

You're reading for free via [Tushar Kanjariya's](#) Friend Link. [Become a member](#) to access the best of Medium.

★ Member-only story

JavaScript + Artificial Intelligence + Machine Learning + Programming +
Web Development +

I Finally Understood LLMs Using JavaScript

Tokenization, embeddings, attention, and text generation rebuilt in runnable JS code.

8 min read · May 18, 2026



Tushar Kanjariya

Follow

Listen

Share

Every time I tried learning how LLMs work, I ran into the same problem.

A blog post with a neural network diagram that looked like a plate of spaghetti Or a Jupyter notebook in Python full of NumPy array operations that I had to mentally translate before I could even follow the idea.

I'm a JavaScript developer.

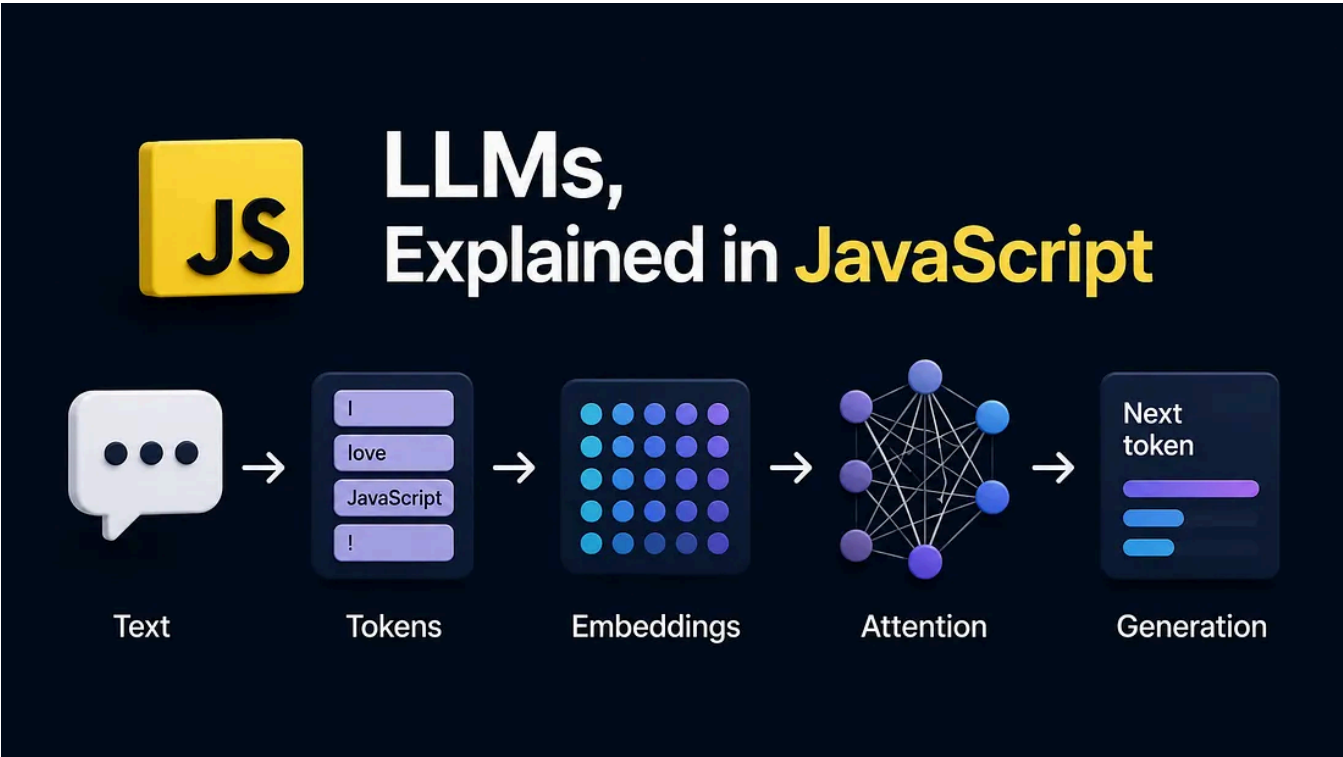
Read Free for non-members.

When I see giant NumPy matrix operations, my brain immediately switches into “translation mode” before I can even understand the actual idea.

I think in JavaScript. So I decided to rebuild each core concept tokenization, embeddings, attention, generation in plain JS.

No libraries. No Python. No linear algebra notation that reads like a ransom note.

This is what I ended up with. It won't train a real model. But it will show you how LLMs work JavaScript-style in code you can actually run, modify, and break on purpose.



How LLMs Work Explained with JavaScript Code

The Big Idea Behind an LLM

Before jumping into code, here's the simplified version of what an LLM actually

[Open in app](#)

[Sign up](#)

[Sign in](#)

