



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Welcome

ACT II BEGINS: TOKENIZATION

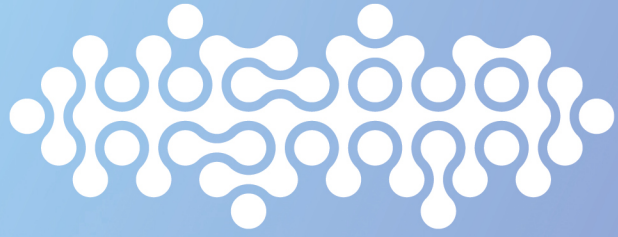
Today's Agenda

Block A (today)

1. Act II begins — from practice to understanding
2. The LLM in one picture
3. A toy model forward pass
4. The strawberry puzzle
5. Word-level → subword → BPE
6. Strawberry explained

Block B (after break)

7. Vocabulary tradeoffs
8. Arithmetic & multilingual failures
9. Glitch tokens
10. Code tokenization
11. End-to-end pipeline
12. Lab exercises



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Act II Begins

1

From Practice to Understanding

In **Act I**, you used AI tools and got impressive results — and puzzling failures.

Act II: Understand the Model Behavior

- Four weeks to open the "black box" layer by layer
- No deep math — just enough to explain what you observed
- Every mysterious failure from Act I will have an answer

Today's layer: Tokenization — the very first step in how a model processes your prompt.

Next-Token Prediction



The model predicts the **next token** from a probability distribution over its entire vocabulary.

Code Generation = Text Prediction

Every AI coding tool — **Copilot, Claude Code, ChatGPT** — runs a Large Language Model.

The core idea:

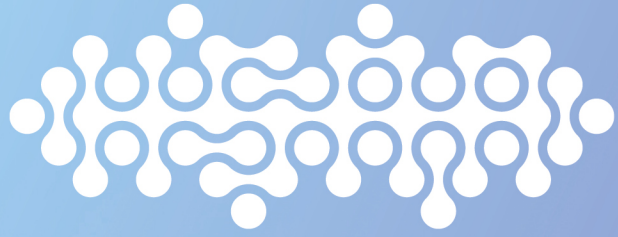
- An LLM is a **next-token prediction machine**
- It picks the most likely next piece of text, given everything before it
- Code generation is text prediction — there is no deeper secret

Every line Copilot ever suggested was produced by this predict-next-token loop.

Six Mysteries from Act I

We will track these across Lectures 4-7 on our [Mystery Board](#):

1. Why did **CLAUDE.md** (a 50-line text file) improve results so much?
2. Why were **English prompts** better than German?
3. Why was **curated context** better than dumping everything?
4. Why did AI **hallucinate** API endpoints that don't exist?
5. Why did **vague prompts** create plausible nonsense?
6. Why doesn't CLAUDE.md **always** work (inconsistent behavior)?

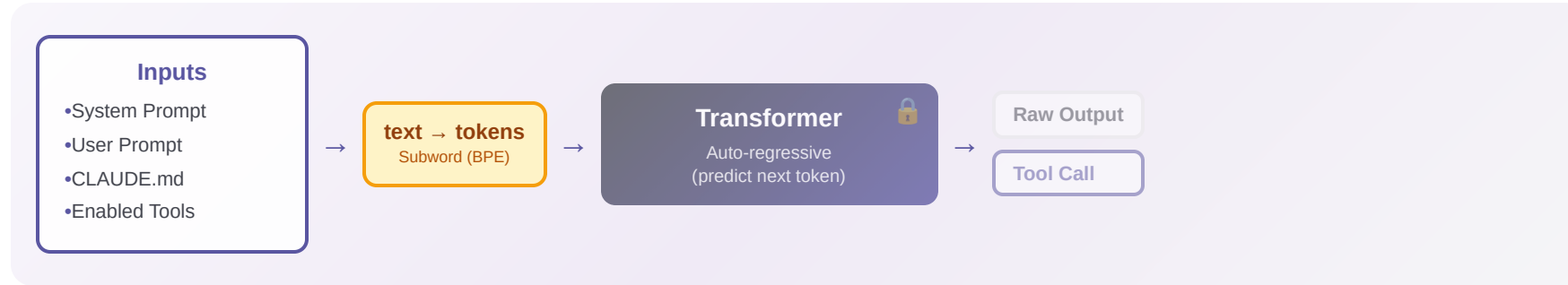


Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The LLM in One Picture

2

LLM Architecture Overview



Today: We zoom into the **text → tokens** step (golden box).

CLAUDE.md enters as regular text — no special "instruction channel" exists.

Why Text Must Become Numbers

Neural networks do not understand text. They understand numbers.

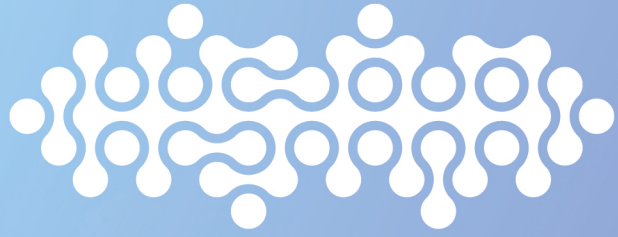
You cannot multiply `def` by `return` — the model needs tensors (multi-dimensional arrays of floats).

The pipeline:

Raw text → **Tokenizer** → Token IDs (integers) → **Embedding Lookup** → Dense vectors (floats) → **Transformer**

The tokenizer is the bridge between human language and machine computation.

Let's trace this entire pipeline end-to-end with a toy example.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Complete Journey in Miniature

3

A Toy Vocabulary

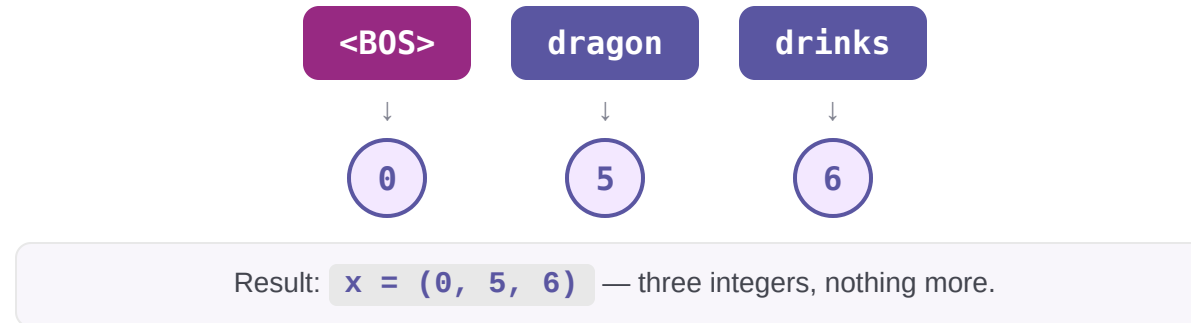
A fantasy world with **V = 10 tokens** and embedding dimension **d = 6**:

ID	Token	ID	Token
0	<BOS>	5	dragon
1	wizard	6	drinks
2	mixes	7	fire
3	glowing	8	troll
4	potion	9	smashes

Dragons drink, wizards mix, trolls smash.

