



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

SoSe 2026

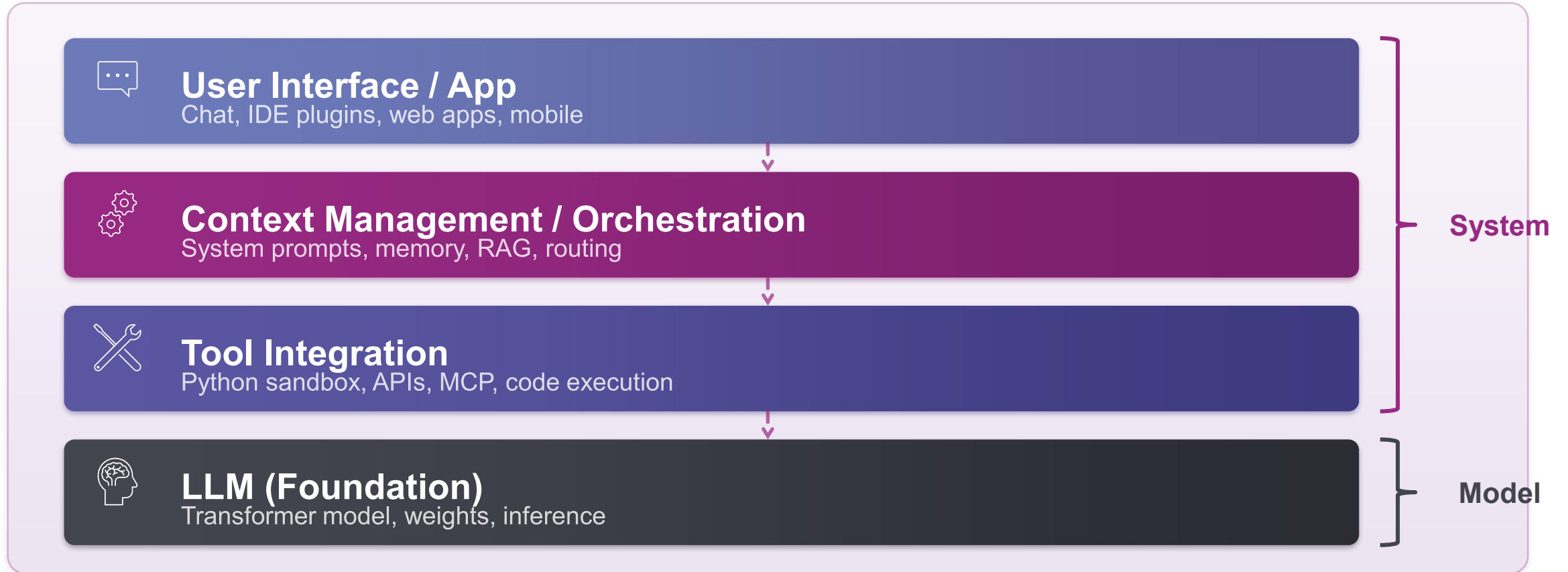
AI Supported Software Development

Week 4 – Tokenization: How Machines Read

Prof. Dr. Dominik Müller

AI Systems in a nutshell

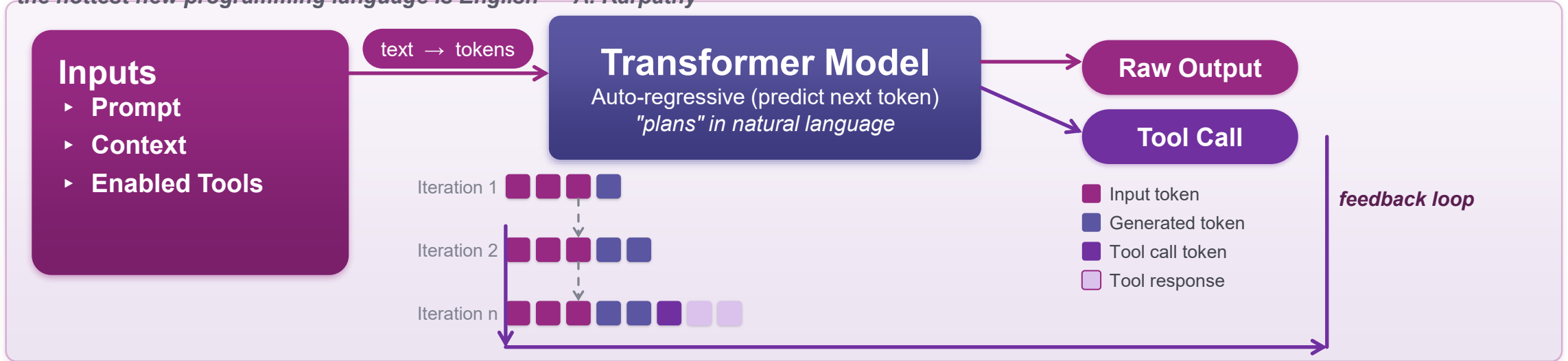
The model is only a (small) part



The LLM provides the **intelligence**, the system provides the **capability**

LLM in one picture

“the hottest new programming language is English” – A. Karpathy



Glossary

- **Prompt** (user provided input instructions), **Context** (extra text the model sees)
- **Token** (sub-word unit, model operates on tokens)
- **MCP** (generic interface / protocol to make any external tool available to an LLM)

Tools / Connectors

- External actions (HTTP, email, git, database, ...) **exposed to the model**
- On the fly **code generation & execution**

Four-Act Structure

Act I Experience Weeks 1–2

“The Wow and the Oops”

We USE AI tools before understanding them. We experience both the power and the failures firsthand, creating the cognitive tension that motivates everything that follows.

Act II Understand Weeks 3–7

“How Does This Actually Work?”

We learn foundational concepts: tokenization, embeddings, attention, context windows, inference, hallucination mechanisms. Prompting is taught as the applied side of these fundamentals.

Act III Master Weeks 8–10

“Wielding It Professionally”

We learn to evaluate AI output quality rigorously, integrate AI into real development workflows (IDE assistants, agentic pipelines, CI/CD), and measure productivity impact.

Act IV Govern Weeks 11–13

“The Bigger Picture”

We confront risks, security, organizational impact, and the economics of AI adoption. The course ends with you defending your own AI integration strategy.

AI AUTONOMY

Chat Assistant

Knowledge Partner

Workflow Integrator

Autonomous Agent

Today's Agenda

Lecture 4 — Tokenization: How Machines Read

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

From Practice to Understanding

Act II — Understand the model behavior

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.

LLMs are Next Token Predictors

1

Next-Token Prediction

What powers your AI tools

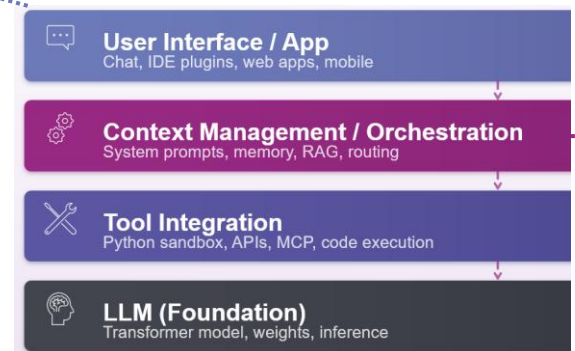
System: You are an expert Python developer.

User: Write a function that validates an email.

Assistant:

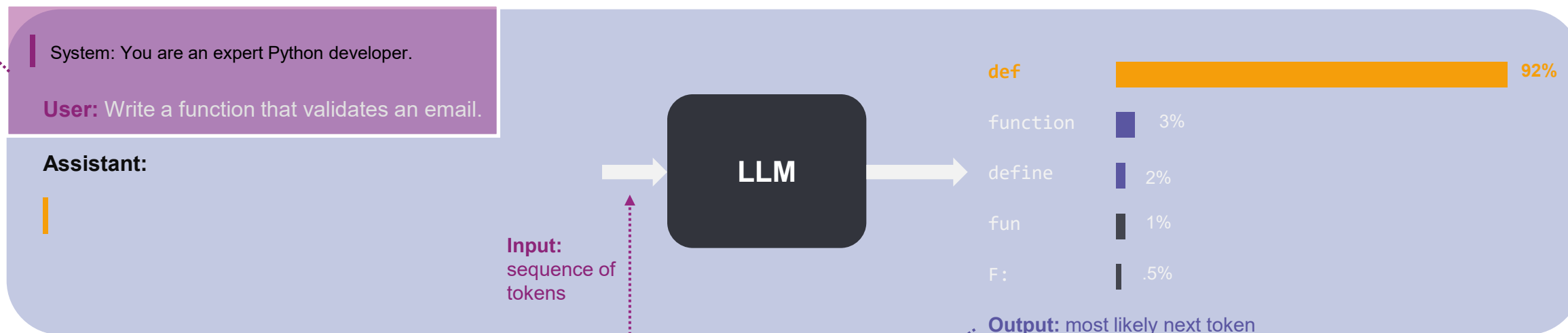


What happens after the user (and system input) such that we end up with generated code?