Medium

Search



Text
↓
Tokens
↓
Embeddings
↓
Attention
↓
Next-token prediction

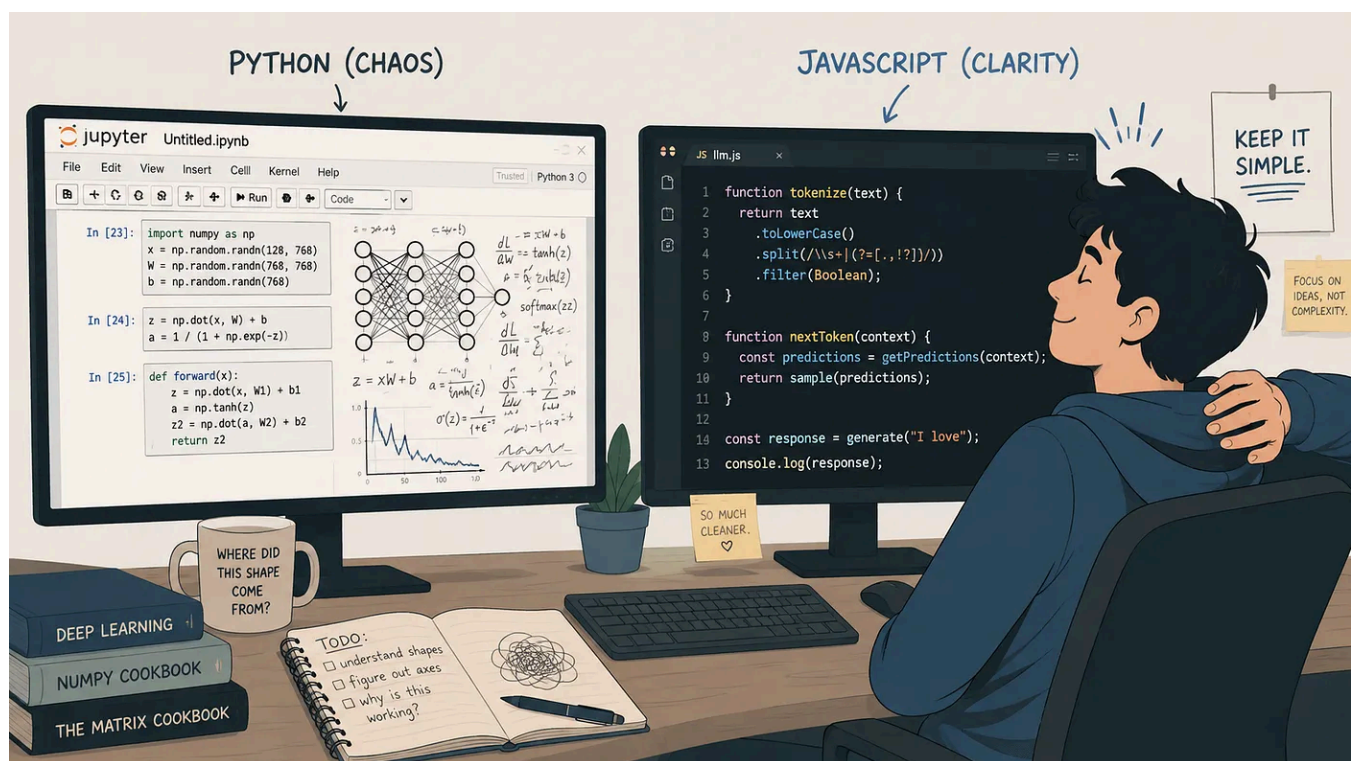
That's it.

An LLM is basically a system that:

1. breaks text into pieces,
2. turns those pieces into numbers,
3. figures out which words matter most,
4. and predicts what should come next.

The scale is massive.

But the core idea is surprisingly understandable once you see it step by step.



Every LLM explainer I found was in Python. So I rewrote the concepts in the language I actually think in.

Step 1: Tokenization — Turning Text into Numbers (IDs)

Computers do not understand words. They understand numbers.

Before an LLM sees a single word, it converts everything to numbers. That process is called tokenization and understanding how LLMs work starts here.

A token is roughly a word, a word-piece, or a punctuation mark.

Real models like GPT use a technique called Byte-Pair Encoding (BPE) which is clever and fairly complex.

For our purposes, splitting on spaces and punctuation gets the idea across.

For example:

```
"I love JavaScript"
```

might become:

```
["I", "love", "JavaScript"]
```

and later:

```
[12, 532, 981]
```

```
function tokenize(text) {
  return text
    .toLowerCase()
    .split(/\s+|(?=[.,!?!])/)
    .filter(Boolean);
}

function buildVocabulary(corpus) {
  const tokens = corpus.flatMap(sentence => tokenize(sentence));
  const unique = [...new Set(tokens)];
  const tokenToId = {};
  const idToToken = {};

  unique.forEach((token, i) => {
    tokenToId[token] = i;
    idToToken[i] = token;
  });
}
```

```
    return { tokenToId, idToToken };
}

const corpus = [
  "the cat sat on the mat",
  "the dog ran in the park",
  "cats and dogs are great"
];

const { tokenToId, idToToken } = buildVocabulary(corpus);

function encode(text) {
  return tokenize(text).map(token => tokenToId[token] ?? -1);
}

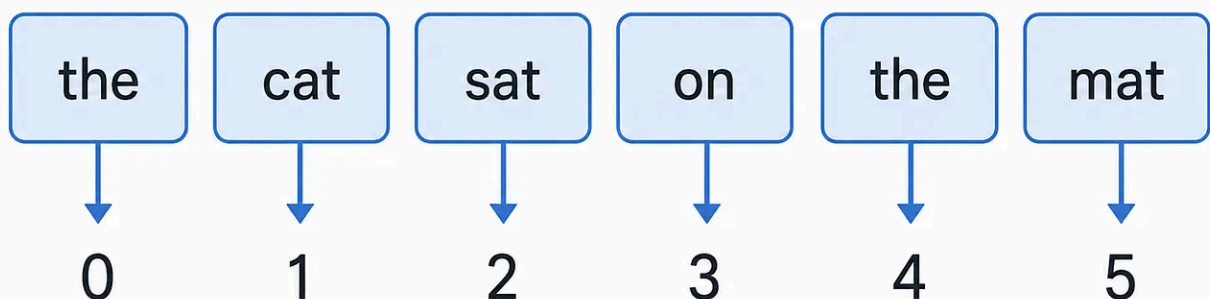
function decode(ids) {
  return ids.map(id => idToToken[id] ?? '[UNK]').join(' ');
}

console.log(encode("the cat sat")); // [0, 1, 2]
console.log(decode([0, 1, 2]));    // "the cat sat"
```

The [UNK] token unknown is what a model returns when it sees a word it's never encountered.

Real models hit this constantly, which is part of why they sometimes produce confident nonsense about very specific things.

“the cat sat on the mat”



Tokenization: every word becomes a number. That number is the only thing the model ever sees.

Step 2: Embeddings — Giving Words a Position in Space (Giving Words Meaning)

This part confused me for a long time.

A token ID by itself means nothing.

A token ID is just a number. An **embedding** turns that number into a vector a list of floating-point values that represents the token's *meaning* in a high-dimensional space.