Our input sentence: <BOS> dragon drinks

Step 1: Text to Token IDs



Each token maps to an integer via a simple lookup table.

These are **discrete IDs** — not yet usable by the neural network.

Step 2: The Embedding Table

Embedding matrix $\mathbf{E} \in \mathbb{R}^{10 \times 6}$ — each token gets a row of 6 learnable numbers:

ID	Token	d_1	d_2	d_3	d_4	d_5	d_6
0	<BOS>	0.12	-0.34	0.56	-0.78	0.91	-0.23
1	wizard	-0.45	0.23	0.67	-0.12	0.34	-0.56
2	mixes	0.78	-0.91	0.11	0.45	-0.67	0.89
3	glowing	0.33	0.52	-0.41	0.19	-0.83	0.07
4	potion	-0.62	0.15	0.88	-0.27	0.46	-0.71
5	dragon	0.67	-0.11	0.43	-0.52	0.28	0.85
6	drinks	-0.39	0.74	-0.18	0.61	-0.45	0.33
7	fire	0.54	-0.68	0.22	0.37	-0.14	0.91
8	troll	-0.21	0.47	-0.83	0.66	0.09	-0.55
9	smashes	0.41	-0.36	0.73	-0.58	0.82	-0.17

Rows 0, 5, 6 (bold) are the ones we need. Values are learned during training.

Step 2: The Lookup Result

We extract rows 0, 5, 6 — the transformer receives a **3 × 6 matrix**:

Token	d ₁	d ₂	d ₃	d ₄	d ₅	d ₆
<BOS>	0.12	-0.34	0.56	-0.78	0.91	-0.23
dragon	0.67	-0.11	0.43	-0.52	0.28	0.85
drinks	-0.39	0.74	-0.18	0.61	-0.45	0.33

This is what the transformer actually receives.

Not text. Not integers. Dense floating-point vectors where each dimension captures some abstract aspect of meaning discovered during training.

Step 3: The Transformer (Preview)

3×6 matrix $\rightarrow$ **Transformer**  $\rightarrow 3 \times 6$ matrix

Same shape in, same shape out — but values are completely different.

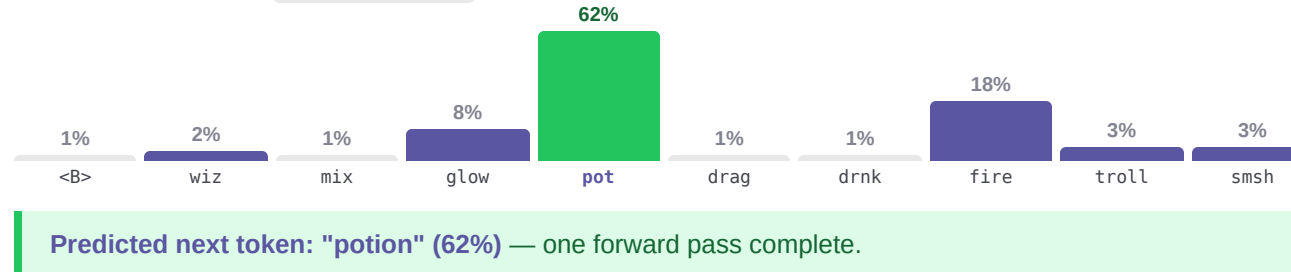
Each layer applies three operations:

1. **Positional encoding** — adds position info ( at position 2 $\neq$ position 3)
2. **Self-attention** — each token asks "how relevant are you?" to every other token
3. **Feed-forward network** — gathers and transforms information

Repeated **12×** (GPT-2) to **96×** (GPT-3). We open this box in **Lecture 5**.

Step 4: Predict the Next Token

Take the **last token's vector** (drinks) → multiply by output weights → 10 logits → softmax:



The Prediction Loop

<BOS> dragon drinks → **potion** (62%)

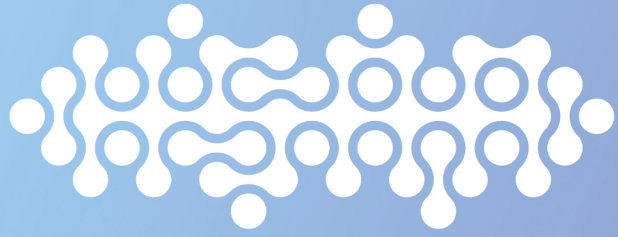
The model appends **potion** and repeats with a 4×6 matrix. Token by token, a story emerges.

Scaling up from toy to real:

	Toy	GPT-2	GPT-3
Vocabulary	10	50,257	50,257
Embedding dim	6	768	12,288
Embedding params	60	38M	618M

The embedding table alone has **38 million parameters** in GPT-2.

The mechanism is identical — just at a vastly different scale.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

What the Model Cannot See

4

The Strawberry Puzzle

How many r's are in the word
STRAWBERRY ?

Take out your phone. Ask your AI model. What does it say?

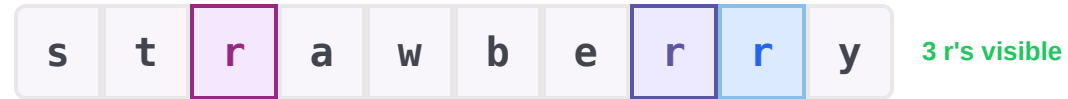
Most models answer **2**. The correct answer is **3**.

Also try: *"How many s's in mississippi?"*

Notebook: [Open §2 in Colab](#) — tokenize `strawberry` and `unbreakable` live

Why Models Get It Wrong

What you see — every letter visible:



↓ tokenization

What the model sees — r's buried inside tokens:



Perception failure, not intelligence failure. The model cannot see individual letters — only token chunks.

