

Byte Pair Encoding (BPE) Tokenization

The Input Pipeline That Determines Everything Downstream

Byte Pair Encoding is the subword tokenization algorithm used by GPT-2, GPT-4, LLaMA, Mistral, and virtually every modern LLM. Starting from a base vocabulary of 256 byte tokens, BPE iteratively merges the most frequent adjacent pairs to build a vocabulary of subword units.

This demo covers:

1. Training process visualization: merge rules learned step by step
2. Compression ratio analysis: characters per token vs vocab size
 - 3. Tokenization examples: English, code, Unicode, emoji
4. Vocabulary size tradeoff: sequence length, memory, compute
5. Byte fallback & special tokens: handling any UTF-8 input
6. Inference impact: KV cache and attention cost differences

Training corpus: 24 documents

Random seed: 42

Number of visualizations: 6

Generated by `demo.py`

Examples: 6

Mathematical Foundation

BPE Training Algorithm

Given corpus C and target vocabulary size $|V|$:

$$V_0 = \{b_i : i \in [0, 255]\} \text{ (base byte vocabulary)}$$

For $k = 1, \dots, |V| - 256$:

$$(a^*, b^*) = \arg \max_{(a, b)} \sum_{w \in C} \text{count}((a, b) \text{ in } w) \cdot \text{freq}(w)$$

$V_k = V_{k-1} \cup \{a^* \| b^*\}$, replace all (a^*, b^*) with $a^* \| b^*$ in C

Encoding (Applying Merge Rules)

Given text t and ordered merge rules $M = [(a_1, b_1), \dots, (a_m, b_m)]$:

$\text{tokens}_0 = [c_1, c_2, \dots, c_n]$ where c_i are byte tokens of t

For each $(a_k, b_k) \in M$: replace all adjacent (a_k, b_k) with $a_k \| b_k$

IDs = [vocab[tok] for tok in final tokens]

Inference Cost Analysis

Let n = sequence length (tokens), L = layers, h = heads, d_h = head dim:

$$\text{Attention FLOPs} = 2 \cdot L \cdot h \cdot n^2 \cdot d_h \Rightarrow O(n^2)$$

$$\text{KV cache memory} = 2 \cdot L \cdot h \cdot n \cdot d_h \cdot b \text{ bytes} \Rightarrow O(n)$$

$$\text{Embedding memory} = |V| \cdot d_{\text{model}} \cdot b \text{ bytes} \Rightarrow O(|V|)$$

Compression Ratio

$$\text{CPT} = \frac{|text|}{|\text{encode(text)}|} \text{ (characters per token, higher = better)}$$

$$\text{CR} = \frac{|text|_{\text{bytes}}}{|\text{encode(text)}|} \text{ (compression ratio)}$$

Vocabulary Size Tradeoff

Smaller $|V|$: longer $n \Rightarrow$ attention $O(n^2) \uparrow$, KV cache $O(n) \uparrow$, embedding $O(|V|) \downarrow$