Next-Token Prediction

What powers your AI tools

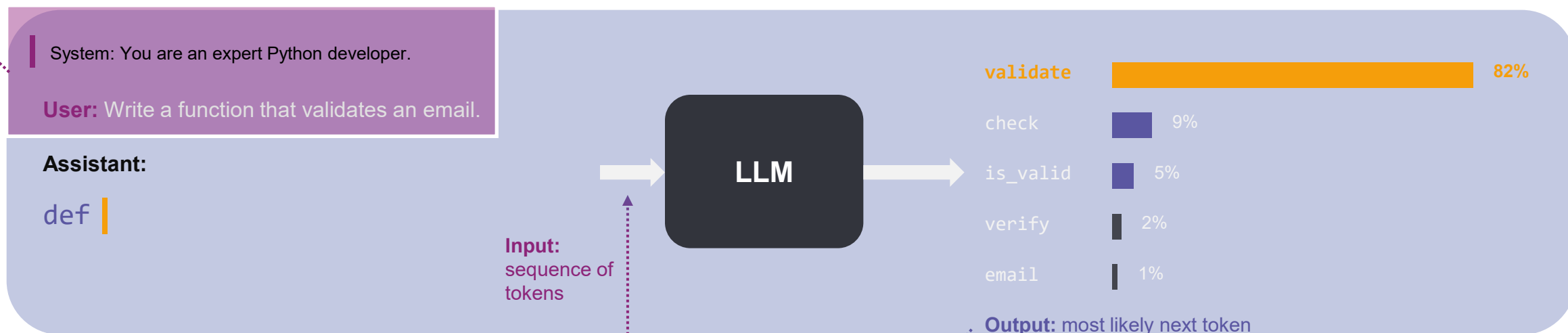


The model predicts the next token from a probability distribution over its entire vocabulary.

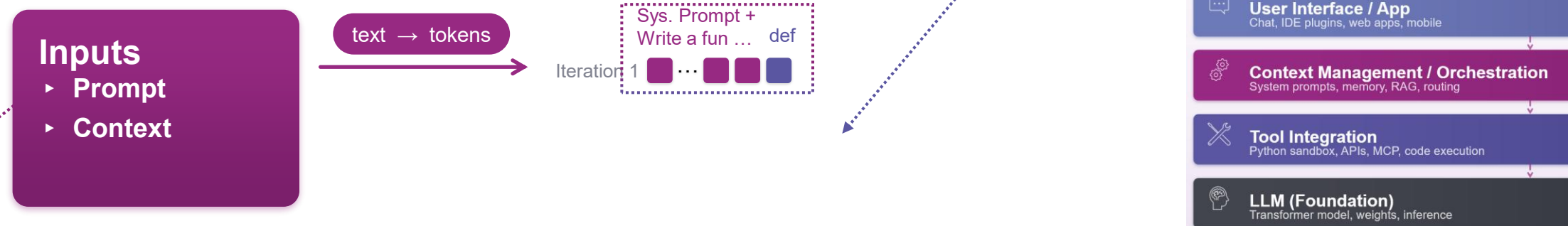


Next-Token Prediction

What powers your AI tools



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



Next-Token Prediction

What powers your AI tools

System: You are an expert Python developer.

User: Write a function that validates an email.

Assistant:

```
def validate
```

Input:
sequence of
tokens

LLM

`_email`

74%

Email

11%

send

5%

user

2%

at

1%

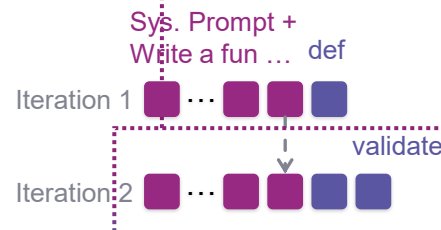
Output: most likely next token

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

Inputs

- Prompt
- Context

text → tokens



Next-Token Prediction

What powers your AI tools

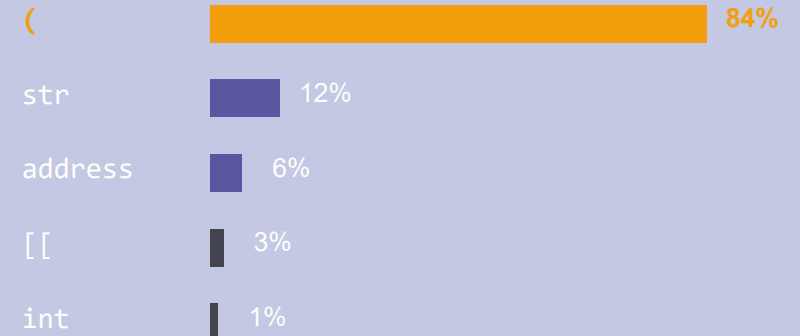
System: You are an expert Python developer.

User: Write a function that validates an email.

Assistant:

```
def validate_email|
```

LLM



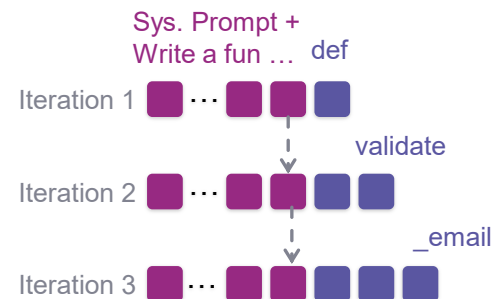
Output: most likely next token

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

Inputs

- Prompt
- Context

text → tokens



Next-Token Prediction

What powers your AI tools

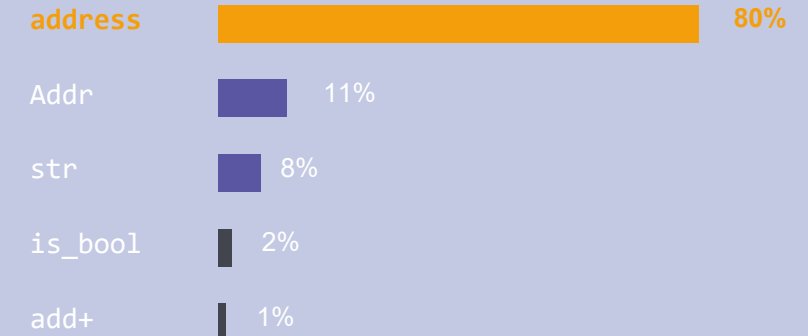
System: You are an expert Python developer.

User: Write a function that validates an email.

Assistant:

```
def validate_email(|
```

LLM



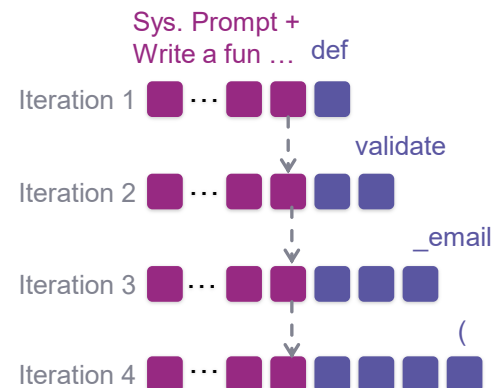
Output: most likely next token

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

Inputs

- Prompt
- Context

text → tokens



Next-Token Prediction

What powers your AI tools

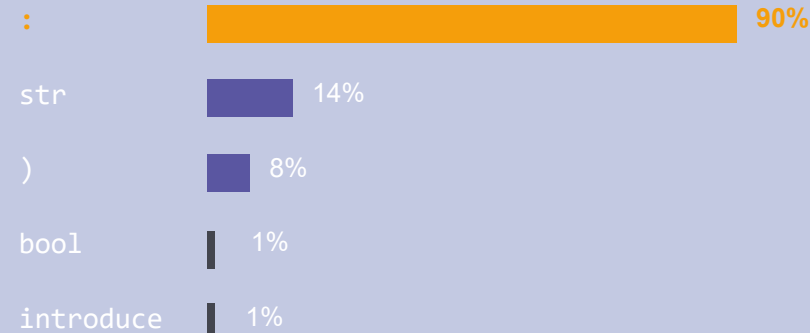
System: You are an expert Python developer.