Try This: 1234 + 5678

$$1234 + 5678 = ?$$

Ask your model. The correct answer is **6912**.

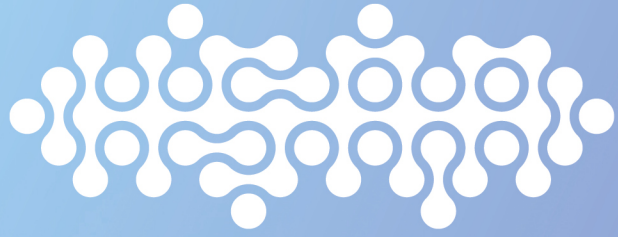
Does it get it right? Wrong? Inconsistently?

Why it fails: The tokenizer splits numbers by learned boundaries, not place value.

Number	Tokenizer split
2249	2 249
2250	22 50
2251	225 1

The model cannot align digits for column arithmetic — because the tokenizer scrambles them.

Notebook: [Open §5 in Colab](#) — tokenize 999, 1000, 1001 and see the inconsistency



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Word-Level Tokenization

5

Three Fatal Flaws

Why not just use one token per word?

1. Vocabulary explosion

- 170,000+ English words, add jargon, typos, inflections → **500,000+** entries
- Embedding cost: $500,000 \times 768 = \mathbf{384 \text{ million parameters}}$ just for the table

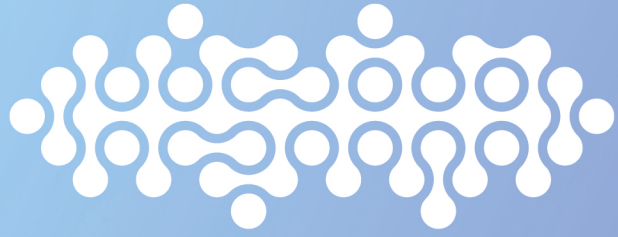
2. The unknown word problem

- Any unseen word → **[UNK]** token → meaning completely lost
- Like reading a book with unfamiliar words blacked out

3. No morphological awareness

- **run**, **running**, **runner**, **runs** = 4 separate entries, no shared structure

Word-level tokenization is essentially **obsolete** for modern LLMs.

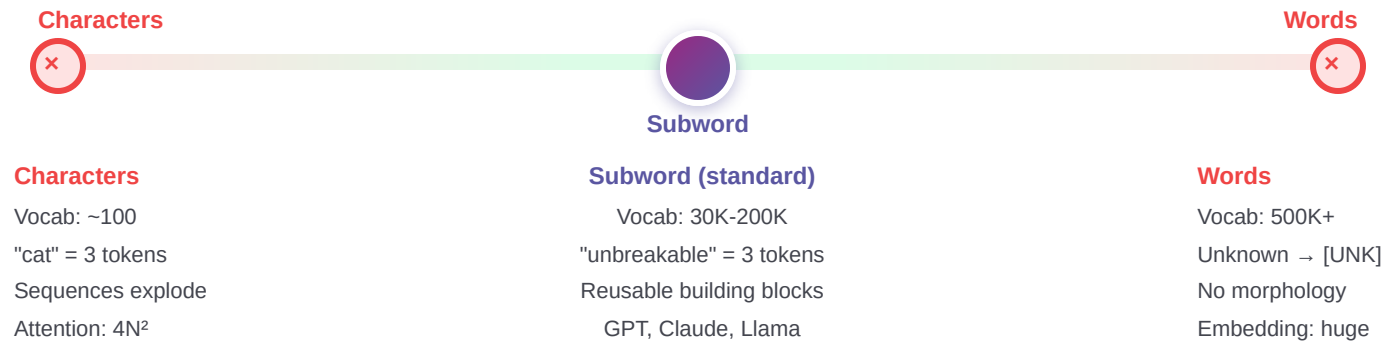


Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Subword Sweet Spot

6

Characters vs Subwords vs Words



Subword tokenization: keep common words whole, decompose rare words into reusable pieces.

The Two Subword Problems

At the extremes, encoding was trivial: every character is a token, or split on spaces. **No ambiguity.**

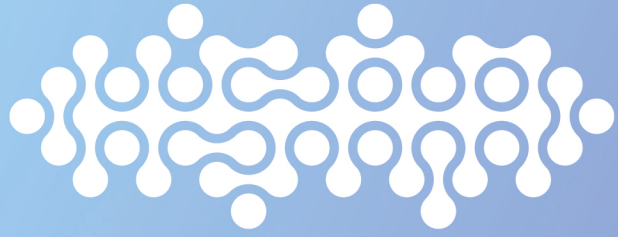
With subwords, two problems arise:

1. **What should be in the vocabulary?** Which subwords do we pick?
2. **How do we encode text into those pieces?** If the vocabulary contains `t`, `h`, `th`, and `the` — how do we split "the"? As `t|h|e`? `th|e`? `the`? All use valid tokens.

A subword vocabulary always contains all characters. Without a smart encoding function, the tokenizer could just use single characters — **wasting every subword entry**.

Tokenization = Vocabulary V + Encoding function f

We need an algorithm that produces both.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Finding the Right Pieces

7

Who Decides What the Pieces Are?

We need subwords — but who picks them?

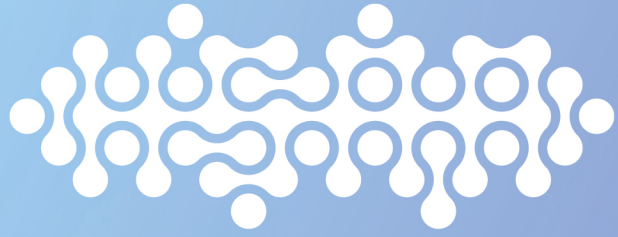
Ask linguists? Too slow, too language-specific. English morphology says nothing about `calculateAverageScore` or Chinese characters.

Ask engineers? Hand-designed rules are arbitrary. Why split after "break" but not "bre"? Doesn't adapt to different corpora.

Let the data decide. Start from raw characters. Merge pairs that co-occur frequently. The most common character sequences naturally become tokens.

This is exactly what **Byte Pair Encoding** does — a compression algorithm (Gage, 1994) repurposed for NLP (Sennrich et al., 2016).