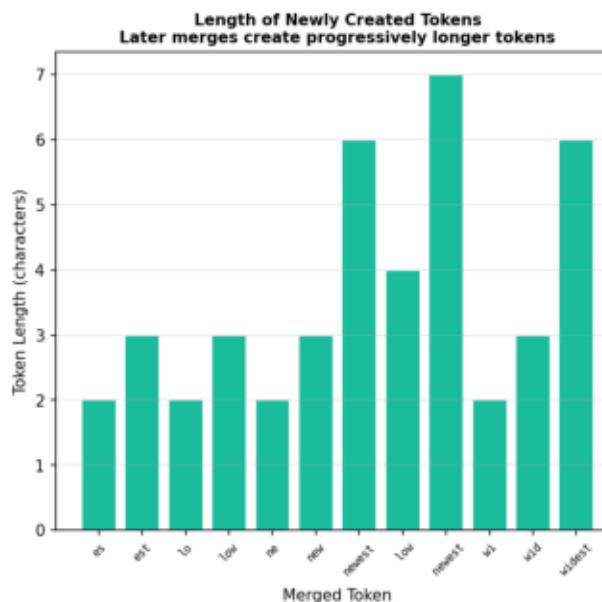
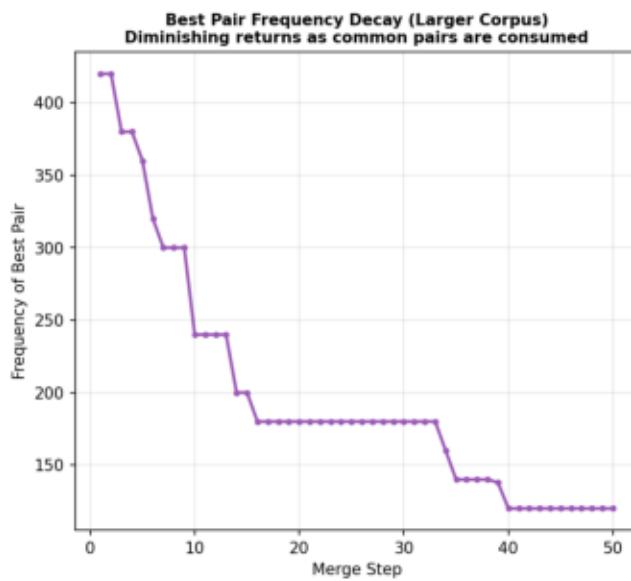
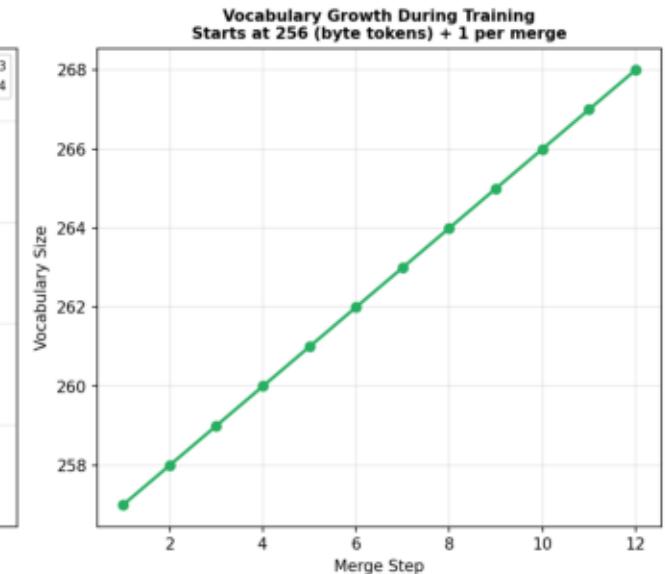
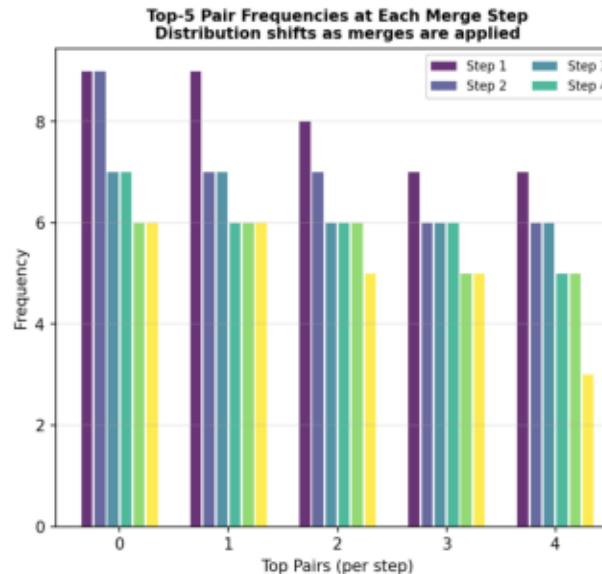
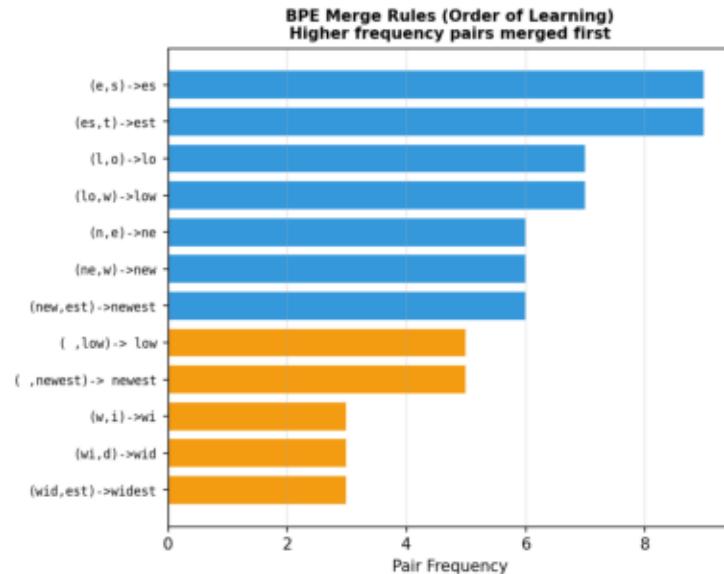
Larger $|V|$: shorter $n \Rightarrow$ attention $O(n^2) \downarrow$, KV cache $O(n) \downarrow$, embedding $O(|V|) \uparrow$

Summary of Findings

1. Training Process: BPE greedily merges the most frequent adjacent byte pairs. The best-pair frequency decays as obvious patterns are consumed. Later merges create progressively longer tokens by combining previously merged tokens.
2. Compression: Characters per token (CPT) improves with vocabulary size, with diminishing returns. English text trained on English corpus achieves best CPT. Random/unseen text compresses poorly (~1.0 CPT, byte-level fallback).
3. Tokenization: English text compresses well (high CPT). Code is moderate. CJK, Arabic, and emoji fall back to byte-level encoding (multiple tokens per character) since the training corpus is English-heavy.
4. Vocab Size Tradeoff: Larger vocabulary produces shorter sequences, reducing attention compute ($O(n^2)$) and KV cache memory ($O(n)$), but increases embedding table size ($O(|V|)$). Sweet spot: 32K-100K for production LLMs.
5. Byte Fallback: The 256 base byte tokens guarantee encoding of ANY UTF-8 input. No `<unk>` token needed. Special tokens (e.g., `<|endoftext|>`) are protected from BPE splitting and always encode to a single token ID. Roundtrip correctness is guaranteed: `decode(encode(text)) == text` for all valid UTF-8.
6. Inference Impact: Better tokenization directly reduces inference cost. Shorter sequences mean less attention compute, smaller KV cache, and more concurrent requests per GPU. At batch scale, the memory savings from even modest CPT improvements translate to significant GPU memory freed for serving.