The idea is that tokens with similar meanings end up with similar vectors. “King” and “Queen” are close together. “Cat” and “database” are not.

In a real model, these vectors are learned through training billions of examples, enormous compute, months of iteration.

For our demo, we'll generate them randomly. The structure is what matters here, not the values.

```
function createEmbeddingTable(vocabSize, embeddingDim) {
  const table = [];
  for (let i = 0; i < vocabSize; i++) {
    const vector = Array.from({ length: embeddingDim }, () =>
      (Math.random() * 2 - 1) * 0.1
    );
    table.push(vector);
  }
  return table;
}

function getEmbedding(tokenId, embeddingTable) {
  return embeddingTable[tokenId];
}

const EMBEDDING_DIM = 4; // Real models use 768, 1024, or more
const vocabSize = Object.keys(tokenToId).length;
const embeddingTable = createEmbeddingTable(vocabSize, EMBEDDING_DIM);

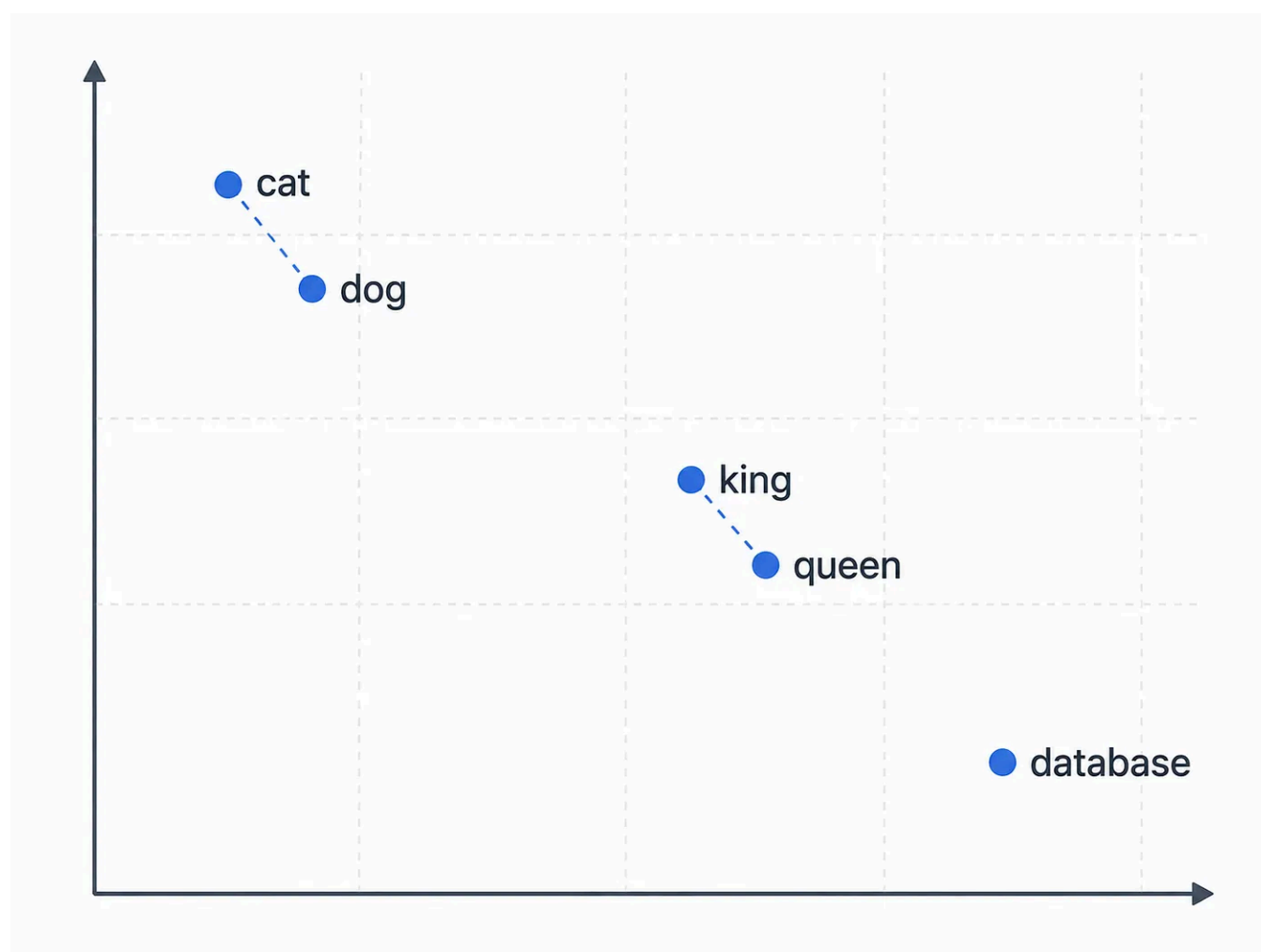
const catEmbedding = getEmbedding(tokenToId['cat'], embeddingTable);
console.log('cat embedding:', catEmbedding);
// Something like: [0.03, -0.07, 0.01, 0.09]
```

The dimension I used here is 4. GPT-3 uses 12,288. That's the same concept, just with enough dimensions to encode remarkably subtle relationships between words and also the reason LLMs take up gigabytes of disk space.

One thing that I found confusing for longer than I'd like to admit:

The embedding table is not the model. It's more like a lookup table the model learns to use.

The actual “intelligence” comes from what the model does *with* those embeddings which is where attention comes in.



Embeddings place words in space. Meaning becomes distance.

Step 3: Attention — How Tokens Notice Each Other

Attention is the mechanism that lets a model figure out which words matter most in relation to other words.

It's the reason "bank" in "river bank" gets interpreted differently from "bank" in "bank account."

The real implementation scaled dot-product attention that involves matrix multiplications with three learned weight matrices called Query, Key, and Value.

Each token asks: *"Which other tokens should I pay attention to right now?"*

For Example: Take this sentence:

The animal didn't cross the street because it was tired.

What does "it" refer to?