User: Write a function that validates an email.

Assistant:

```
def validate_email(address
```

LLM



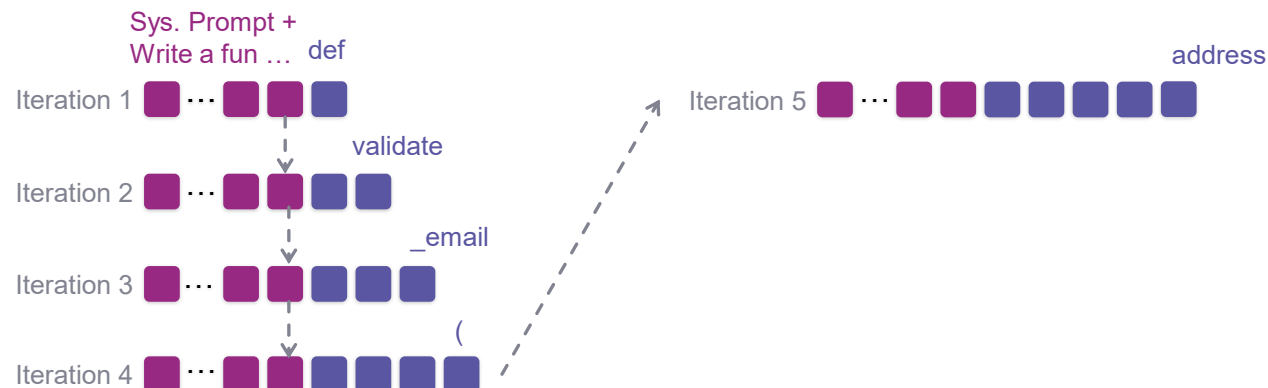
Output: most likely next token

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

Inputs

- Prompt
- Context

text → tokens



Next-Token Prediction

What powers your AI tools

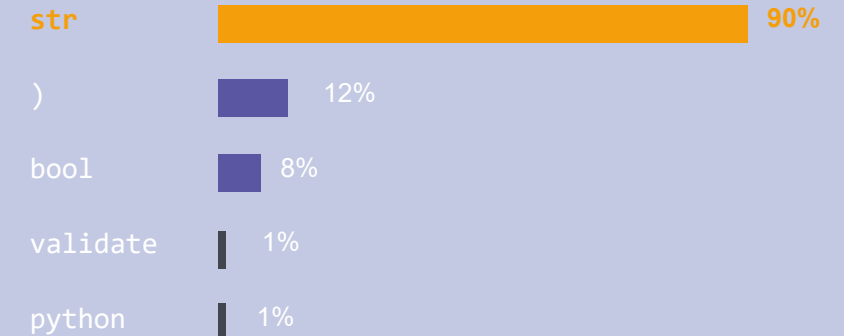
System: You are an expert Python developer.

User: Write a function that validates an email.

Assistant:

```
def validate_email(address: |
```

LLM



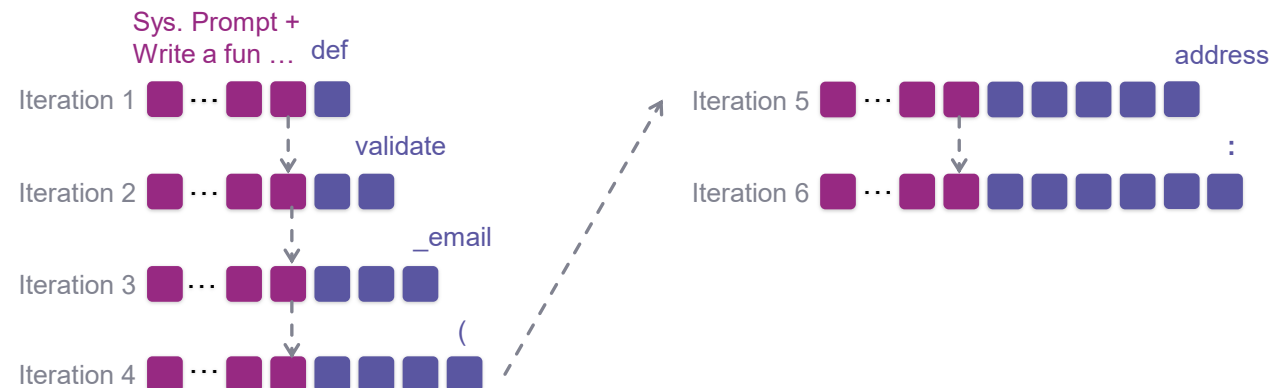
Output: most likely next token

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

Inputs

- Prompt
- Context

text → tokens



Next-Token Prediction

What powers your AI tools

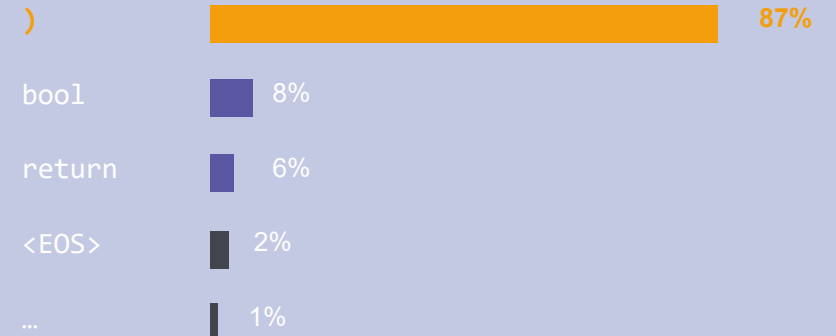
System: You are an expert Python developer.

User: Write a function that validates an email.

Assistant:

```
def validate_email(address: str
```

LLM



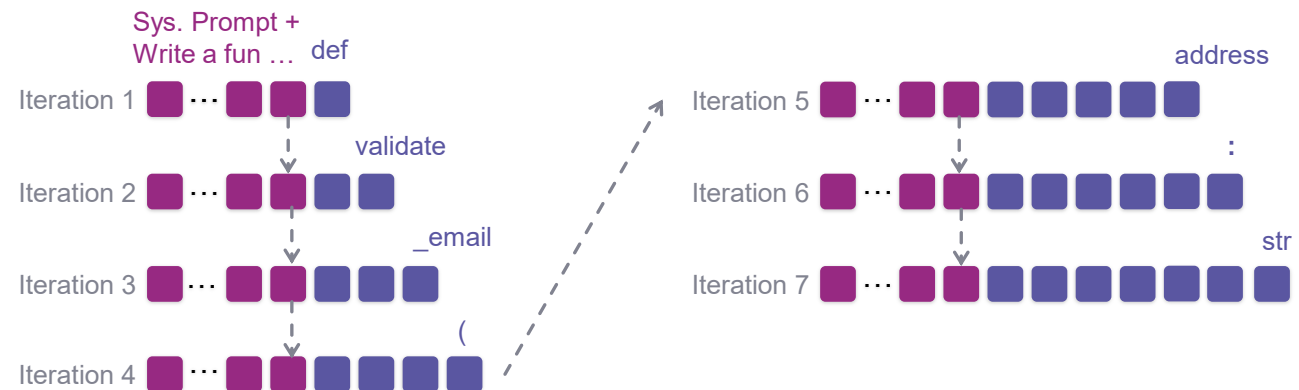
Output: most likely next token

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

Inputs

- Prompt
- Context

text → tokens



Next-Token Prediction

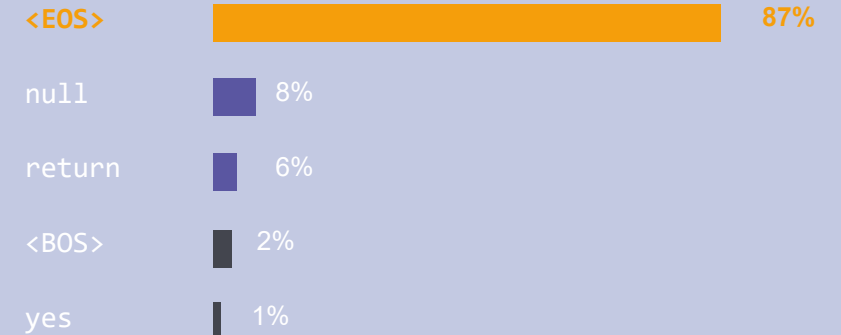
What powers your AI tools

System: You are an expert Python developer.