Example 1: BPE Training Process

BPE Training: Learning Merge Rules from Corpus



BPE TRAINING PROCESS

Algorithm:

1. Start with 256 byte tokens
2. Count all adjacent pairs
3. Merge most frequent pair
4. Update corpus representation
5. Repeat until target vocab size

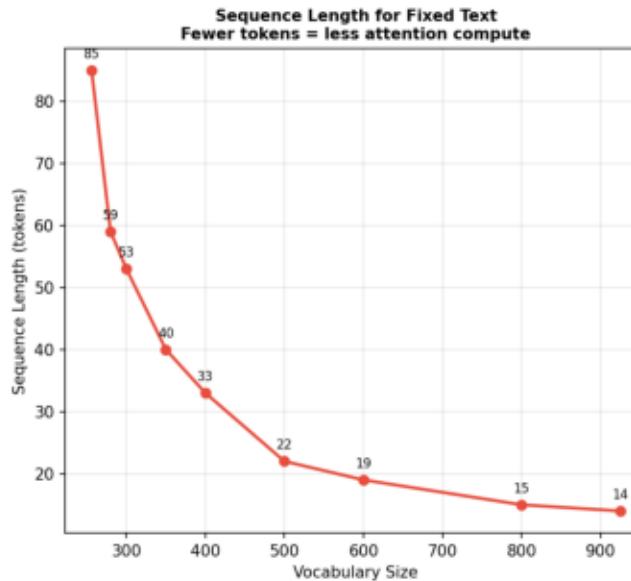
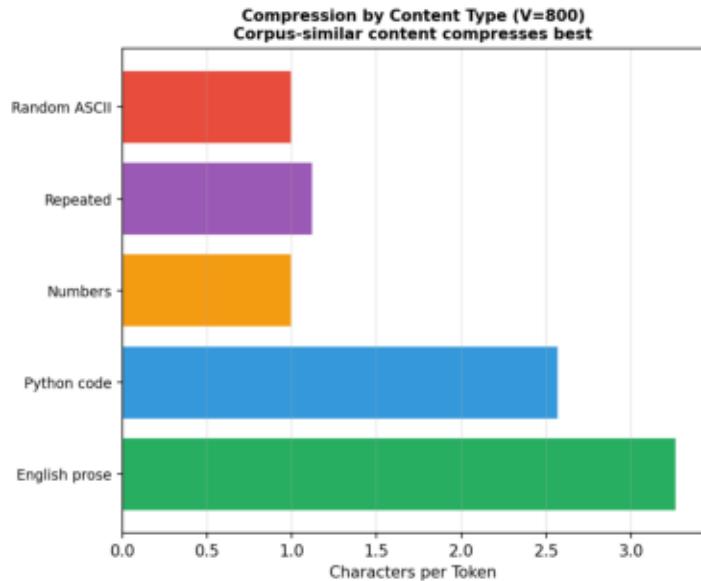
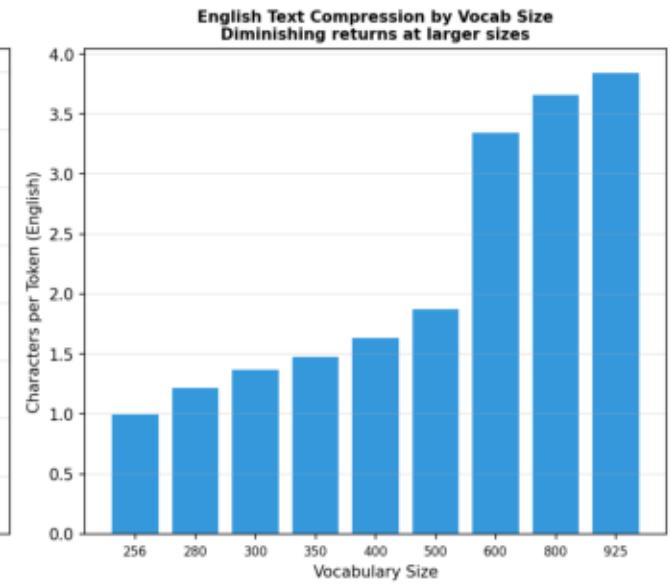
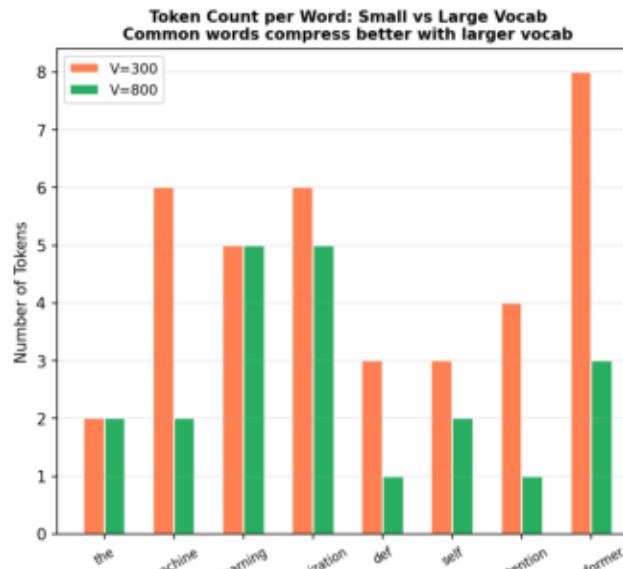
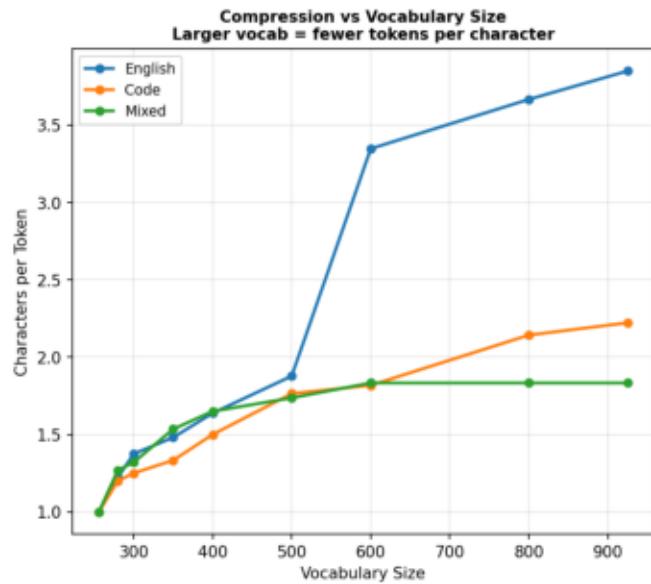
Key observations:

- Frequent pairs merged first (spaces, common bigrams)
- Best-pair frequency decays as obvious patterns merge
- Token length grows over time (merging previously merged)
- Training is deterministic: same corpus = same merges

Small corpus: 4 docs
Merges shown: 12

Example 2: Compression Ratio Analysis

BPE Compression Ratio Analysis: Vocabulary Size vs Efficiency



COMPRESSION ANALYSIS

Characters per Token (CPT):
Higher = better compression
= fewer tokens for same text
= less attention compute

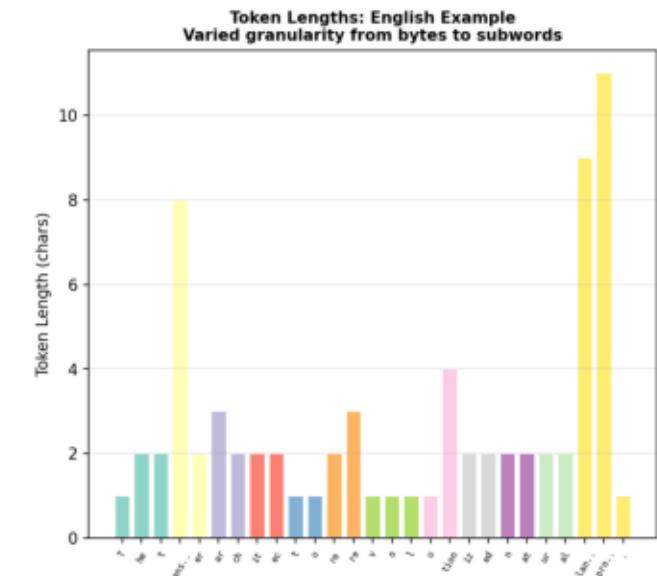
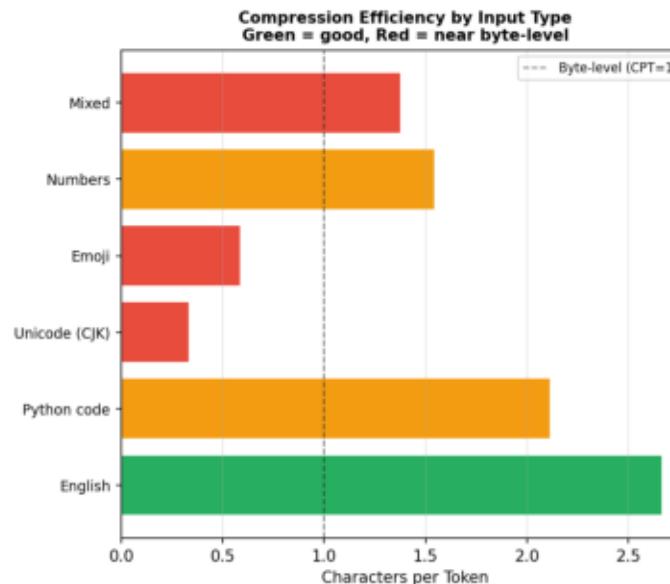
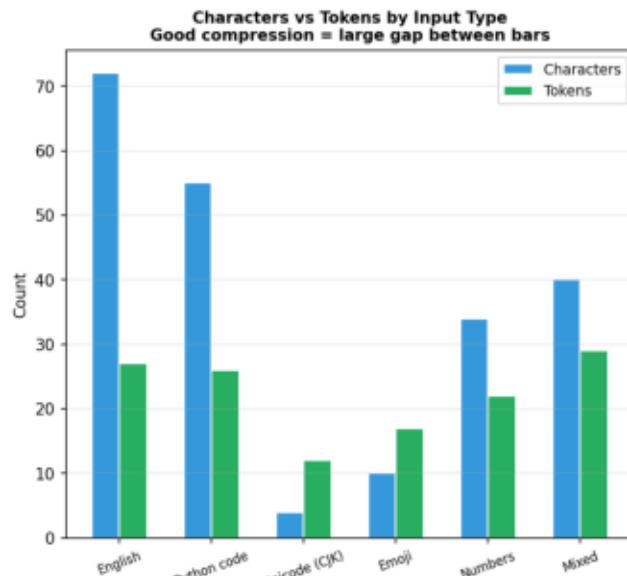
Key findings:
V=256 (bytes only): ~1.0 CPT
V=300 English: 1.3B CPT
V=800 English: 3.67 CPT
V=1000 English: 3.85 CPT

Content-type effects:
- Trained-on content: best CPT
- Code: moderate (training has some code patterns)
- Random text: ~1.0 CPT (no learnable patterns)

Diminishing returns: doubling vocab from 500->1000 gives less gain than 256->500.