One simple rule: find the most frequent pair → merge → repeat.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Byte Pair Encoding

8

BPE Step 0: Initialize

Training corpus: "the cat in the hat"

Split every word into individual characters:



Vocabulary: {t, h, e, c, a, i, n} — 7 tokens

The algorithm: repeatedly find the most frequent adjacent pair and merge it.

BPE Step 1: First Merge

Most frequent pair: **t + h** (appears 3 times) → merge into **th**

th e . c a t . i n . **th** e . h a t

Rule 1: **t + h** → **th**

Vocabulary: 8 tokens (+1 new)

Notice: **h a t** still has standalone **h** because we merged **t+h**, not **h+a**.

BPE Steps 2-3: Words Emerge

Step 2: a + t $\rightarrow$ at (appears 2×)

th e . c at . i n . th e . h at

Rule 2: a + t $\rightarrow$ at — Vocab: 9 tokens

Step 3: th + e $\rightarrow$ the (appears 2×) — a whole word emerges!

the . c at . i n . the . h at

Rule 3: th + e $\rightarrow$ the — Vocab: 10 tokens

3 merges discovered "the" from pure frequency statistics. No dictionary, no rules.

BPE at Scale

The algorithm is identical — just applied to hundreds of gigabytes of text:

- **GPT-2:** 50,000 merges
- **GPT-4:** ~100,000 merges

Merge progression:

- Early merges: common character combos (`th` , `in` , `er`)
- Middle merges: subwords and short words (`the` , `ing` , `tion`)
- Late merges: full words and phrases (`function` , `return` , `import`)

Key insight: Entirely data-driven. Requires no linguistic rules, no dictionary, no knowledge of any language. The frequency statistics do all the work.

BPE Solves Both Problems

Training produces two outputs:

1. A **vocabulary**: $V = \{t, h, e, c, a, i, n, th, at, the\}$ — 10 tokens
2. An **ordered list of merge rules**:
 - Rule #1: `t + h → th`
 - Rule #2: `a + t → at`
 - Rule #3: `th + e → the`

The merge rules are not just a record of how the vocabulary was built. They **are** the encoding function.

When the tokenizer encounters new text, it does not re-run training. It **replays the merge rules in order**.

Encoding New Text: "the hat"

Start: split into characters: t h e h a t

Rule #1 (t+h → th): th e h a t

(The h in "hat" is preceded by a space, not t — no match there.)

Rule #2 (a+t → at): th e h at

Rule #3 (th+e → the): the h at

No more rules. Result: the | h | at — three tokens.

Not "longest match" — **merges replay in the order they were learned**. Rule #1 fires first because t+h was the most frequent pair during training. The merge order *is* the priority.

When to Stop

In our toy example, we stopped after 3 merges — the remaining pairs all have count 1 (no signal left).

In practice, the **target vocabulary size** is a hyperparameter chosen before training:

Model	Vocabulary	Merges
GPT-2	50,257	50,000
GPT-4	~100,000	~100,000
LLaMA 3	128,000	~128,000

On billions of words, there is always enough frequency signal. The algorithm runs until the vocabulary reaches the target size, then stops.

Strawberry Explained

Back to our puzzle: `strawberry` → `str | aw | berry`

Where are the r's?

- `str` — one r (at the end)
- `aw` — zero r's
- `berry` — two r's

The model sees **3 token chunks**, not 10 letters. The r's are scattered across token boundaries.

Tokens are atomic. The model cannot reach inside a token to examine individual characters.

Layer 1 — perception: The model cannot see what it would need to count.

But Wait — Would Letter Tokens Fix It?

Suppose we tokenized letter by letter: `s | t | r | a | w | b | e | r | r | y`. Every r is now visible. **Would the model reliably answer "3"?**

Probably not. To count, the model would need to:

1. Scan every token, check if it is `r`
2. Maintain a **running count** that increments
3. Output the final number

A transformer has **no variables, no loops, no accumulator**. It produces the next token from a probability distribution — not by executing a procedure.

Layer 2 — architecture: The answer "3" would have to emerge as the statistically most likely continuation, learned as a memorized pattern from training data.

That's why training on "How many r's in strawberry? → 3" *fixes* the famous cases — pattern memorization, not improved counting. Try a less famous word and the failure returns.

Byte-Level BPE: The Unknown-Character Problem

Character-level BPE breaks down on real-world text:

- English has 26 letters. Chinese has **tens of thousands** of characters.
- Add emoji (3,600+), math symbols (2,000+), diacritics, rare scripts...
- Character-level base vocabulary: **over 100,000 entries** before any merges.

Worse: Unicode adds new emoji every year. A tokenizer trained last week has **no way to encode them**. The UNK problem comes back, just dressed differently.

We need a base vocabulary that is universal, small, and future-proof.

Standing on the Shoulders of Unicode & UTF-8

LLMs inherited a 30-year head start on text encoding:

- **Unicode (1991):** every character in every script → a unique integer (code point). ~150,000 assigned today.
- **UTF-8 (1993):** encode every code point as 1-4 bytes. ASCII-compatible. Now powers >98% of the web.

Character	Bytes	Hex
H	1	0x48
ü	2	0xC3 0xBC
☐	3	0xE4 0xB8 0x96
😊	4	0xF0 0x9F 0x98 0x8A

Tokenizer design starts *after* text has been reduced to bytes — the hard part was solved long before transformers.

What Is a Byte, Actually?

A byte = **8 bits** = (2^8) = **256 distinct values**, numbered 0 to 255. That is *all* a byte is.

The same byte can be written in four different notations — they all mean exactly the same thing:

Notation	The letter H	When you see it
Binary	01001000	low-level systems, protocol diagrams
Hexadecimal	0x48	hex editors, protocol specs
Decimal integer	72	programming languages, tokenizer code
ASCII character	H	text editors, console output

Context decides meaning. Byte **72** is the number 72 in arithmetic, the letter **H** in ASCII text, or a gray pixel shade in an image.

Python's `list("Hello".encode("utf-8"))` → `[72, 101, 108, 108, 111]`. Integers are the form tokenizer code actually uses.

How 4 Bytes Cover More Than 150,000 Characters

If one byte has only 256 possible values, how can UTF-8 encode Unicode's 150,000+ code points?

The trick: the first byte announces how many bytes the character occupies.