The street?

Or the animal?

Humans instantly know it means the animal.

Attention is the mechanism that helps models figure this out.

Here's a simplified version that captures the logic:

```
function dotProduct(a, b) {
  return a.reduce((sum, val, i) => sum + val * b[i], 0);
}

function softmax(scores) {
  const maxScore = Math.max(...scores);
  const exps = scores.map(s => Math.exp(s - maxScore));
  const total = exps.reduce((a, b) => a + b, 0);
  return exps.map(e => e / total);
}

function selfAttention(embeddings) {
  const n = embeddings.length;
  const output = [];

  for (let i = 0; i < n; i++) {
    // Each token scores itself against every other token
    const scores = embeddings.map(other => dotProduct(embeddings[i], other));

    // Softmax turns raw scores into weights that sum to 1
    const weights = softmax(scores);
```

```

// Weighted sum of all token embeddings
const attended = embeddings[0].map((_, dim) =>
  embeddings.reduce((sum, emb, j) => sum + weights[j] * emb[dim], 0)
);

output.push(attended);
}

return output;
}

const tokenIds = encode("the cat sat");
const embeddings = tokenIds.map(id => getEmbedding(id, embeddingTable));
const attended = selfAttention(embeddings);

console.log('attended output:', attended);

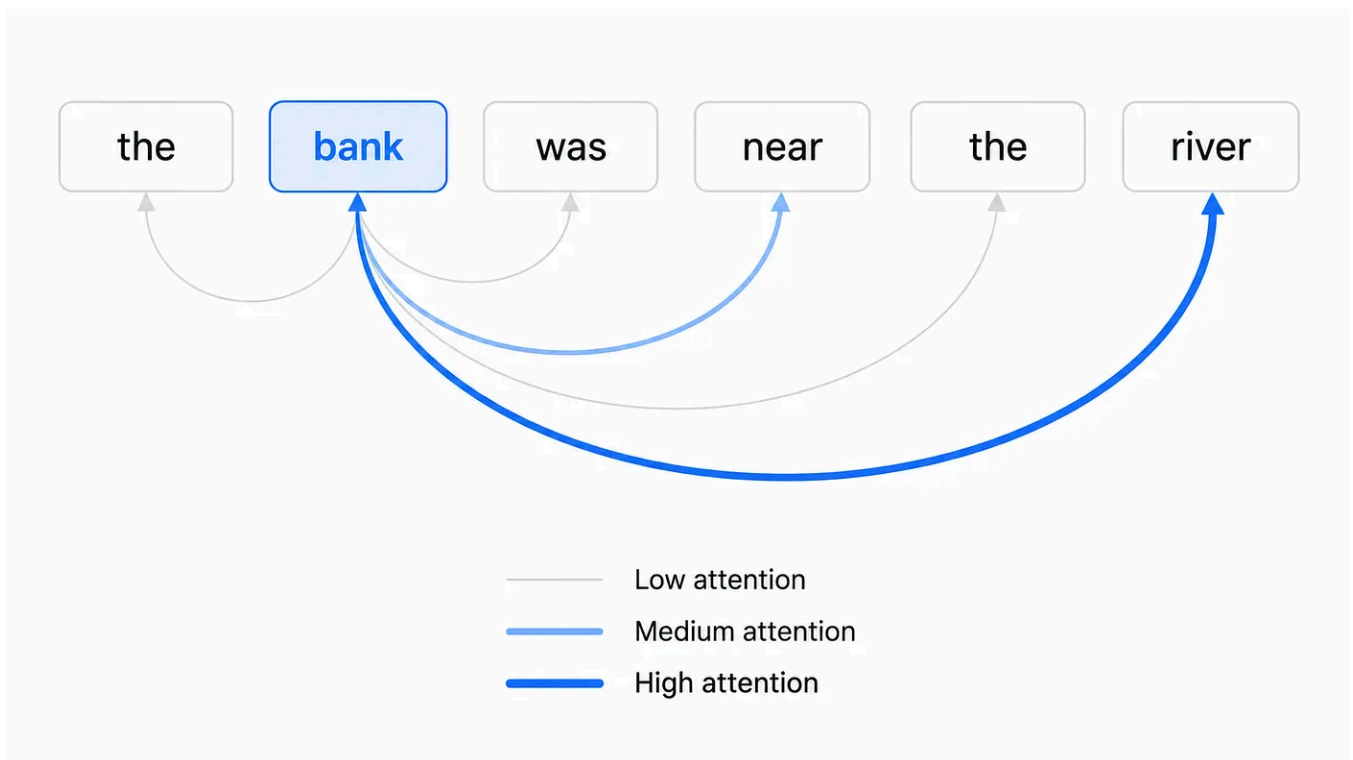
```

The `softmax` function is doing something important:

It turns a pile of raw scores into a probability distribution. Every token gets a weight between 0 and 1, and they all add up to 1. The token with the highest score gets the most “attention.”

Real transformers run this process in parallel across many “heads” multiple attention mechanisms running simultaneously, each potentially learning to track different relationships (syntax, coreference, semantics).

That’s the “multi-head” in multi-head attention. Running it once in our example is enough to see the shape of the idea.



Attention: each token decides which other tokens matter. Thickness = weight.

Step 4: Text Generation — Predicting One Token at a Time

Here's something that surprised me when I first understood it properly: LLMs don't "know" what they're going to say. They predict one token at a time, each prediction conditioned on everything before it.