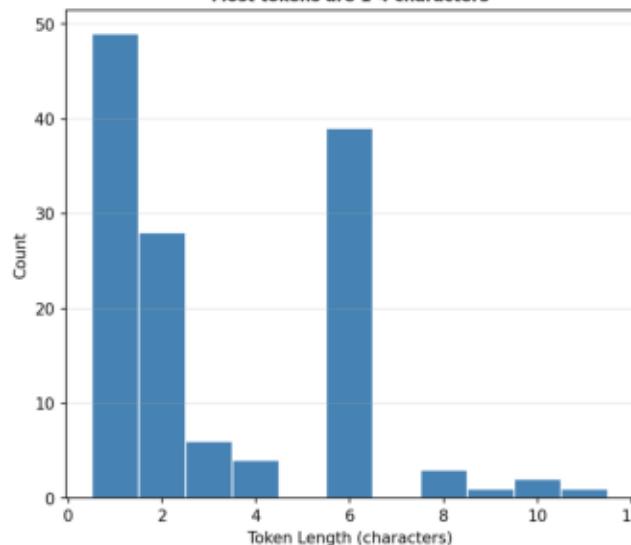
Example 3: Tokenization Examples

BPE Tokenization: How Different Text Types Get Tokenized



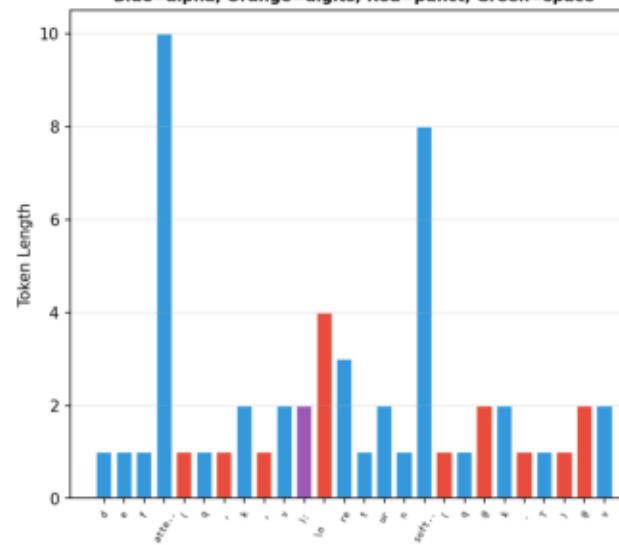
Distribution of Token Lengths (All Examples)

Most tokens are 1-4 characters



Code Tokenization Detail

Blue=alpha, Orange=digits, Red=punct, Green=space



TOKENTZATTON EXAMPLES

Vocab size: 600

English

27 tokens, CPT=2.67

Python code:

26 tokens, CPUT=2.12
Unicode (UTF8):

12 tokens, CPT=0.33

Emoji:

17 tokens, CPT=0.59

22 tokens - CPT=1.55

Mixed:

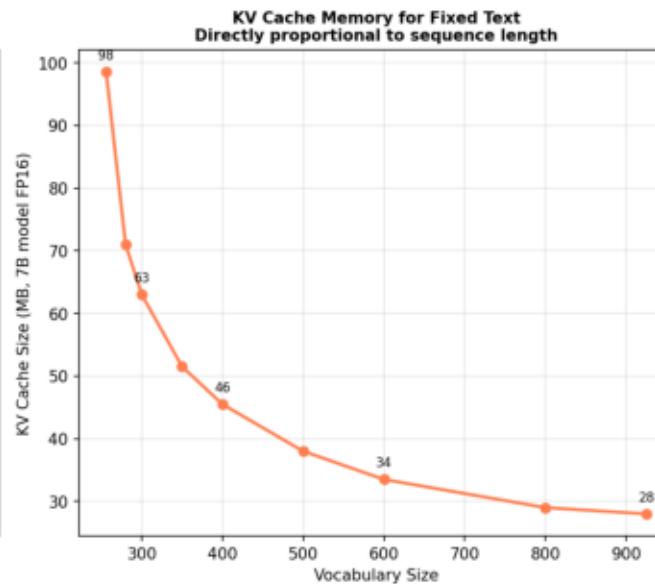
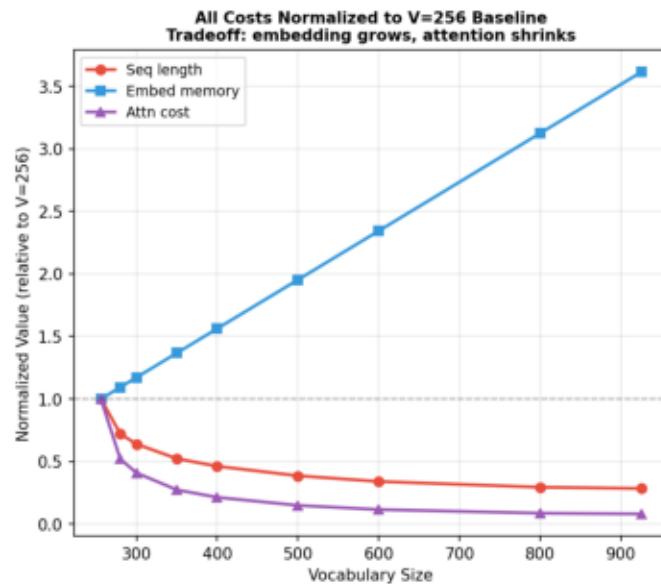
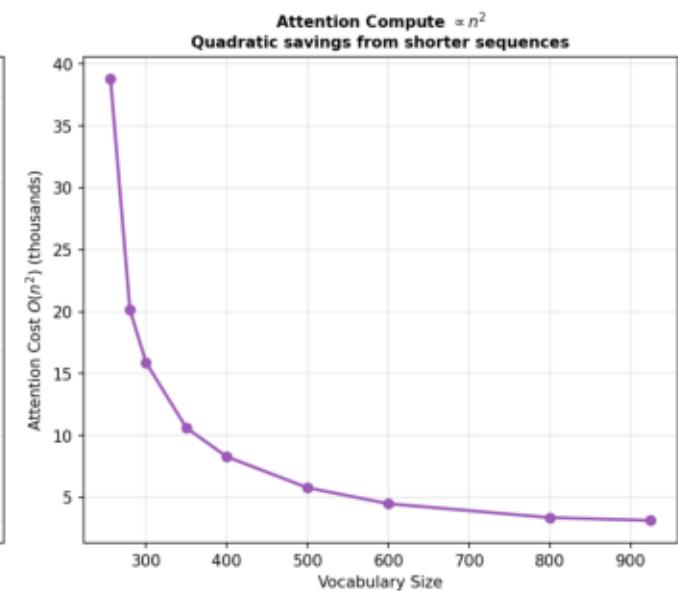
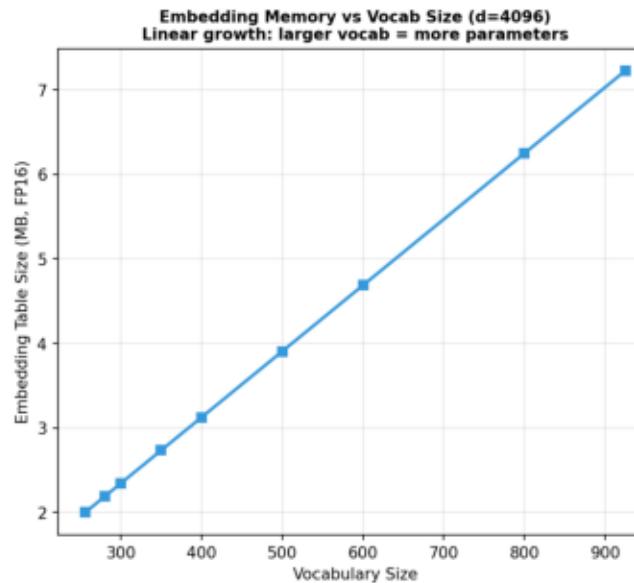
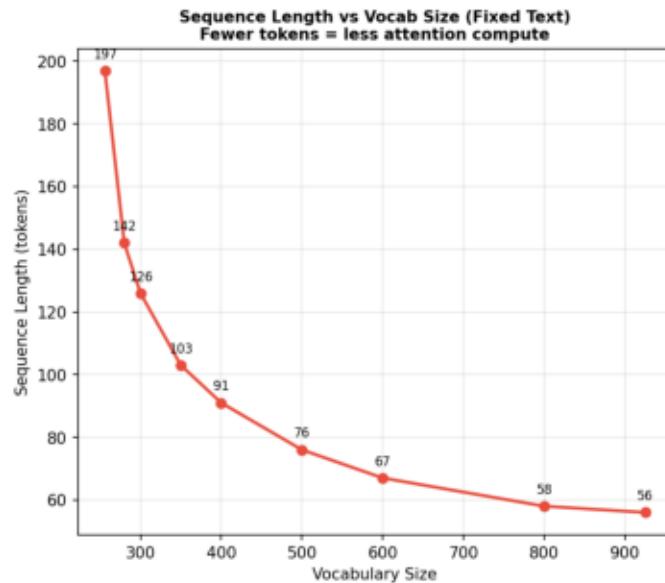
29 tokens, CPT=1.38

Key observations:

- English: best compression (corpus is English-heavy)
 - CJK/emoji: poor compression (falls to byte-level)
 - Code: moderate (some code patterns in training data)
 - Numbers: depends on whether number patterns were seen

Example 4: Vocabulary Size Tradeoff

Vocabulary Size Tradeoff: Sequence Length, Memory, and Compute



VOCABULARY SIZE TRADEOFF
=====

Smaller vocab (e.g. 256):

- + Small embedding table
- Long sequences
- High attention cost $O(n^2)$
- Large KV cache

Larger vocab (e.g. 50K):

- + Short sequences
- + Low attention cost
- + Small KV cache
- Large embedding table
- Rare tokens under-trained