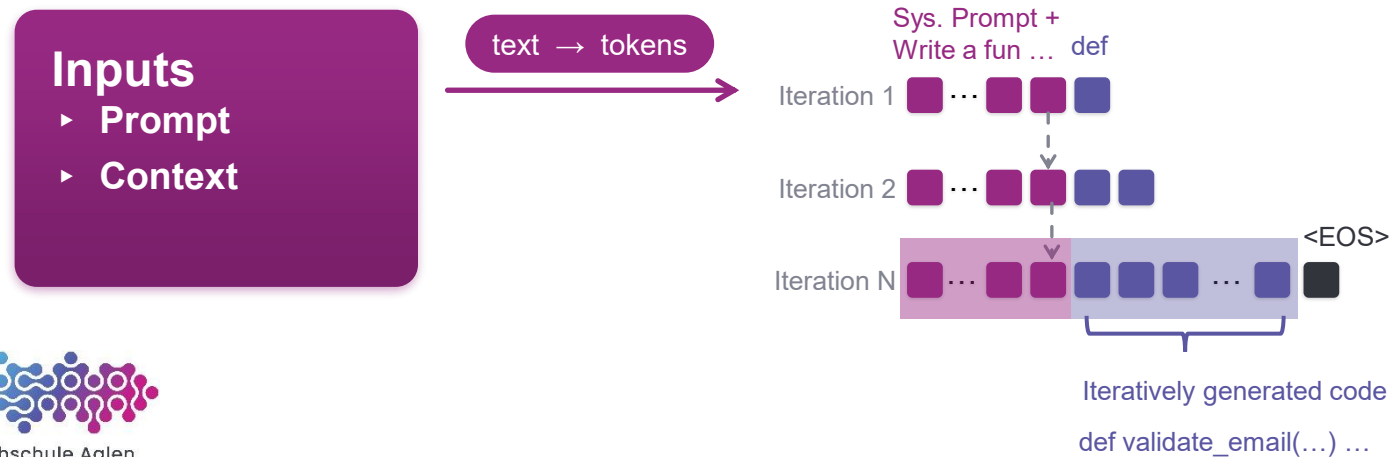
User: Write a function that validates an email.

Assistant:

```
def validate_email(address: str) -> bool:
    """
```



After a special token (<EOS>) was predicted the loop is stopping.



Next-Token Prediction

What powers your AI tools

System: You are an expert Python developer.

User: Write a function that validates an email.

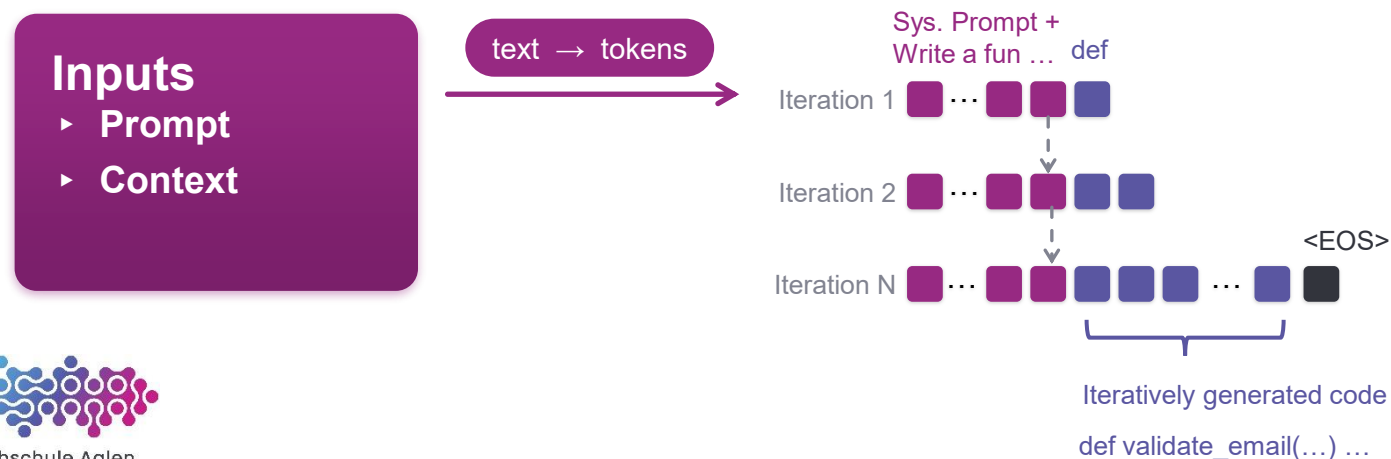
Assistant:

```
def validate_email(address: str)-> bool:
    ``
```

User: Let's directly write a unit test for this function.

Assistant:

New iteration begins on top of all the previous generated tokens.



Next-Token Prediction

What powers your AI tools

System: You are an expert Python developer.

User: Write a function that validates an email.

Assistant:

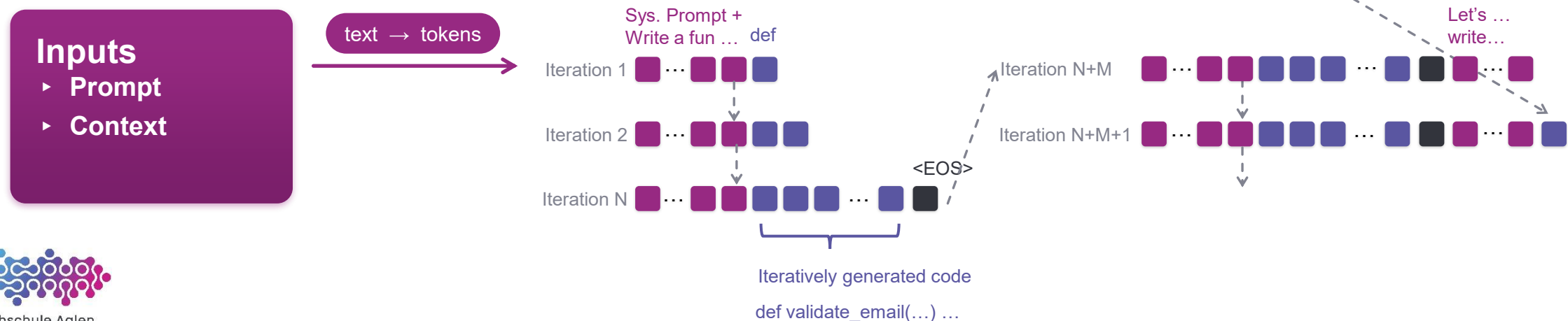
```
def validate_email(address: str)-> bool:
    ``
```

User: Let's directly write a unit test for this function.

Assistant:

Iteratively continue to predict the new tokens ...

New iteration begins on top of all the previous generated tokens.



Code Generation = Text Prediction

What powers your AI tools

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

The unsolved mysteries of Act I — tracked on the Mystery Board

We will track these across Lectures 4+ 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)?

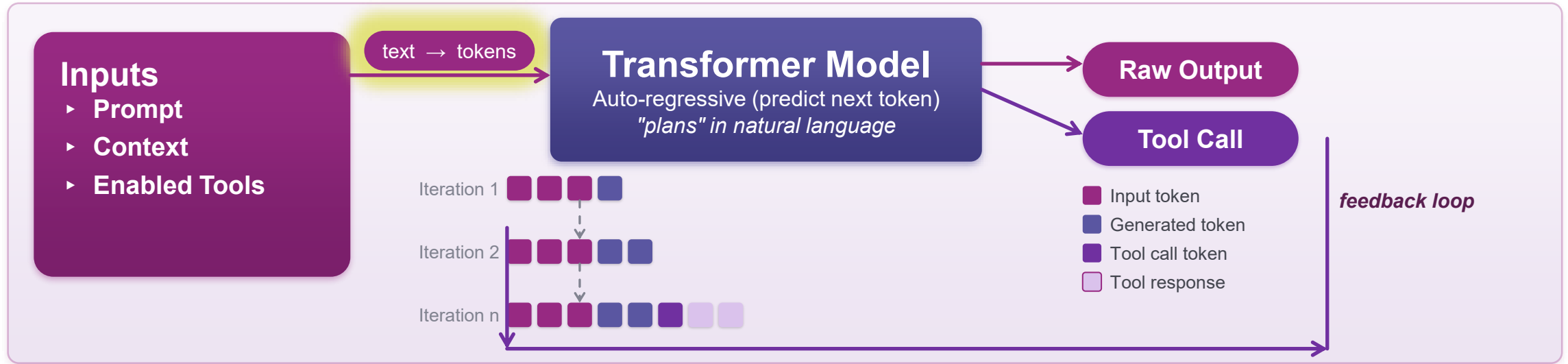
Mystery Board



The LLM in One Picture

2

LLM Architecture Overview



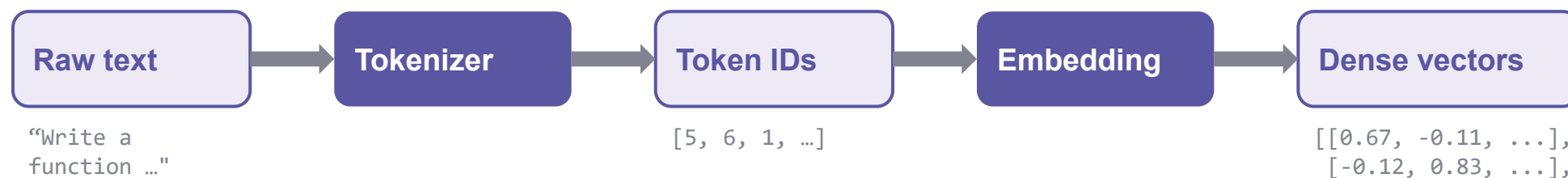
Today: We zoom into the **text → tokens** step.