That's the whole trick. A very large, very well-trained function that says: *"Given this sequence of tokens, what token is most likely to come next?"*

```
function createNgramModel(corpus, n = 2) {  
  const model = {};  
  
  corpus.forEach(sentence => {  
    const tokens = tokenize(sentence);  
    for (let i = 0; i < tokens.length - n; i++) {  
      const context = tokens.slice(i, i + n).join(' ');  
      const next = tokens[i + n];  
      if (!model[context]) model[context] = {};  
      model[context][next] = (model[context][next] || 0) + 1;  
    }  
  });  
  
  return model;  
}
```

```

}

function sampleNext(model, context, temperature = 1.0) {
  const options = model[context];
  if (!options) return null;

  const tokens = Object.keys(options);
  const counts = tokens.map(t => options[t]);

  // Temperature: higher = more random, lower = more deterministic
  const scaled = counts.map(c => Math.pow(c, 1 / temperature));
  const total = scaled.reduce((a, b) => a + b, 0);
  const probs = scaled.map(c => c / total);

  let rand = Math.random();
  for (let i = 0; i < tokens.length; i++) {
    rand -= probs[i];
    if (rand <= 0) return tokens[i];
  }
  return tokens[tokens.length - 1];
}

function generate(model, startTokens, length = 6, temperature = 1.0) {
  const tokens = [...startTokens];

  for (let i = 0; i < length; i++) {
    const context = tokens.slice(-2).join(' ');
    const next = sampleNext(model, context, temperature);
    if (!next) break;
    tokens.push(next);
  }

  return tokens.join(' ');
}

const ngramModel = createNgramModel(corpus, 2);
console.log(generate(ngramModel, ['the', 'cat'], 4));
// "the cat sat on the mat"

```

The **temperature** parameter here is the same concept used in real LLMs when you see “temperature” in an API call.

Low temperature (0.1) = the model picks the highest-probability token almost every time, producing repetitive but consistent output.

High temperature (1.5+) = more randomness, more creativity, more occasional nonsense.

It's a dial between "predictable" and "surprising." Most production use cases sit around 0.7–1.0.

Our n-gram model is obviously primitive compared to a real transformer it only looks at the last 2 tokens as context, while GPT-4 has a context window of 128,000 tokens.

But the core loop is identical: Look at what came before, pick what comes next, repeat.

What We Just Built (and What We Skipped)

Working from top to bottom, we covered:

- **Tokenization** — text to integers
- **Embeddings** — integers to semantic vectors
- **Attention** — vectors noticing each other
- **Generation** — predicting one token at a time with temperature sampling

What real LLMs have that we don't:

- **Learned weights** — our embeddings are random; real ones are trained across trillions of tokens
- **Multi-layer transformers** — real models stack dozens of attention layers on top of each other
- **Positional encoding** — a way to tell the model where in the sequence each token sits
- **Massive scale** — GPT-4 has hundreds of billions of parameters; our model has a few dozen numbers

The gap is enormous.

I'm not pretending this tutorial recreates GPT-4.

But honestly?

Understanding the *shape* of the pipeline changed everything for me.

Research papers stopped looking impossible.

Transformer diagrams stopped feeling mystical.

And AI APIs suddenly made much more sense.

HTMLCSSJSResultEDIT ON

the cat sat on the mat

Try: the dog ran in · cats and dogs · the mat is on

Step 1Tokenization

— text → integer IDs. Click any token to inspect it.

the
id:0

cat
id:1

sat
id:2

on
id:3

the
id:0

mat
id:4

Step 2Embedding

— "sat" as a 6-dim vector (seeded table, same structure as real models)

d0	+0.0628
d1	-0.0668
d2	+0.0375
d3	+0.1028
d4	+0.0847
d5	+0.0603

Resources1×0.5×0.25×Rerun

The Biggest Misconception About LLMs

Before learning this properly, I assumed LLMs “thought” like humans.

They don’t.

They are prediction systems.

Extremely advanced prediction systems.

That distinction matters a lot.

Because once you understand this, many AI behaviors suddenly make sense:

- hallucinations,
- repetition,
- weird confidence,
- random mistakes,

and why prompting style matters so much.

The model is always trying to answer:

“What token is most likely next?”

Conclusion

I spent way too long avoiding transformer articles because they felt overly academic.

Ironically, rebuilding tiny versions in JavaScript made the whole field dramatically easier to understand.

Not because the real systems are simple.

They absolutely are not.

But because the *core ideas* are understandable once you strip away the scale.

And honestly, that was the missing piece for me.

Thanks for reading 🙏

Connect with me 📧

--	--

Tushar Kanjariya | Linktree

Full Stack Developer who occasionally writes

linktr.ee

JavaScript

Artificial Intelligence

Machine Learning

Programming

Web Development



Follow



Written by Tushar Kanjariya

2.6K followers · 186 following

Full-Stack Developer sharing insights on code, design, and AI. 6+ Years of Experience in Tech.

Responses (1)



To respond to this story,
get the free Medium app.

Open in app



Kamrun Nahar she/her

May 19



haha this was exactly what i needed all day honestly!