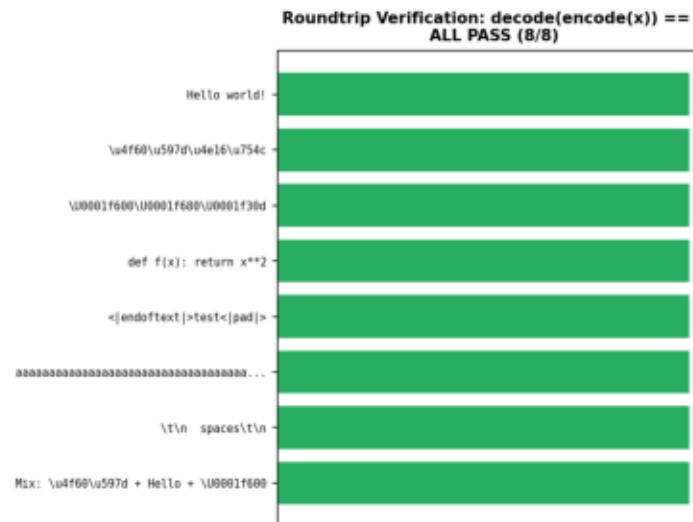
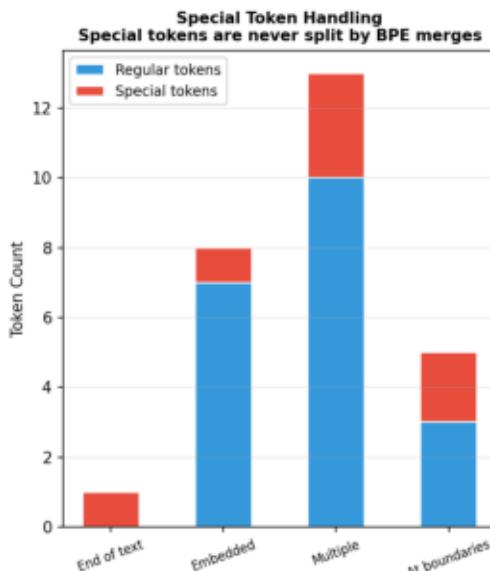
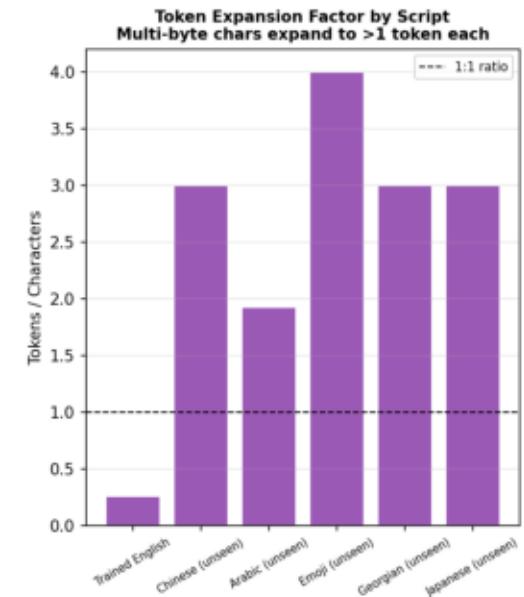
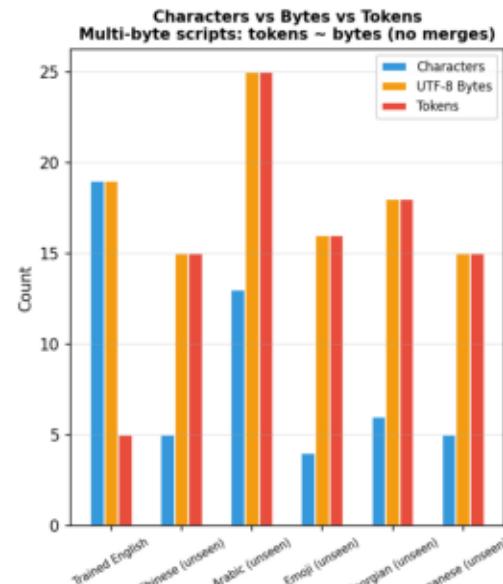
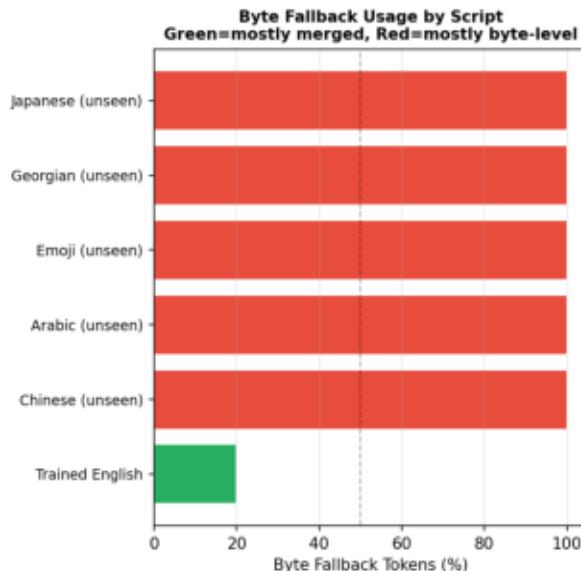
Production choices:

- GPT-2: 50,257 tokens
- GPT-4: 100,256 tokens
- LLAMA: 32,000 tokens
- Mistral: 32,000 tokens

Sweet spot: 32K-100K tokens
balances all costs.

Example 5: Byte Fallback & Special Tokens

Byte Fallback, Special Tokens, and Roundtrip Verification



BYTE FALBACK & SPECIALS

Byte Fallback:

- Base vocab: 256 byte tokens
- ANY UTF-8 byte encodable
- No `<unk>` token needed
- Unseen chars: byte tokens
- Seen chars: merged tokens

Special Tokens:

- Never split by BPE merges
- Encode to single token ID
- Detected before BPE encoding
- Preserved through roundtrip

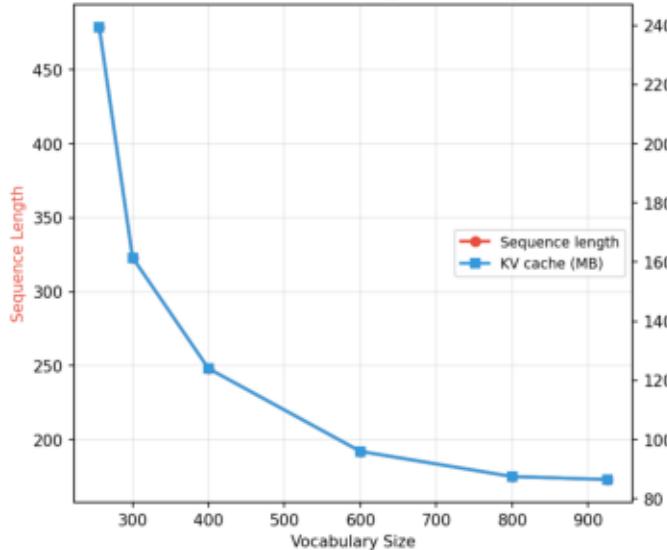
Roundtrip guarantee:
`decode(encode(text)) == text` for ALL valid UTF-8 strings.