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

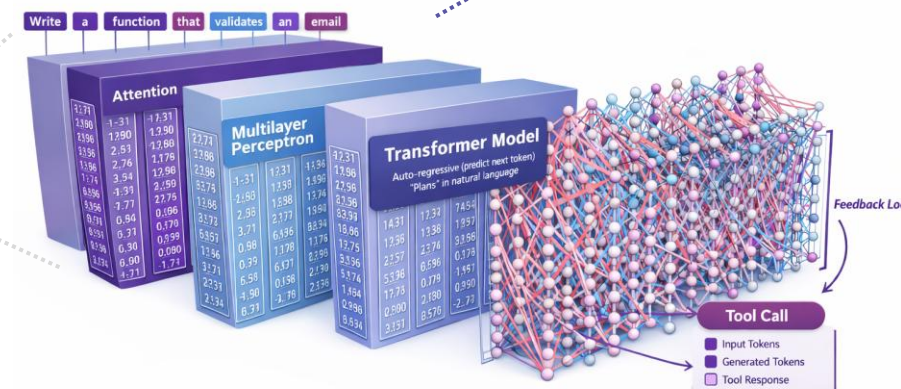
Why Text Must Become Numbers

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



The background is a photograph of a long, modern building with a facade made of vertical wooden slats. The building is situated next to a paved walkway with young trees. Three people are sitting on a low concrete bench in the foreground. The entire image has a blue and purple color overlay. On the right side, there is a large white circle containing a blue number '3'.

The Complete Journey in Miniature

3

A Toy Vocabulary

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

A fantasy world with **V = 10 tokens**:

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

Dragon drinks, wizard mixes, troll smashes.

Here (almost) every token is a full word. Real tokenizers split into **sub-words**.

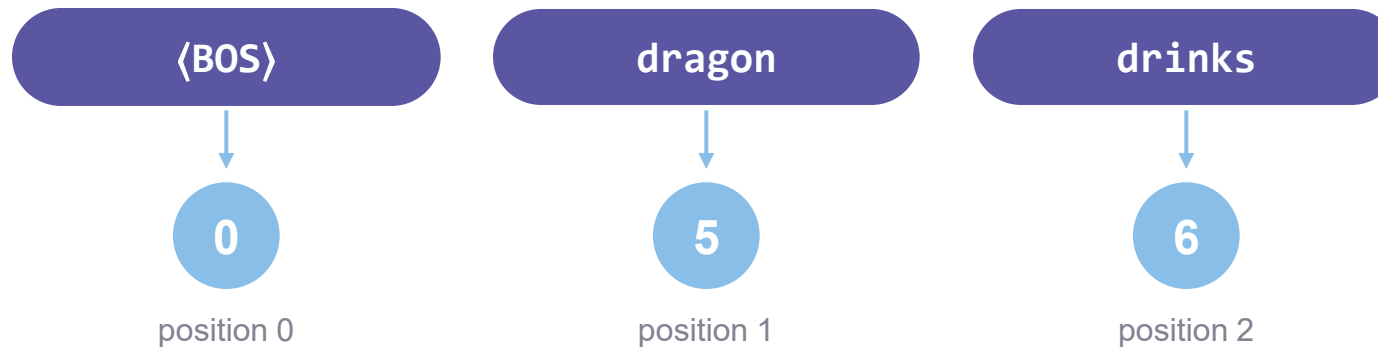
Our example input sentence will be

Input string: <BOS> dragon drinks

Step 1: Text to Token IDs

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

Input string: **<BOS>** dragon drinks



Result: $\mathbf{x} = (0, 5, 6)$

$T = 3$ tokens, $V = 10$ vocabulary size, $\mathbf{x} \in \{0, \dots, 9\}^3 = \{0, \dots, V - 1\}^T$

Vocabulary ($V = 10$)

0: <BOS>

1: wizard

2: mixes

3: glowing

4: potion

5: dragon

6: drinks

7: fire

8: troll

9: smashes

- ▶ Here every token is a full word. Real tokenizers split into **sub-words**.
- ▶ The model only ever sees the **integer IDs**, never the strings themselves.

⚠ These are integers — discrete, not differentiable. No gradients possible yet.

Step 2: The Embedding Table

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

Each token in the vocabulary gets its own row of d learnable numbers

From Before: $\mathbf{x} = (0, 5, 6)$, $T = 3$ tokens, $V = 10$ vocabulary size

Embedding table $E \in \mathbb{R}^{V \times d} = \mathbb{R}^{10 \times 6}$

	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

$d = 6$ columns (embedding dimension)

What is d ?

- ▶ The number of features per token — like (height, weight, age, ...) for a person, but in d dimensions
- ▶ Each dimension captures some abstract aspect of meaning (not human-interpretable)
- ▶ More dimensions = richer, but more parameters

What is E ?

- ▶ A learnable matrix: values are trained via gradient descent
- ▶ Each row is a d -dimensional vector representing one token
- ▶ Similar tokens end up with similar vectors after training

▶ We need rows 0, 5, 6 for our input → highlighted

Step 2: The Lookup Result

The transformer receives a 3×6 matrix

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

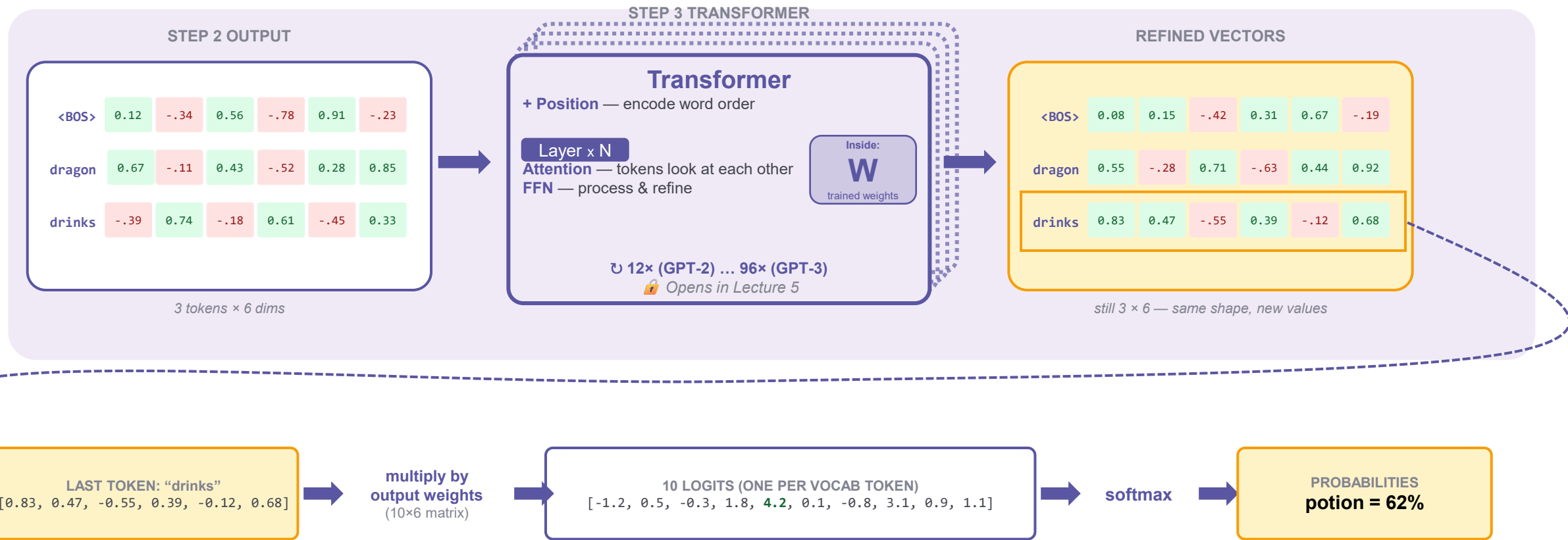
Token	d_1	d_2	d_3	d_4	d_5	d_6
<BOS>	0.12	-0.34	0.56	-0.78	0.91	-0.23
dragon	0.67	-0.11	0.43	-0.52	0.28	0.85
drinks	-0.39	0.74	-0.18	0.61	-0.45	0.33

This is what the transformer receives.