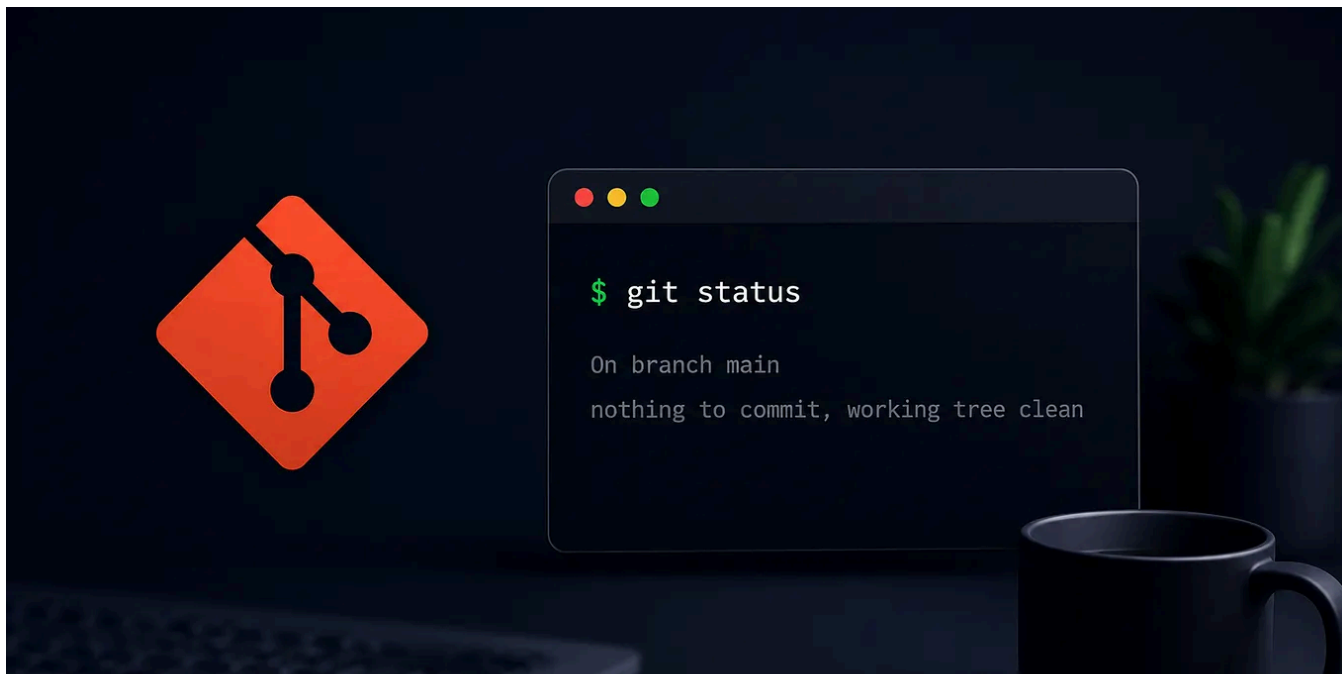


13



1 reply

More from Tushar Kanjariya

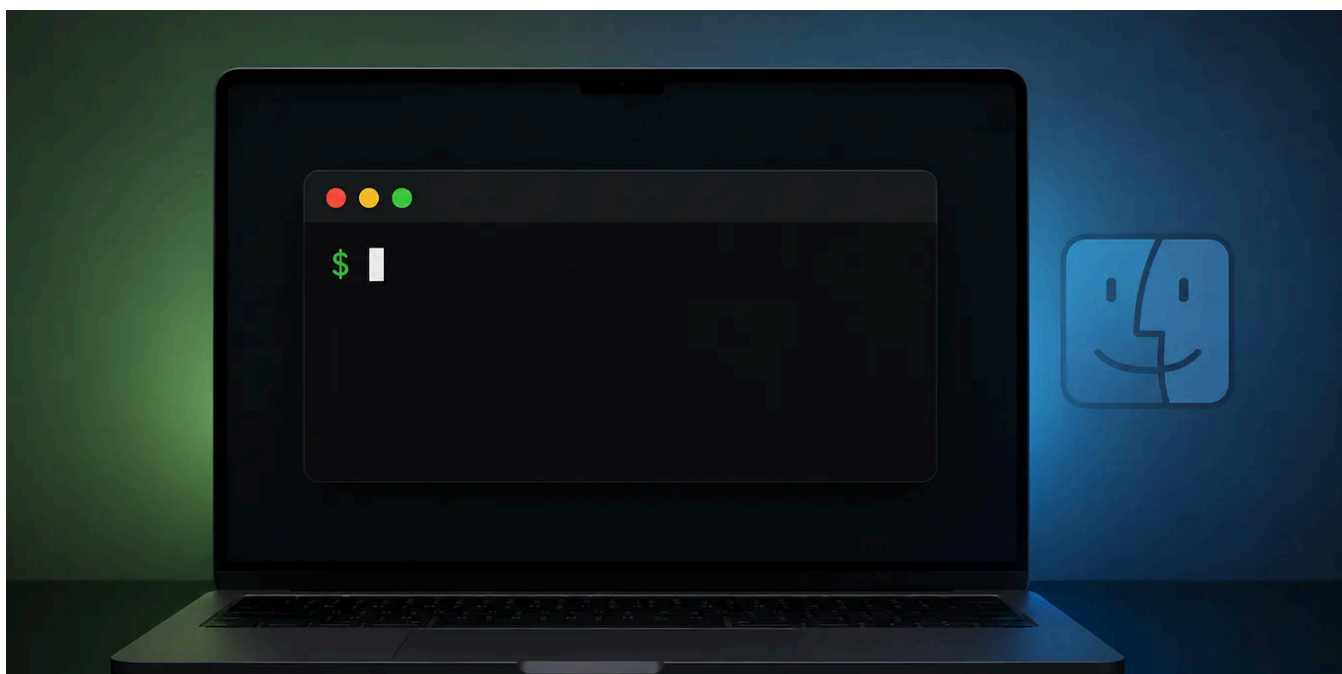


 Tushar Kanjariya · May 8

I Used Git Wrong for Years

What senior developers do differently with Git

★ 🖱️ 2.3K 💬 35 ↺ 34

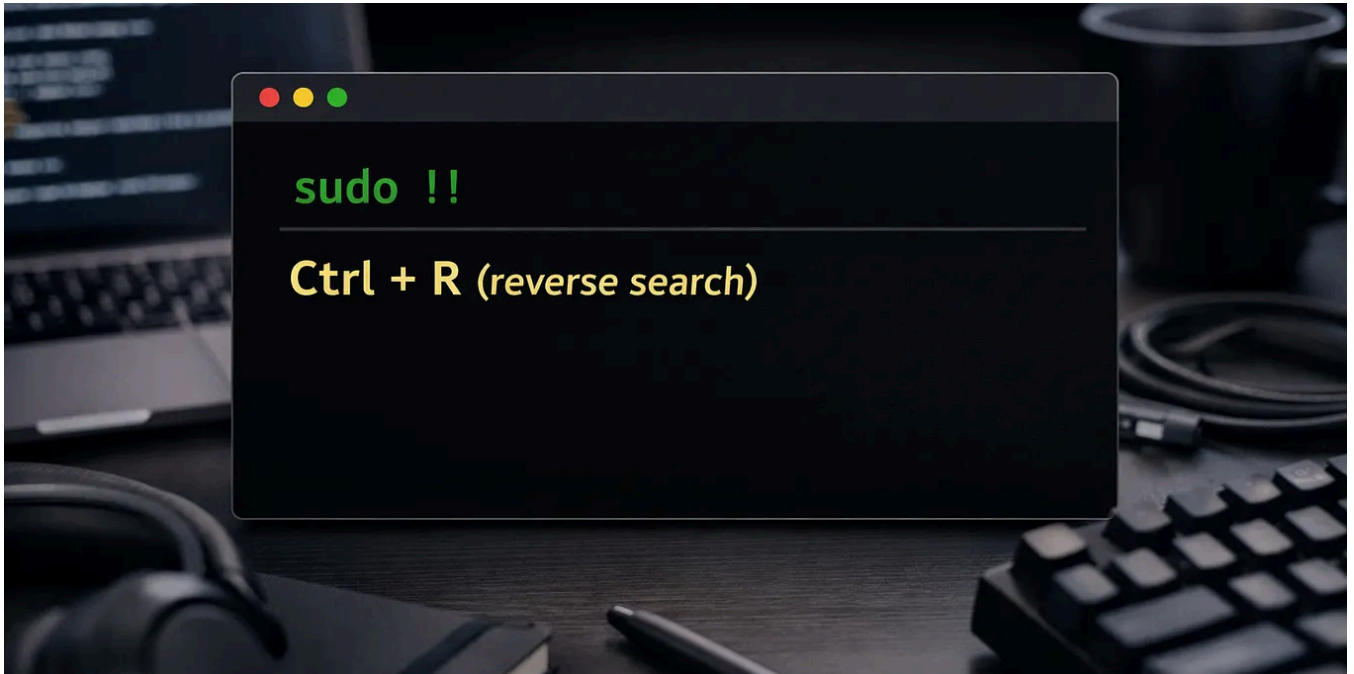


 Tushar Kanjariya · Jun 29

I Ignored These Mac Terminal Commands for Years