Length	Byte pattern	Payload bits	Code points
1 byte	0xxxxxxx	7	128 (ASCII)
2 bytes	110xxxxx 10xxxxxx	11	2,048
3 bytes	1110xxxx 10xxxxxx 10xxxxxx	16	65,536
4 bytes	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	21	2,097,152

Self-describing leading bits: 0... standalone • 110... starts 2-byte • 1110... starts 3-byte • 11110... starts 4-byte • 10... continuation (never starts).

4 bytes → 21 payload bits → 2M+ values — more than enough for all of Unicode. ASCII stays cheap; non-ASCII pays only where it must.

The Universal 256-Byte Base Vocabulary

Once text is bytes, the base vocabulary becomes trivial:

256 possible byte values → **256 base tokens**. That is it.

This one design choice covers:

- Every written language (Latin, CJK, Arabic, Cyrillic, ...)
- Every emoji, mathematical symbol, currency symbol
- Any emoji that Unicode adds **next year**
- Even raw binary data

Compare: character-level BPE needs **100,000+** entries and still misses next year's additions.

Byte-level BPE: **256 forever**.

No UNK token. No gaps. Ever.

Why a New Emoji Doesn't Require Retraining

Scenario (2027): Unicode 17.0 adds a brand-new emoji. Its UTF-8 bytes are `F0 9F AA 99` = `[240, 159, 170, 153]` — never seen by GPT-4o's 2024 tokenizer.

End-to-end walkthrough:

1. **Encode** — no merge rule matches (those bytes never co-occurred in training). Emoji → 4 base-byte tokens. **No UNK, no error, no retrain.**
2. **LLM process** — embedding rows 240, 159, 170, 153 existed from day one (§13.3). Forward pass unchanged.
3. **Generate** — model can emit the 4 bytes in order. Verbatim reproduction works; semantic meaning comes from in-context learning.

Tokenizer type	Behavior on post-training emoji
Word-level	<code>[UNK]</code> — information lost
Character-level (~100k base)	<code>[UNK]</code> — code point not in vocab
 Byte-level (256 base)	4 base-byte tokens. Lossless.

Takeaway: UTF-8 is forward-compatible. Every future Unicode character is just more bytes

The Algorithm, at Byte Level

Same five functions as character-level BPE — now operating on bytes:

1. `get_stats(ids)` — count adjacent token pairs
2. `merge(ids, pair, new_id)` — replace every occurrence of `pair` with `new_id`
3. `train_bpe(text, num_merges)` — repeat 1+2, record merges **in learning order**
4. `encode(text, merges)` — apply merges **in training order** to new text
5. `decode(ids, vocab)` — concatenate byte sequences, UTF-8-decode

Merge order matters. Later merges may depend on earlier ones: if `(t, h)→256` and `(256, e)→257`, you must apply merge 1 first or merge 2 finds nothing.

Exercise: implement all five functions yourself in the companion notebook.

Worked Example: "Müller" at the Byte Level

Corpus: "Müller Müller Müller" — encode as UTF-8:

```
[77, 195, 188, 108, 108, 101, 114, 32, 77, 195, 188, ...]  
M  ( ü )  l  l  e  r  M  ( ü )
```

BPE does not know these are bytes. It just counts adjacent integer pairs. Top pairs in this corpus:

Pair	Count	What it is
(195, 188)	3	the two bytes of <code>ü</code>
(108, 108)	3	<code>ll</code>
(101, 114)	3	<code>er</code>

Merge (195, 188) → 256. Token 256's bytes are `b'\xc3\xbc'` → decodes to `"ü"`.

BPE just "discovered" the umlaut without being told about Unicode. UTF-8 guarantees the two bytes always co-occur — so BPE merges them automatically.

Lossless vs. Efficient: Two Different Worries

If English dominates training and `ü` appears rarely, does BPE *lose* `ü`? Depends what you mean.

Can `ü` become unrepresentable? No, never.

The 256 base bytes exist before any training runs. `"Müller"` still encodes as `[77, 195, 188, 108, 108, 101, 114]` and decodes perfectly. **Byte-level BPE is lossless, always.**

Can `ü` become expensive? Absolutely.

If `(195, 188)` is rare, it never wins the frequency contest. `"Müller"` costs 7 tokens; English `"Miller"` might cost 2-3. **Same word, 3× the cost.**

Same bytes, different groupings. The vocab always stores UTF-8 bytes. Frequent patterns get their own token (cheap); rare patterns stay as base bytes (expensive but correct).

This is the **multilingual fairness gap** — a consequence of the training-data mix, not the algorithm.

Scale Reality: Toy vs Production

System	Training corpus	Merges	Final vocab
Our toy BPE	one sentence	20	276
GPT-2 (2019)	WebText, ~40 GB	50,000	50,257
GPT-4 (c l100k_base)	multi-TB crawl	~100,000	100,277
GPT-4o (o200k_base)	multi-TB crawl	~200,000	200,019
LLaMA 3	undisclosed	128,000	128,256

Same algorithm — **six orders of magnitude** apart in scale. Embedding-matrix cost: 200K vocab $\times$ 12K dim $\times$ 2 B $\approx$ **4.9 GB**, before the transformer itself uses a byte.

Where the Data Comes From

"Trained on the internet" hides a real pipeline:

1. **Extraction** — strip HTML, nav, ads. Keep main text.
2. **Language detection** — curate the mix (don't take what the crawl happens to sample).
3. **Quality filtering** — drop spam, low-signal docs.
4. **Deduplication** — MinHash / SimHash. Duplicates bias merges and cause memorization.
5. **Mixture construction** — combine web + books + Wikipedia + code + papers.

The merges you get are a fingerprint of your data pipeline, not a property of the algorithm. Two labs running identical BPE code on different data produce very different tokenizers. This is why data curation matters more than most people realize.

Try It Yourself

Two companion notebooks turn this theory into output you can watch:

- [**Build BPE From Scratch**](#) — implement the 5 functions yourself in pure Python, in-browser (JupyterLite). Now trains on a mixed multilingual + code + emoji corpus so you see byte-level BPE discover German umlauts, Python indentation, and emoji bytes side by side.
- [**Tokenizer Playground**](#) — load tiktoken + HuggingFace, compare GPT-4o, LLaMA 3, Gemma 2 on identical text (Colab).

The lesson: byte-level BPE is simple enough to implement in an afternoon — *and* it is exactly what every frontier LLM uses. The cleverness of GPT-2 was noticing the pieces (Unicode, UTF-8) were already on the shelf.