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

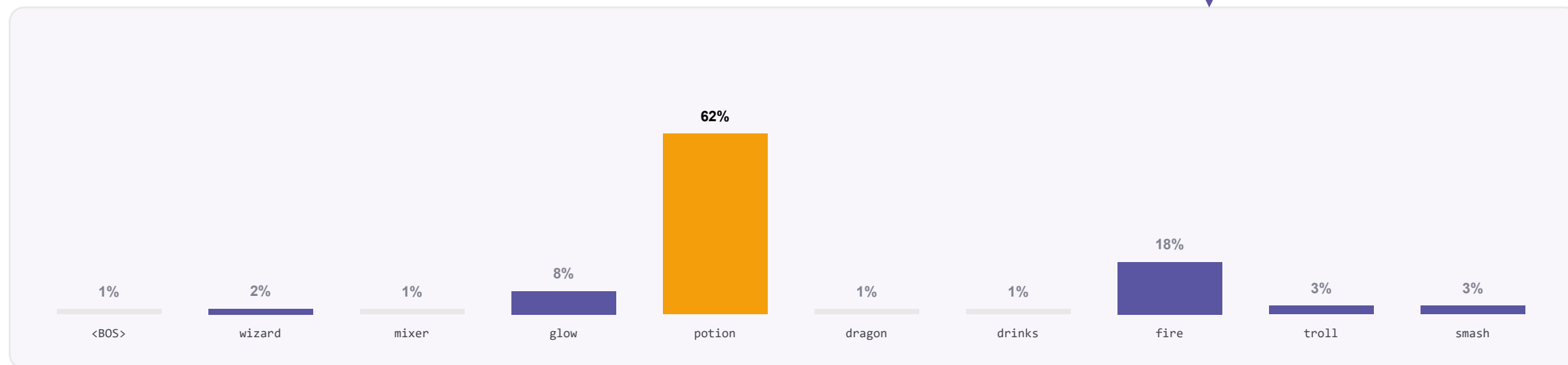
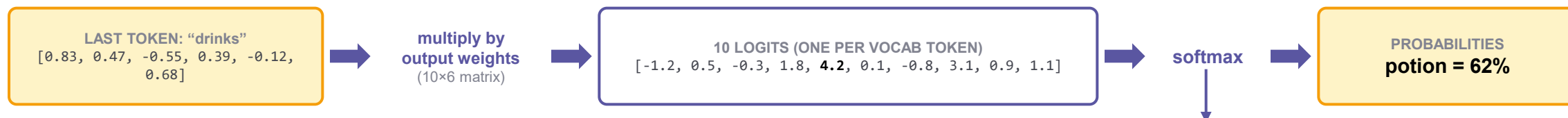
Step 3: The Transformer (Forward Pass)

Embedding vectors enter the transformer and emerge as context-aware representations



Step 4: From the Last Vector to a Prediction

The last token's refined vector is projected onto the vocabulary to predict the next word



Predicted next token: “potion” (62%) — one forward pass complete. The model appends “potion” to the sequence and repeats the entire process to predict the next token.

The Prediction Loop

The mechanism is identical — just at a vastly different scale

<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**.



What the Model Cannot See

4

The Strawberry Puzzle

How many r's are in the word
STRAWBERRY ?

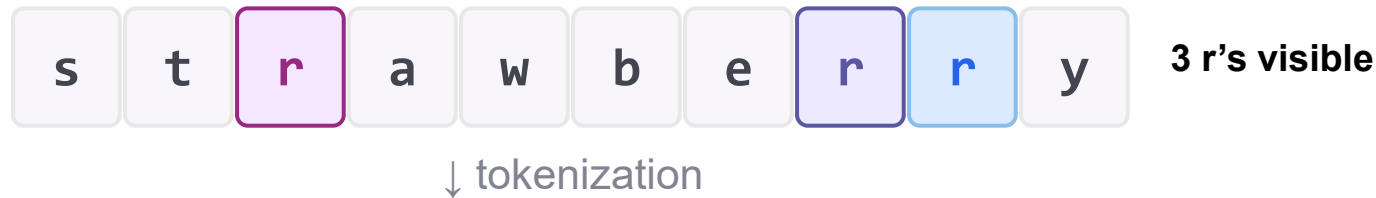
- 2023: every major LLM (GPT-4, Claude, Gemini) confidently answered “**two**” — the correct answer is **three**: strawbe**rry**
- A system that writes working code **could not count to three**
- Today most models get **strawberry** right — **specifically trained** on it
- Underlying problem **not solved**, just papered over
- **Try it:** “How many s’s in **mississippi**?” / “How many c’s in **Connecticut**?”

Why? Next-token predictors fail at counting letters.

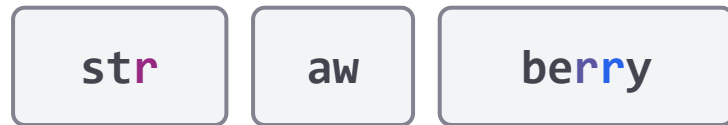
Notebook: Open [§2 in Colab](#) — tokenize strawberry and unbreakable live

Why Models Get It Wrong

What you see — every letter visible:



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



Reason 1 — Tokens are atoms. The model receives opaque vectors, not character sequences. You cannot count what you cannot see.

Reason 2 — Prediction ≠ counting. Even with single-letter tokens, a next-token predictor has no loop or accumulator. It would need to have *memorized* the answer from training data.

How the model actually processes token sequences → Future lectures (Attention)

Try This: 2249 + 2250

Tokenization scrambles digit alignment

$$2249 + 2250 = ?$$

- **Ask your model.** The correct answer is **4499**
- 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

Word-Level vs. Character-Level Tokenization

5

Three Fatal Flaws

Why not just use one token per word?

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

1. Vocabulary explosion

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

2. The unknown word problem

- Unseen word → **[UNK]** → meaning completely lost

3. No morphological awareness

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

The Character Extreme

One token per character — why the other extreme fails:

1. Tiny vocabulary (~100 entries)

- Letters, digits, punctuation — no unknown words ever

But sequences explode

- “cat” = 3 tokens, “unbreakable” = **11 tokens**
- 500-word paragraph → **~2,500 character tokens**

Neither extreme works.
Character-level trades
vocabulary explosion for
sequence explosion.

2. Attention scales quadratically (N^2)

- 2,500 tokens: 6.25M operations
- 500 tokens: 250K operations — a **25× difference** for the same text



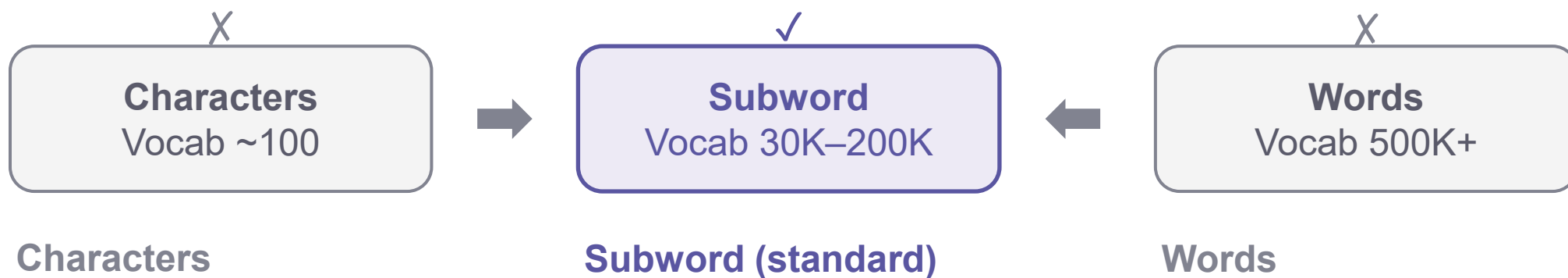


The Subword Sweet Spot

6

Characters vs Subwords vs Words

The tokenization spectrum — finding the sweet spot



- Vocab: ~100
- “cat” = 3 tokens
- Sequences explode
- Attention cost: $4N^2$

- Vocab: 30K–200K
- “unbreakable” = 3 tokens
- Reusable building blocks
- GPT, Claude, Llama

- Vocab: 500K+
- Unknown → [UNK]
- No morphology
- Embedding: huge

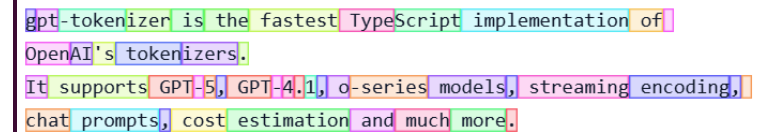
Subword tokenization: keep common words whole, decompose rare words into reusable pieces.