What most Mac developers never learn

★ 🖱️ 1.3K 💬 20 ↺ 4

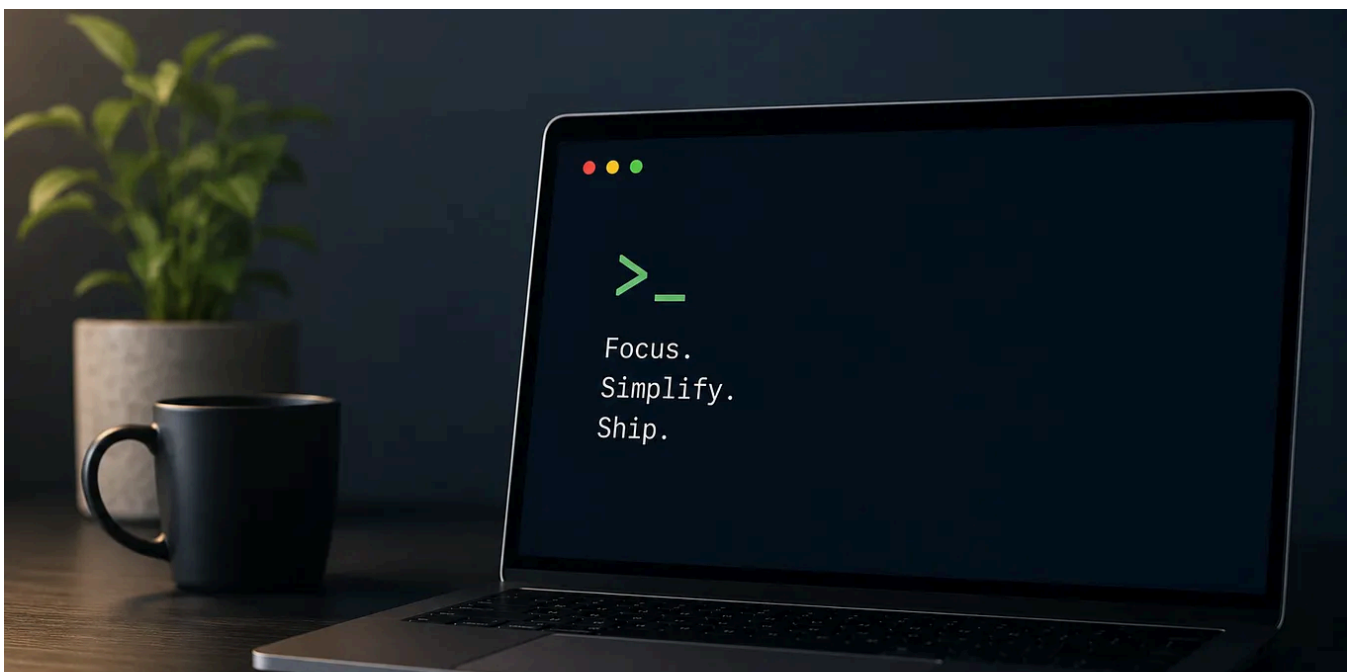


 Tushar Kanjariya · Mar 19

I Used the Terminal Wrong for Years

What senior devs do differently in the terminal

★ 🖱️ 3.3K 💬 66 ↺ 41





Tushar Kanjariya · Jun 9

I Was Optimizing My Terminal All Wrong

After years of tweaking my terminal, these are the only things I still use.