Three Algorithms Compared

Algorithm	Direction	Criterion	Used By
BPE	Bottom-up	Most frequent pair	GPT-2/3/4, Claude, Llama, Mistral
WordPiece	Bottom-up	Highest likelihood ratio	BERT, DistilBERT
Unigram	Top-down	Least loss when removed	T5, mT5, Gemma

BPE has won. Nearly every frontier model (2024-2025) uses byte-level BPE.

All three solve the same two problems: building a **vocabulary** and defining an **encoding function**. Different methods, same goal.

Attribution & Sources

This lecture's tokenization content draws on:

- **"How AI Actually Reads Text | Tokenization"** by Attention Visualized (YouTube)
- minbpe by Andrej Karpathy — inspiration for the BPE notebook exercise

The narrative structure, toy example, diagrams, and pedagogical framing are original to this course.

See you after the break for vocabulary tradeoffs, real-world failures, and the lab.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

BREAK

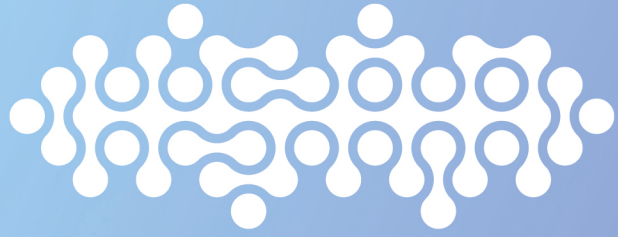
Back in 15 minutes for Block B.

Where We Left Off

Block A recap:

- LLMs are **next-token prediction machines**
- Text → Token IDs → Dense vectors → Transformer
- Toy example: `<BOS> dragon drinks` → **potion** (62%)
- The **strawberry puzzle**: tokenization shapes what the model can see
- **BPE**: data-driven subword tokenization — the industry standard

Block B: What happens when tokenization meets the real world?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Vocabulary Size Tradeoffs

9

The Engineering Balance

Smaller vocabulary

- Words split into more pieces
- Sequences get longer
- Attention scales quadratically (expensive)
- Less text fits in context window

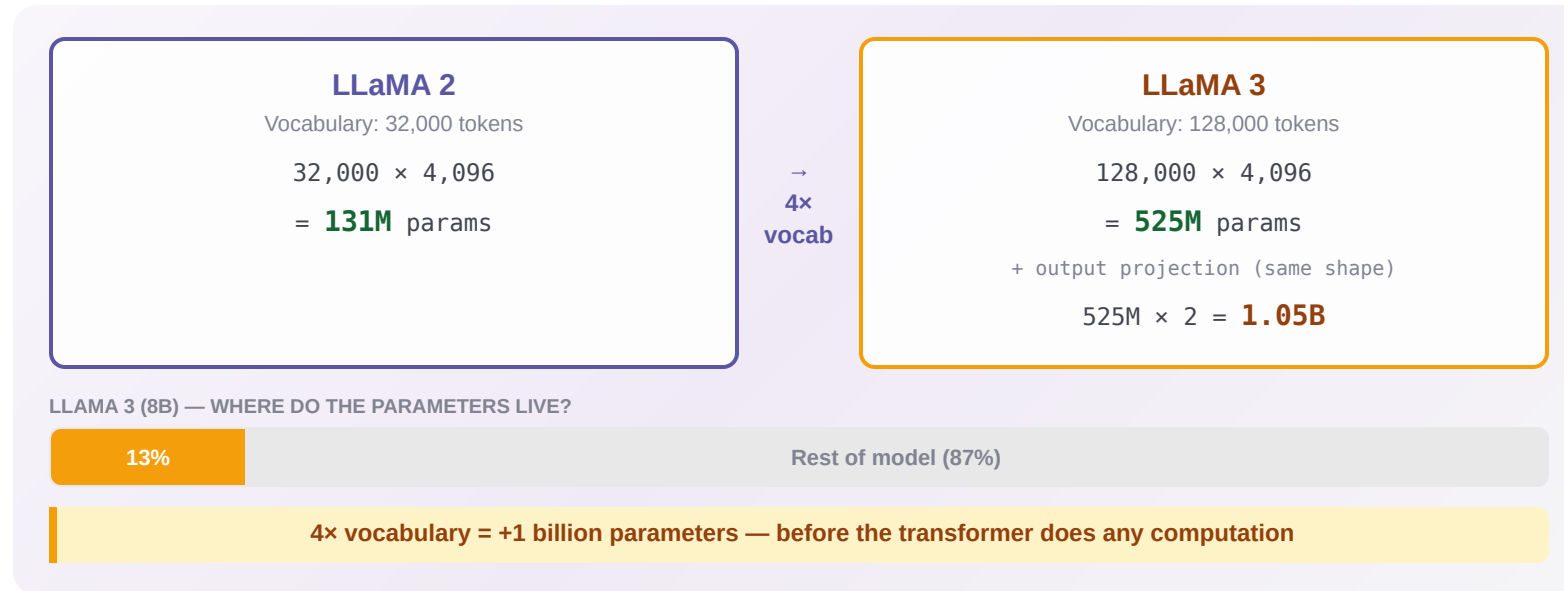
Larger vocabulary

- Fewer tokens per sentence
- But: embedding matrix grows linearly
- Output projection matrix too
- More parameters = more memory

The tradeoff: compression gains vs. parameter cost.

The industry is trending toward larger vocabularies as models grow.

The Parameter Cost



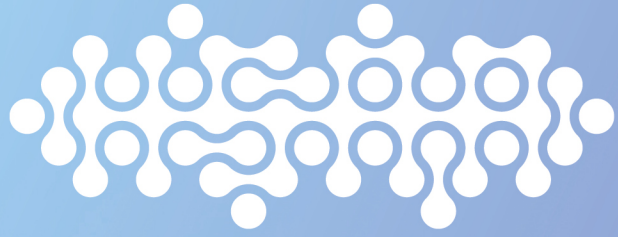
Vocabulary Sizes Over Time

Model	Year	Vocabulary	Method
BERT	2018	30,000	WordPiece
GPT-2	2019	50,000	Byte BPE
LLaMA 2	2023	32,000	BPE/SentencePiece
GPT-4	2023	100,000	Byte BPE
GPT-4o	2024	200,000	Byte BPE
LLaMA 3	2024	128,000	Byte BPE
Gemma	2024	256,000	BPE/SentencePiece

Trend: Larger vocabularies, especially as models grow and training data expands.

