte-level BPE	about 100k
Llama 3 family	byte-level BPE (tiktoken-based)	128,256
GPT-4o (o200k_base)	byte-level BPE	about 200k
Gemma family	SentencePiece	about 256k
Vocabulary sizes of well-documented tokenizers (published configs)		

Quantity	Formula	Reading
Embedding matrix	$V \times d$	one row of d numbers per token
Output (unembedding) matrix	$d \times V$	same size; shared with the embedding when weights are tied
Tokens for a text	$\approx C/\bar{c}$	bigger vocab → larger mean chunk, fewer tokens

where V = vocabulary size (tokens); d = model (hidden) dimension (numbers per vector); C = length of the text (characters); $\bar{c}$ = average characters per token for that text (characters/token)

What the vocabulary costs in parameters. Worked: Llama 3 8B has $V = 128,256$ and $d = 4,096$, so one matrix holds $128,256 \times 4,096 \approx 525$ M parameters, two (untied) ≈ 1.05 B, about 13 % of the 8 B model

Bigger vocabulary	Smaller vocabulary
fewer tokens per text: cheaper, longer effective context	small matrices, every token well trained
better coverage of other languages and code	longer sequences: more compute, less text per window
larger embedding and output matrices	words split into many pieces
rare tokens seen too little in training	fine for small or single-language models

Languages, numbers and other quirks		
Quirk	Cause	Status in 2026
Miscounting the r's in 'strawberry'	the model sees 2-3 tokens, not letters	partial
Weak multi-digit arithmetic in older models	digits merged in arbitrary chunks (GPT-2); Llama 1 split every digit, GPT-4 groups of ≤ 3	partial
'Hello' ≠ 'Hello'	leading space is part of the token in byte-level BPE	yes
Odd output after a prompt ending in a space	training text rarely has a lone space token there	yes

Glitch tokens (' SolidGold-Magikarp', GPT-2/3 era)	token in the vocabulary, almost never in training data	no
Code indentation costs many tokens	runs of spaces are split unless whitespace tokens exist	partial

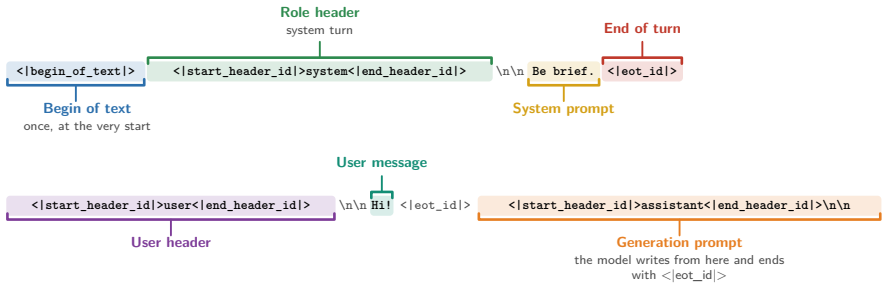
Behaviours explained by token boundaries

💡 Count tokens, do not guess

Use the provider's own tokenizer or token-count endpoint (e.g. tiktoken for OpenAI models, the transformers tokenizer for open models, the Anthropic token-counting API for Claude).
Counts for the same text differ by model family; the 4-characters rule is only for English prose.

```
Encode and decode with Hugging Face transformers
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("gpt2")
ids = tok.encode(" lowest newest")
print(ids, len(ids))
print(tok.convert_ids_to_tokens(ids)) # Ġ = leading space
print(tok.decode(ids))                # round trip
```

📄 Special tokens and chat templates



The Llama 3 instruct chat template, one turn each. Every model family has its own; a wrong template silently degrades answers

Token	Family	Role
[CLS] [SEP] [MASK] [PAD]	BERT	sequence summary, separator, masked slot, padding
< endoftext >	GPT-2 to GPT-4	document boundary / stop
<s> </s>	Llama 2, Mistral	begin / end of sequence
< im_start > < im_end >	ChatML (Qwen and others)	open / close one role turn
< eot_id >	Llama 3	end of a turn; generation stops here

Special tokens you will meet

⚠️ Template mismatches are silent

Fine-tuning or serving with another family's template, a missing generation prompt, or a doubled begin-of-text token gives worse, rambling or never-ending answers without any error. Always build prompts with `tokenizer.apply_chat_template(messages, add_generation_prompt=True)` or the server's chat endpoint.

🗨️ Interview question

"Why can't you just add 1,000 domain words to a trained model's vocabulary?" You can, but each new token gets a fresh embedding and output row that the model never learned: it needs continued training (or initialisation from the mean of its old sub-tokens) before it helps.

★ Myths and recap

MYTH

❌ A token is a word

FACT

✅ A token is any frequent chunk: word, piece, space-plus-word, byte
English averages about 0.75 words per token; code, numbers and other languages differ a lot.

❌ Every model counts tokens the same way

✅ Each family has its own tokenizer and counts
The same prompt can differ by tens of percent in tokens, hence in price and context use, across providers.

❌ Tokenizers learn during pretraining

✅ The tokenizer is trained before and then frozen
Its merges come from statistics on a corpus sample, not from gradient descent.

❌ Bigger vocabulary is always better

✅ It trades shorter sequences for bigger matrices and rare, undertrained tokens
Small models often use 32k-50k; large multilingual models 128k-256k.

Beliefs about tokens that cost money or accuracy

Key takeaways

01

Text → bytes → merges → ids
byte-level BPE never needs an unknown token

02

Learned once, frozen
merges from corpus statistics, applied in order

03

Vocab size is a trade-off
 $V \times d$ parameters vs sequence length

04

Tokens are the bill
price, latency and context all count tokens; languages differ

05

Templates must match
special tokens are part of the trained behaviour

06

Quirks come from boundaries
letters, digits, spaces, glitch tokens

Q1 What does BPE merge at each training step?

A The most frequent adjacent pair of symbols in the corpus (weighted by word counts), creating one new token; repeated until the vocabulary size is reached.

Q2 Why does byte-level BPE never output an unknown token?

A Its base vocabulary is all 256 byte values, so any UTF-8 text decomposes into known tokens.

Q3 WordPiece vs BPE in one line?

A WordPiece merges the pair with the best likelihood gain (frequency of the pair relative to its parts), BPE the most frequent pair; WordPiece marks continuations with `##`.

Q4 How does a Unigram tokenizer build its vocabulary?

A It starts from a large candidate set and repeatedly removes the pieces whose loss least reduces corpus likelihood; encoding picks the most probable segmentation.

Q5 Embedding parameters for $V = 128,256$ and $d = 4,096$?

A $V \times d \approx 525$ million per matrix; about 1.05 billion for untied input and output matrices.

Q6 Rule of thumb for English tokens?

A About 4 characters or 0.75 words per token, so 1,000 tokens $\approx$ 750 words; not valid for code or other languages.

Q7 Why do some languages cost more per sentence?

A Higher fertility: the tokenizer was trained mostly on English, so their words split into more tokens.

Q8 What is a chat template and why must it match?

A The exact string with special role and end-of-turn tokens a chat model was trained on; a different format silently degrades answers.

Q9 What is a glitch token?

A A token in the vocabulary that was almost absent from training data, so its embedding is untrained and the model misbehaves on it (e.g. ' SolidGoldMagikarp').