235

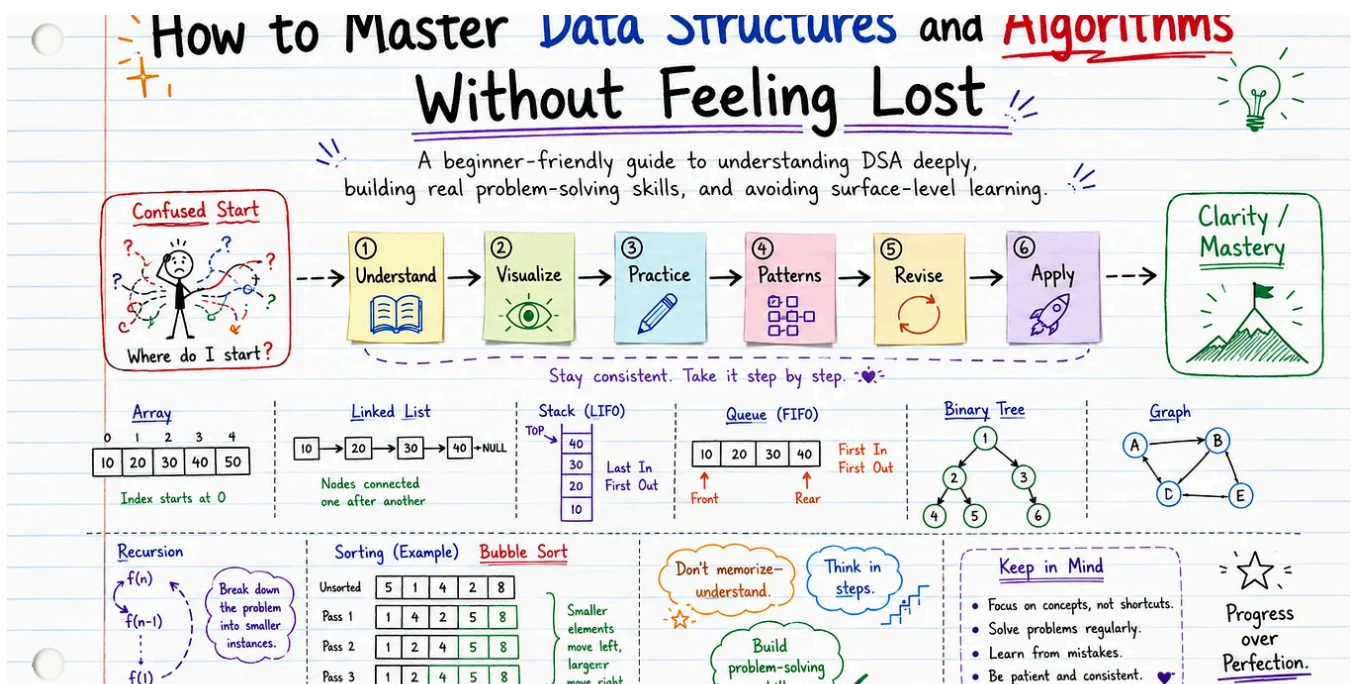


3



See all from Tushar Kanjariya

Recommended from Medium



In Let's Code Future by Deep concept · Jul 3

How to Master Data Structures and Algorithms Without Feeling Lost

A beginner-friendly guide to understanding DSA deeply, building real problem-solving skills, and avoiding the surface-level learning that...

841 6 21



In Skill Stuff by CodeByUmar · Jun 8

7 Coding Patterns I Stole From Senior Engineers

Most developers do not become better because they learn more syntax. They become better because they stop making code harder than it needs...

★ 🖱️ 3.6K 💬 61 ↺ 69



 Michal Malewicz 🌐 · Jul 7

You're not going to like what AI becomes after July 12th.

This is your boiling the frog moment.

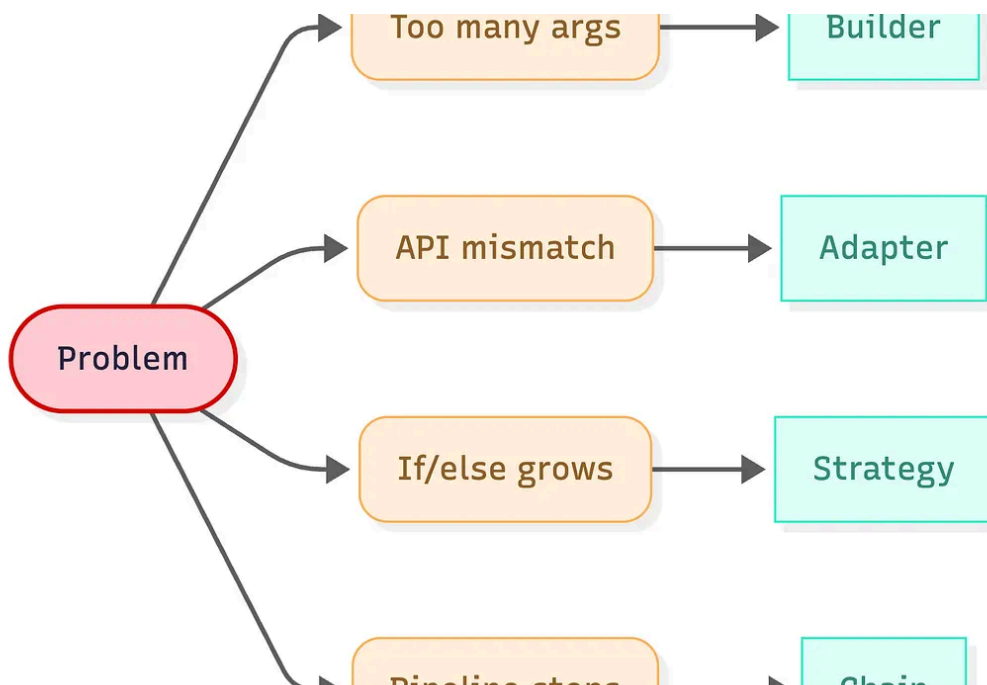
★ 🖱️ 4.3K 💬 167 ↺ 40



 In No Time by Pranit naik · Jun 23

Japan Just Beat Claude Mythos And Nobody Saw It Coming

★ 🖱️ 1.2K 💬 24 ↺ 26



In Women in Technology by Alina Kovtun★ · Jan 29

Stop Memorizing Design Patterns: Use This Decision Tree Instead

Choose design patterns based on pain points: apply the right pattern with minimal over-engineering in any OO language.

★ 🖱️ 9.1K 💬 95 ↺ 77





Should You Still Learn to Code in 2026?

The answer isn't as obvious as I used to believe.



5.5K



287



229



See more recommendations