Compression gains outweigh parameter cost — if you have enough training examples per token.

Notebook: [Open §3 in Colab](#) — compare GPT-4o, GPT-4, LLaMA 3, Gemma 2 on the same text



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Why LLMs Fail at Arithmetic

10

The Place-Value Problem

Hand arithmetic: align digits by place value, work right-to-left.

But the tokenizer splits numbers by **learned boundaries**, not place value:

Number	Tokenizer split	Boundary
2249	2 249	After 1st digit
2250	22 50	After 2nd digit
2251	225 1	After 3rd digit
999	999	One token
1000	1000	One token
1001	100 1	Two tokens

Adding **1** to a number can completely change its internal representation.

The model cannot align digits for column arithmetic.

The Fix Is in the Tokenizer

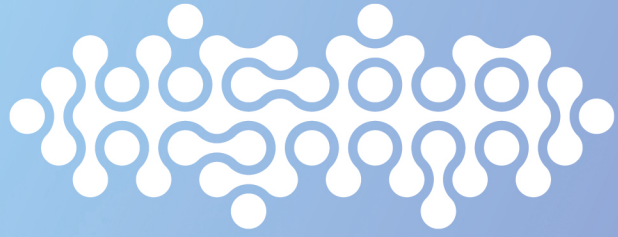
Gemma (Google): Forces every digit to its own token, regardless of BPE.

2164 → 2 | 1 | 6 | 4 — perfect place-value alignment.

ICLR 2025 research: Tokenizing digits **right-to-left** improves arithmetic by **22 percentage points**.

The fix is a **smarter tokenizer**, not a smarter model.

This is a recurring theme: many apparent model failures trace back to the tokenizer.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Multilingual Fairness Gap

11

The Token Tax

"artificial intelligence" in English: **2 tokens**. The same concept in other languages:

Language	Approx. Tokens	Relative to English
English	2	1×
Spanish	3-4	1.7×
Chinese	2-4	1.5×
German	3-5	2×
Arabic	5-8	3×
Hindi	8-12	5×
Burmese	12-20	8×

German prompts consume context window roughly **2× faster** than English.

API cost: what costs \$1 in English costs ~\$2 in German and ~\$5 in Hindi.

Why This Happens

Training data: ~85% English, 15% all other languages combined.

BPE learns efficient subwords for English. German compounds, Hindi conjuncts, Burmese sequences are rare in training — they never get merged into efficient tokens.

The practical impact:

200K context window	English	German	Hindi
Words that fit	~150K	~100K	~30K
API cost (same content)	\$1	\$1.50-2	\$4-5

The gap is narrowing (LLaMA 3 halved Hindi token count), but it persists.

Not model bias — tokenizer bias, at the very first pipeline step.

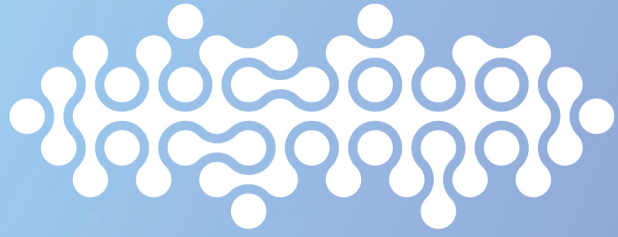
Mystery 2: SOLVED

Mystery 2: Why were English prompts better than German?

Answer: Token efficiency. English prompts use fewer tokens for the same content. Fewer tokens = more room in context window, less attention computation, lower cost, and better quality.

This is not about the model understanding English better. It's about the **tokenizer compressing English more efficiently** because it was trained on predominantly English data.

Notebook: [Open §4 in Colab](#) — measure the EN-DE token ratio yourself



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Glitch Tokens

12

SolidGoldMagikarp

January 2023: Certain tokens cause models to produce **nonsense**.

Prompt: *"Please repeat: SolidGoldMagikarp"* → Model says "Distribute" or garbles output.

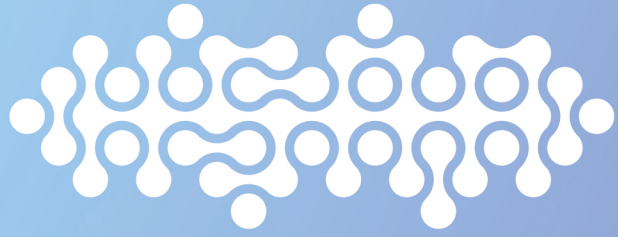
Root cause: Tokenizer and model trained on **different datasets**.

- Tokenizer's data: raw internet (Reddit, code, forums)
- Model's data: curated, filtered
- `SolidGoldMagikarp` = super-active Reddit username (r/counting subreddit)
- Tokenizer gave it a dedicated token; model never saw it during training
- Embedding = **pure random noise**

2024 study: ~7,900 glitch tokens across 7 major models — roughly **4% of all tokens**.

A token without a trained embedding is worse than useless — it's noise.

Notebook: [Open §7 in Colab](#) — check if SolidGoldMagikarp is still a single token in GPT-4o



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Code Tokenization

13

Indentation Is Expensive

Python uses whitespace for structure. Every indentation level costs tokens.

- Flat function: ~20 tokens
- Same logic nested 3 levels: ~30 tokens
- **10 tokens just for indentation** — structural meaning, no semantic content

Implication: Shorter, flatter code gives the model more room for reasoning. This is a **token budget decision**, not just a style preference.

Naming, Boilerplate, and Language

Naming conventions:

- `calculateAverageScore` (camelCase): 1-2 tokens (common pattern)
- `calculate_average_score` (snake_case): more tokens (splits at underscores)

Boilerplate vs. novel code:

- `def __init__(self):` → 5-6 tokens (seen billions of times)
- `def calibrate_magnetometer(readings):` → 12+ tokens (rare identifiers)
- Model sees boilerplate with **high resolution**, novel code with **low resolution**

Programming languages differ:

- `public static void main(String[] args)` (Java) costs more tokens than Python equivalent
- Same concept as the multilingual gap — verbose languages consume more context



Tokenizers Built for Code

Everything above assumed a **general-purpose tokenizer** (GPT-4o, Llama). Code-focused labs sometimes retrain.