The Subword Challenge: Two Problems to Solve

At the extremes, encoding was trivial

- Characters: every character **is** a token — no splitting needed
- Words: split on spaces — each word maps 1:1 to a token
- In both cases, the encoding function was **obvious**

**Tokenization =
Vocabulary + Encoding Rules**

A screenshot of the GPT Tokenizer Playground interface. It shows a text input area with the sentence: "gpt-tokenizer is the fastest TypeScript implementation of OpenAI's tokenizers. It supports GPT-5, GPT-4.1, o-series models, streaming encoding, chat prompts, cost estimation and much more." The text is displayed with colored boxes around each token, illustrating how the tokenizer splits words and handles punctuation and special tokens. For example, "gpt-tokenizer" is split into "gpt" and "tokenizer", and "TypeScript" is split into "Type" and "Script".

[GPT Tokenizer Playground – Visualize Tokens for OpenAI Models](#)

Subwords change everything — now we need to solve two problems:

- ① **What** should be in the vocabulary? Which subwords do we pick?
- ② **How** do we encode text into those pieces? What's the splitting rule?

Why ② matters

- A subword vocabulary **contains all characters too**
- Without smart encoding, the model could just use single characters
→ The subword entries would be **wasted**, and we're back to the character extreme

Finding the Right Pieces

An algorithm that learns subwords from data

We need subwords, but who decides what they are?

- Not **linguists** — too slow, too language-specific
- Not **engineers** — too arbitrary, doesn't scale

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

Let the **data decide**: merge characters that **co-occur frequently**

This is exactly what **Byte Pair Encoding** does

- Originally a **compression algorithm** (Gage, 1994)
- Repurposed for NLP (Sennrich et al., 2016)

Byte Pair Encoding

7

BPE Step 0: Initialize

Take a sample text (in practice: the full training data of the LLM):

“the cat in the hat”

Split every word into **individual characters** — the smallest possible tokens:

t h e · c a t · i n · t h e · h a t

In real tokenizers, spaces become part of tokens (e.g. `_the`). A pre-tokenization step splits text into chunks first, so BPE only merges within chunks — never across word boundaries. We simplify here by working within words directly.

Initial vocabulary: {t, h, e, c, a, i, n} — **7 tokens**

Goal: grow this vocabulary by discovering useful subwords.

How? Repeatedly find the **most frequent adjacent pair** and merge it into one new token

BPE Step 1: First Merge

Count every adjacent pair. Most frequent: **t** + **h** (appears **2×**) → merge into **th**

Replace every 't h' in the sequence with the new token **th**:

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

Merge rule #1: **t** + **h** → **th**

Vocabulary: {t, h, e, c, a, i, n, **th**} — **8 tokens** (grew by 1)

Note: 'h a t' keeps its standalone h — we only merged t+h, not h+a.

BPE Steps 2-3: Words Emerge

Whole words emerge from pure frequency statistics

Step 2: most frequent pair now is a + t (2×) → merge into **at**

th e · c **at** · i n · th e · h **at**

Merge rule #2: a + t → at — Vocab: 9 tokens

Step 3: most frequent pair now is th + e (2×) → merge into **the**

the · c at · i n · **the** · h at

Merge rule #3: th + e → the — Vocab: 10 tokens

Result after 3 merges: a whole word (“the”) emerged from pure statistics.

We could keep going (merge h+at, c+at, ...) until we hit a **target vocabulary size**.

Why the Merge Order Matters

Phase 1 — Training (what we did on slides 44–46)

- Scan a corpus, count pairs, merge the most frequent, repeat
- Output: an **ordered list of merge rules** (#1, #2, #3 ...)

Phase 2 — Encoding (every time GPT reads new text)

- Start from characters, then **replay merge rules in the order learned**
- Rule #1 fires first, then #2, then #3 — order = priority

Example: encoding “the” with our 3 rules

- Start: t h e (3 character tokens)
- Rule #1 t+h→th: **th** e (2 tokens)
- Rule #2 a+t→at: no match — skip
- Rule #3 th+e→the: **the** (1 token ✓)

“the” = th + e → the,
not t + h + e.
Order is everything.

Tokenization = Vocabulary + Encoding Rules

After training we have:

- Vocabulary **V** = {t, h, e, c, a, i, n, **th**, **at**, **the**} (10 tokens)
- Merge rules #1 t+h→th #2 a+t→at #3 th+e→the

Now encode “the hat” (BPE works within each word separately)

- Start: t h e h a t ← 6 character tokens
- Rule #1 t+h→th: **th** e h a t ← only “the” has t+h
- Rule #2 a+t→at: th e h **at** ← a+t merges in “hat”
- Rule #3 th+e→the: **the** h at ← th+e merges; h stays alone
- Result: **the h at** ← 3 tokens — no “hat” token exists!

The key insight

- Tokenization = **V** (vocabulary) + **f** (encoding function)
- BPE produces **both**: the vocabulary **and** the ordered merge rules that encode text
- It starts from the **character extreme** and builds up — one merge at a time

Two More Things to Notice

Ties and when to stop

1 Ties are real

- In Step 1, t+h, h+e, and a+t all appeared 2× — a **three-way tie**
- The implementation picks one **deterministically** (e.g. first in scan order)
- A different tiebreaker → slightly different vocabulary
- On real data (billions of words), ties are **extremely rare**

2 The stopping criterion

- Stop when the vocabulary reaches a **target size** — a hyperparameter
- GPT-2: **50,257** tokens | GPT-4: **~100,000** tokens
- Our toy corpus runs out of signal fast — real data never does

BPE at Scale

Same algorithm, hundreds of gigabytes of text

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. No linguistic rules, no dictionary, no language knowledge.

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
世	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.

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	Header
1 byte	0xxxxxxx	7	128 (ASCII)	0... standalone
2 bytes	110xxxxx 10xxxxxx	11	2,048	110... starts 2-byte
3 bytes	1110xxxx 10xxxxxx 10xxxxxx	16	65,536	1110... starts 3-byte
4 bytes	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	21	2,097,152	11110... starts 4-byte

Self-describing leading bits: 10x... continuation.

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

Inspect the bytes yourself in Python

No libraries beyond the standard library — every tool you need is built into the language.

Showing the list of integers is simple: `list("ü".encode("utf-8"))`

Character	Byte pattern	Integers
H	01001000	[72]
ü	11000011 10111100	[195, 188]
世	11100100 10111000 10010110	[228, 184, 150]
😊	11110000 10011111 10011000 10001010	[240, 159, 152, 138]

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**

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.

We are really comparing and merging integers

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

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

BPE does not know these are bytes. It just counts **adjacent integer** pairs.

Merge (195, 188) → 256. Token 256's bytes decode to "ü".

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

BPE just "discovered" the umlaut without being told about Unicode.

We are really comparing and merging integers

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

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

Pair	Count	What it is
(195, 188)	3	the two bytes of ü

Merge (195, 188) → 256. Token 256's bytes decode to "ü".
[77, 256, 108, 108, 101, 114, 32, 77, 256, ...]

Growing
Vocabulary

ID (key)	Recursive Occurrence	Flat Bytes	decoded (utf-8)
256	(195, 188)	b'\xc3\xbc'	ü
257	(77, 256)	b'M\xc3\xbc'	Mü
...	...	...	...

195, 188

Lossless vs. Efficient: Two Different Worries

If English dominates training and **ü** appears rarely, does BPE lose it? 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. Frequent patterns get their own token (cheap); rare patterns stay as base bytes (expensive but correct).