This is the key advantage of byte-level BPE over word-level or character-level tokenization.

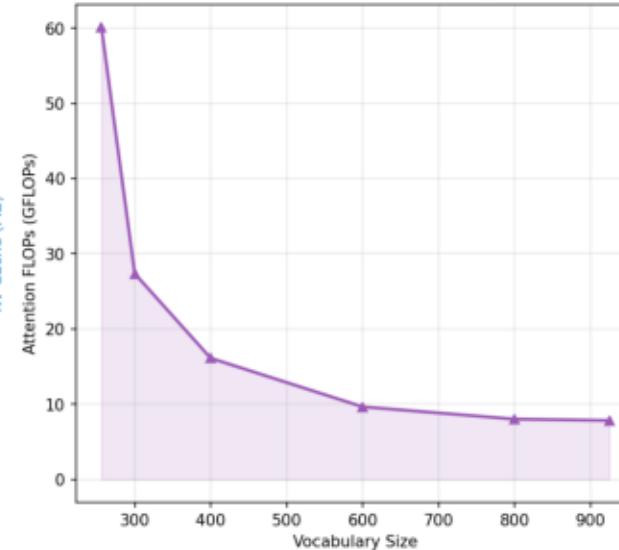
Example 6: Inference Impact Analysis

Tokenization Impact on Inference: Memory, Compute, and Cost

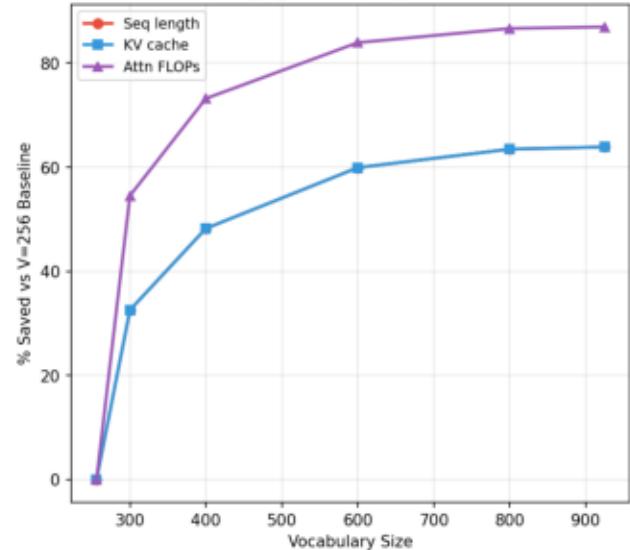
Tokens & KV Cache vs Vocab Size
Both decrease with larger vocabulary



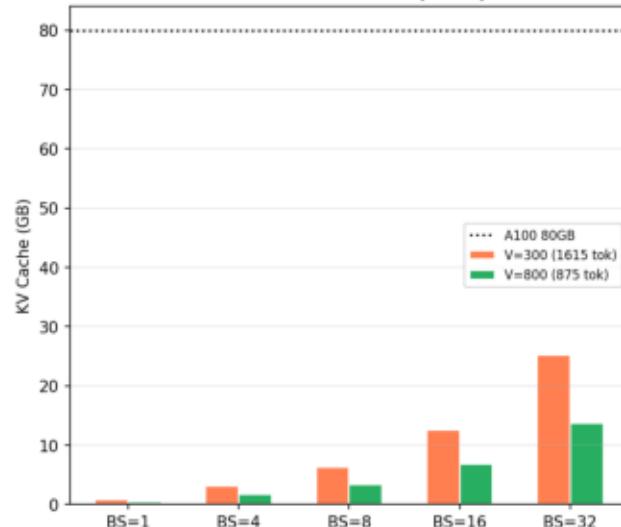
Attention Cost $O(n^2)$ vs Vocab Size
Quadratic reduction from shorter sequences



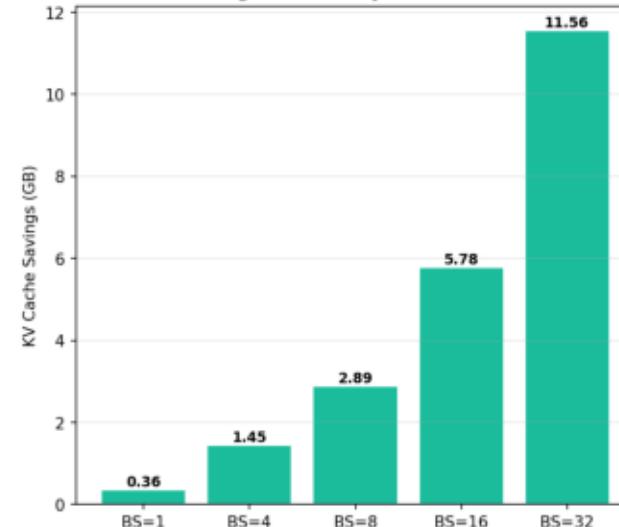
Savings Relative to Byte-Level Baseline
Attn savings grow faster (quadratic effect)



KV Cache: V=300 vs V=800 (Batched)
Better tokenization = more requests per GPU



Memory Saved by Better Tokenization
Savings scale linearly with batch size



INFERENCE IMPACT

- Token count determines:
1. Attention cost: $O(n^2)$
 2. KV cache memory: $O(n)$
 3. Generation latency
 4. Max concurrent requests

This prompt (479 chars):
 V=256: 479 tokens
 V=1000: 173 tokens
 Attn savings: 87%
 KV savings: 64%

Real-world implication:
 Better tokenization means
 more users per GPU, lower
 latency, and lower cost.

This is why tokenizer choice
 is a critical design decision
 for production LLM systems.