Two patterns in production:

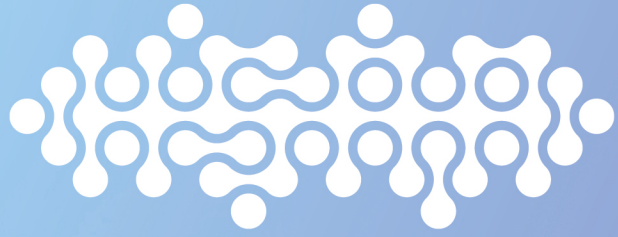
Model	Approach
Code Llama	Reused Llama 2's tokenizer + added <code><FIM_*></code> special tokens for infilling
StarCoder2	Custom tokenizer on The Stack v2 — 67.5 TB, 619 programming languages

DeepSeek Coder retrains; the original Codex reused GPT-3's unchanged.

A code-aware tokenizer fixes §20.1–20.4 pain points: indentation becomes cheap (4 spaces → 1 token), identifiers compress more aggressively, Rust/Java/JSX syntax gets first-class treatment, the programming-language gap shrinks.

Fill-in-the-middle (FIM): `<FIM_PREFIX>before<FIM_SUFFIX>after<FIM_MIDDLE>` — 3 hand-added special tokens (not BPE-learned) unlock IDE-style completion.

For you: Claude Code / Cursor / Copilot run frontier models with *general* tokenizers. The gap to a specialist is smaller than you'd guess (typically 20–40% on code snippets) but real.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Complete Journey

14

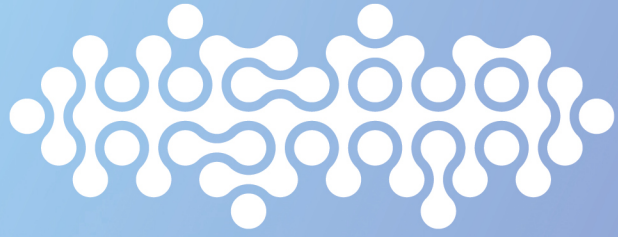
6 Stations: Text to Transformer

The full pipeline in a real model (e.g., GPT-4o):

1. **Pre-tokenization** — regex splits at word boundaries (`don't` → `do` | `n't`)
2. **BPE segmentation** — applies learned merge rules (`tokenizer` → `token` | `izer`)
3. **Token ID assignment** — maps each piece to an integer (`token` → 11241)
4. **Special token insertion** — adds `<|endoftext|>` boundaries
5. **Embedding lookup** — each ID indexes the embedding matrix (50K+ rows, 768+ dims)
6. **Positional encoding** — wave pattern overlays encode token position

The transformer receives dense, position-aware vectors — not your original text.

Every tokenizer bias carries forward through all layers.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

What This Means for CLAUDE.md

15

First Clue to Mystery 1

CLAUDE.md is text → becomes tokens → becomes vectors → enters transformer.

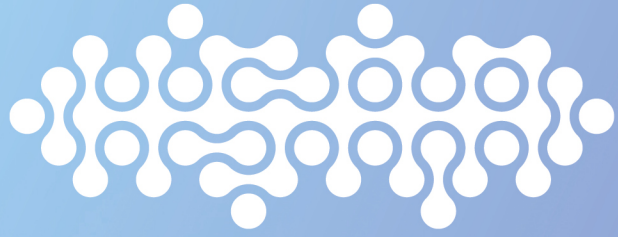
- A 50-line CLAUDE.md $\approx$ 200-500 tokens
- Same BPE pipeline. Same embedding lookup. Same positional encoding.
- Processed by the **exact same transformer** as your question.

There is no special "instruction channel." No configuration system. No separate memory.

From the model's perspective, CLAUDE.md tokens are **indistinguishable** from question tokens. The distinction exists only in the software managing the model, not in the model itself.

This is our first clue to Mystery 1 — but the complete answer requires understanding attention (Lecture 5) and orchestration (Lecture 7).

Notebook: [Open §8 in Colab](#) — estimate how many tokens your CLAUDE.md consumes



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Key Takeaways

16

Five Core Principles

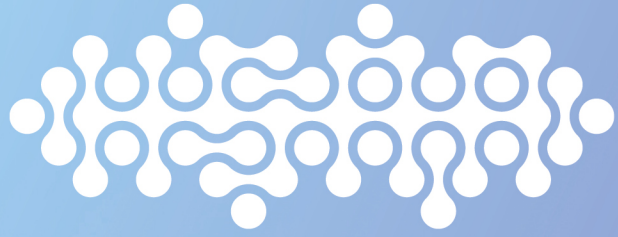
1. **Tokenization is the bridge** — text → integers → vectors → transformer. Always two things: a **vocabulary** V and an **encoding function** f
2. **Subword tokenization is the universal standard** — BPE solves both problems via ordered merge rules; WordPiece and Unigram viable alternatives
3. **Consequences are real and measurable** — arithmetic failures, multilingual cost gap, glitch tokens. Not model bugs — **tokenizer bugs**
4. **Vocabulary size is an engineering tradeoff** — smaller = longer sequences; larger = more parameters
5. **Tokenization is architecture** — it shapes what the model sees, how much it holds in context, and where it stumbles. When a model surprises you: **consider the tokenizer first**

Mystery Board Update

#	Mystery	Status
1	Why did CLAUDE.md improve results?	Partial — it's tokens, but <i>how</i> they influence = Lectures 5-7
2	Why English > German prompts?	Solved ✓ — token efficiency
3	Curated context > dump everything?	Partial — token budget matters; attention = Lecture 5
4	AI hallucinating API endpoints?	Open — Lecture 6
5	Vague prompts → plausible nonsense?	Open — Lecture 6
6	CLAUDE.md inconsistent?	Open — Lectures 6-7

One solved, two partial, three open. Next week we open the transformer box.

Full board: [Mystery Board](#)



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Lab: Tokenizer Playground

17

Notebooks and Deliverables

Tokenizer Playground — we explored §2-§8 live during the lecture. Complete **§9 (Your Findings)** as your deliverable: [Open in Colab](#)

Build BPE From Scratch (self-study): [Open in JupyterLite](#)

Deliverables (AI Diary entry — notebook §9):

- At least **5 surprising tokenization results** with explanations
- **English-German token count comparison** with calculated ratio
- **CLAUDE.md token budget estimate** — how much context window does it consume?
- **One change to your prompting approach** based on tokenization understanding



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Questions?

THANK YOU

Your feedback is welcome.