Why a New Emoji Doesn't Require Retraining

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

End-to-end walkthrough:

- 1. Encode** — no merge rule matches; emoji → 4 base-byte tokens. **No UNK, no error, no retrain.**
- 2. LLM process** — embedding rows 240, 159, 170, 153 existed from day one. Forward pass unchanged.
- 3. Generate** — model can emit the 4 bytes in order. Verbatim reproduction works; semantics come from in-context learning.

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

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.

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 (cl100k_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: $200\text{K vocab} \times 12\text{K dim} \times 2\text{B} \approx \mathbf{4.9\ GB}$, before the transformer itself uses a byte.

Where the Data Comes From

"Trained on the internet" hides a real pipeline:

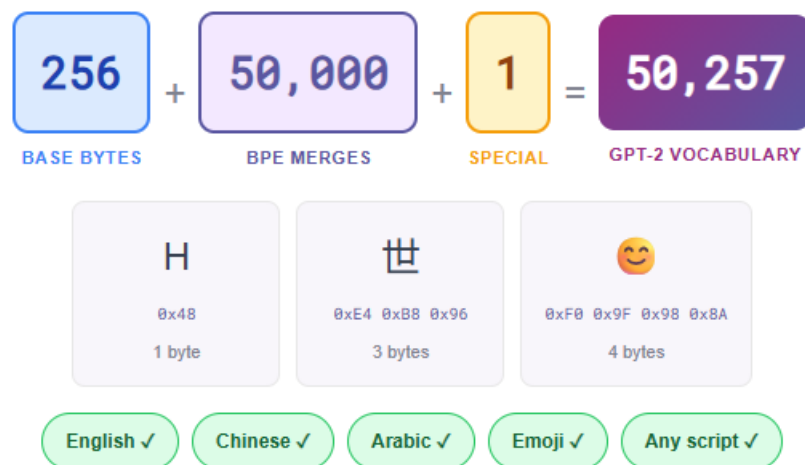
- 1. Extraction** — strip HTML, nav, ads. Keep main text.
- 2. Language detection** — curate the mix (don't take what the crawl happens to sample).
- 3. Quality filtering** — drop spam, low-signal docs.
- 4. Deduplication** — MinHash / SimHash. Duplicates bias merges and cause memorization.
- 5. Mixture 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. Data curation matters more than most people realize.

Tokenization = Vocabulary + Encoding Rules

Vocabulary = 256 Bytes + Number of maximum allowed merges + Special Tokens

Encoding = Merge rules in the order they occurred in the BPE



System	Merges (Hyperparameter)	Final vocab
Our toy BPE	20	276
GPT-2 (2019)	50,000	50,257
GPT-4 (cl100k_base)	~100,000	100,277
GPT-4o (o200k_base)	~200,000	200,019
LLaMA 3	128,000	128,256

Tokenization = Vocabulary + Encoding Rules

<https://tiktokenizer.vercel.app/>

```
Dominik Müller.  
Course about software engineering.  
  
1212 + 99 = ???  
  
Some CODE:  
  
def show_utf8(char):  
    bs = list(char.encode("utf-8"))  
    for i, b in enumerate(bs):  
        print(f" byte {i}: {b:3d} 0x{b:02X} {b:08b}")  
  
show_utf8("H") # 1-byte: 0xxxxxxx  
show_utf8("ü") # 2-byte: 110xxxxx 10xxxxxx
```



```
Dominik Müller.  
Course about software engineering.  
  
1212 + 99 = ???  
  
Some CODE:  
  
def show_utf8(char):  
    bs = list(char.encode("utf-8"))  
    for i, b in enumerate(bs):  
        print(f" byte {i}: {b:3d} 0x{b:02X} {b:08b}")  
  
show_utf8("H") # 1-byte: 0xxxxxxx  
show_utf8("ü") # 2-byte: 110xxxxx 10xxxxxx
```

Vocabulary = GPT-2 (2019) – A map of 50,257 entries (ID, Flat Bytes)

Encoding = Merge rules in the order they occurred in the BPE

43417, 1134, 40790, 6051, 13, 198, 49046, 546, 3788, 8705, 13, 198, 198, 1065, 1065, 1343, 7388, 796, 34913, 198, 198, 4366, 42714, 25, 198, 198, 4299, 905, 62, 40477, 23, 7, 10641, 41, 2599, 198, 220, 220, 220, 275, 82, 796, 1351, 7, 10641, 13, 268, 8189, 7203, 40477, 12, 23, 48774, 198, 220, 220, 20, 329, 1312, 11, 275, 287, 27056, 378, 7, 1443, 2599, 198, 220, 220, 220, 220, 220, 220, 220, 3601, 7, 69, 1, 220, 18022, 1391, 72, 38362, 1391, 65, 25, 18, 67, 92, 220, 657, 87, 90, 65, 25, 2999, 55, 92, 220, 1391, 65, 25, 2919, 65, 92, 4943, 198, 198, 12860, 62, 40477, 23, 7203, 39, 4943, 20, 220, 1303, 352, 12, 26327, 25, 657, 12343, 31811, 198, 12860, 62, 40477, 23, 7203, 9116, 4943, 220, 220, 1303, 362, 12, 26327, 25, 9796, 12343, 87, 838, 12343, 5324, 198

The prompt in the chat window is converted to a list of integers. Each integer representing the index in the vocabulary

So, what, really, is a token?

After all that machinery, we can finally answer the deceptively simple question this chapter opened with:

A token is an integer with meaning assigned by its vocabulary.

The word “token” is also used at three different levels depending on context, all consistent with this definition:

Context	What token means	Example
Tokenizer code	The integer ID	ids = [77, 256, 108] — three tokens
Vocabulary / on disk	The (id → bytes) mapping	vocab.json entry: "token 3788 is b' software'"
Inside the model	The embedding vector for that ID	Row 3788 of the embedding matrix — a vector of ~12,000 floats

Attribution & Sources

Key references for this lecture

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?



Where We Left Off

Recap of Block A and transition to Block B

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?



Vocabulary Size Tradeoffs

8

The Engineering Balance

The fundamental vocabulary size tradeoff

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
- 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

How 4× vocabulary adds 1 billion parameters

LLaMA 2

Vocabulary: 32,000 tokens

$$32,000 \times 4,096$$

= 131M params

→
4× vocab

LLaMA 3

Vocabulary: 128,000 tokens

$$128,000 \times 4,096$$

= 525M params

+ output projection (same shape)

$$525M \times 2 = 1.05B$$

LLaMA 3 (8B) — where do the parameters live?

13%

Rest of model (87%)

4× vocabulary = +1 billion parameters — before the transformer does any computation

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

Why LLMs Fail at Arithmetic

9

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

Digit-level tokenization restores place-value alignment

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.

The Multilingual Fairness Gap

10

The Token Tax

Same concept, different cost across languages

“artificial intelligence” in English: **2 tokens**. 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

Why English prompts outperformed German

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

Glitch Tokens

11

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 = Reddit username (r/counting) → got a token, but model never saw it
- 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

Code Tokenization

12

Indentation Is Expensive

Python's whitespace structure has a token cost

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
- `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**

Naming conventions:

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

Notebook: Open [§6 in Colab](#) — compare indentation cost, naming conventions, and Java vs 18 Python



The Complete Journey

13

6 Stations: Text to Transformer

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

1. **Pre-tokenization** — regex splits text at word boundaries before BPE starts (don't → do | n't). *We haven't covered this step — it simply prevents BPE from merging across words.*
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. *We haven't covered this step yet — it will be part of the next lectures.*

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

Every tokenizer bias carries forward through all layers.



What This Means for CLAUDE.md

14

First Clue to Mystery 1

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

- A 50-line CLAUDE.md ≈ 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.

This is our first clue to Mystery 1 — but the complete answer requires understanding attention and orchestration.

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

Key Takeaways

15

Five Core Principles

1. **Tokenization is the bridge** — text $\rightarrow$ integers $\rightarrow$ vectors $\rightarrow$ 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
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

Mystery Board Update

Tracking our six mysteries across the lecture series

#	Mystery	Status
1	Why did CLAUDE.md improve results?	Partial — it's tokens, but how 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
5	Vague prompts → plausible nonsense?	Open
6	CLAUDE.md inconsistent?	Open

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

Mystery Board

Lab: Tokenizer Playground

16

Notebooks and Deliverables

Hands-on lab work and AI Diary submission requirements

Tokenizer Playground — we explored §2–§8 live during the lecture.

Complete **§9 (Your Findings)** as your deliverable.

Build BPE from Scratch (self-study) — [available in JupyterLite](#)

Deliverables (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?

