AI and ML for Web Developers

Version 2026-10 · verified 2026-10-25

What a web developer needs to ship AI features — LLM APIs, streaming, tools, embeddings, RAG, browser inference and security.

Video walkthrough — scan the code to watch.

Contents

Mental model 2

ML vocabulary 2

Tokens, context and cost 3

Calling an LLM API 4

Streaming responses 6

Structured output and tools 7

Embeddings and vector search 8

Retrieval-augmented generation 9

Running models in the browser 10

Security and evaluation 12

Further reading 13

Mental model

A trained model is a function: numbers in, numbers out, with behavior fixed by weights learned from data. An LLM is that function applied over and over to predict the next token, and it remembers nothing between calls — chat history, documents, and tool results exist only because your code puts them back into the prompt each time. Treat its output like user input: probably right, never guaranteed, and never trusted with anything you would not hand a stranger.

ML vocabulary

Term	Meaning
Training	Fitting weights to data — slow, expensive, done once
Inference	Running the fitted model on new input — every request
Parameters	The learned weights; billions for an LLM
Supervised	Learning from labeled examples (spam / not spam)
Fine-tuning	Extra training of an existing model on your data

Almost all web work is inference against someone else's model. Training is how the function got its shape; inference is just arithmetic with the weights frozen, which is why the same model runs on a server, a phone, or in a browser tab.

```
// A "model" with two learned parameters
const w = 2.1, b = 0.4; // fixed after training
```

```
function predict(x: number): number {
  return w * x + b; // inference: plain math
}

predict(10); // 21.4
```

> **Gotcha:** Fine-tuning is rarely how you teach a model your facts. It shapes style and format; for knowledge that changes, put the facts in the prompt instead (see RAG below) — that updates instantly.

Tokens, context and cost

Concept	What it means for you
Token	A word piece; the unit of input, output and billing
Context window	Max tokens per request: prompt plus reply
`max_tokens`	Your cap on reply length; required by some APIs
Input vs output price	Output tokens cost several times more
Latency	Grows with output length, not just input

Models read and write tokens, not characters. English prose runs a bit under one word per token; code, JSON, and many non-English languages use more tokens for the same text. Every request is billed for the whole prompt, so a long chat history costs more on every turn.

```ts
// Rough cost of one request in USD
function cost(
  inTok: number,
  outTok: number,
  inPerM: number,   // $ per 1M input tokens
  outPerM: number,  // $ per 1M output tokens
) {
  const usd = inTok * inPerM + outTok * outPerM;
  return usd / 1e6;
}

cost(20_000, 1_000, 4, 20); // 0.1
```

> **Gotcha:** Do not estimate tokens with a library built for another vendor's model — tokenizers differ, even between versions of one model family. Use the provider's token-counting endpoint, or read `usage` on the response.

Calling an LLM API

Field	Purpose
`model`	Which model; pin an exact ID
`system`	Instructions that frame the whole conversation
`messages`	Alternating `user` / `assistant` turns
`max_tokens`	Reply cap; a cut-off reply says so in `stop_reason`

The API is stateless: to continue a chat you append the assistant's reply and the next user turn, then send the full array again. The reply arrives as an array of content blocks; text is in blocks of `type: "text"`. Always check `stop_reason` — `"max_tokens"` means the answer was cut off.

```
const api = "https://api.anthropic.com";
const res = await fetch(`${api}/v1/messages`, {
  method: "POST",
  headers: {
    "content-type": "application/json",
    "x-api-key": process.env.ANTHROPIC_API_KEY!,
    "anthropic-version": "2026-10-21",
  },
  body: JSON.stringify({
    model: "claude-opus-5-5",
    max_tokens: 1024,
    system: "Answer in one sentence.",
    messages: [{ role: "user", content: "Hi" }],
  }),
});
```

> **Warning:** Never call a model API from browser code with your key. Anything shipped to the client is public; a leaked key is billed to you. Call the API from your server and expose your own endpoint.

Streaming responses

SSE event	Carries
`message_start`	Message ID and model
`content_block_delta`	The next slice of text
`message_delta`	`stop_reason` and output token count
`message_stop`	End of stream

A full reply can take many seconds; streaming shows text as it is generated. Set `"stream": true` and the API answers with Server-Sent Events. Your server can pass the upstream body straight through to the browser, which reads it as a stream.

```
const res = await fetch("/api/chat", {
  method: "POST",
  body: JSON.stringify({ prompt: "Hi" }),
});
const reader = res.body!.getReader();
const decoder = new TextDecoder();
while (true) {
  const { done, value } = await reader.read();
  if (done) break;
  const opts = { stream: true };
  const text = decoder.decode(value, opts);
  console.log(text); // raw SSE text
}
```

> **Gotcha:** Network chunks do not line up with SSE events. One read can hold half an event or three of them. Buffer the text and split on the blank line (`"\n\n"`) before parsing each `data:` line as JSON.

Structured output and tools

Technique	Guarantees
“Reply in JSON” in the prompt	Nothing — usually works
Structured outputs (JSON schema)	Reply matches your schema
Tool use	Model asks you to call a function
Strict tools	Tool arguments match the schema

For data you will parse, ask for a JSON schema rather than politely requesting JSON. Tools go further: you describe functions, the model replies with `stop_reason: "tool_use"` and the arguments, your code runs the function, and you send the result back so the model can continue. The model never executes anything itself — that loop is yours.

```
const tools = [{
  name: "get_weather",
  description: "Current weather for a city",
  input_schema: {
    type: "object",
    properties: { city: { type: "string" } },
    required: ["city"],
  },
```

```
}];
// Reply block: { type: "tool_use", id,
//   name: "get_weather", input: { city } }
// You answer: { type: "tool_result",
//   tool_use_id: id, content: "18°C, cloudy" }
```

> **Gotcha:** A schema guarantee holds only for a completed reply. If `stop_reason` is `"max_tokens"` or `"refusal"`, the output can be truncated or off-schema — check it before `JSON.parse`.

Embeddings and vector search

Term	Meaning
Embedding	Fixed-length vector that encodes meaning
Cosine similarity	1 = same direction, 0 = unrelated
Vector index	Finds nearest vectors fast (approximate)
pgvector `<=>`	Cosine distance in PostgreSQL

An embedding model (separate from the chat model) turns text into a vector. Texts with similar meaning land close together, so “reset my password” finds “forgot login” with no shared words. That makes embeddings the base of semantic search, deduplication, and recommendations.

```
function cosine(a: number[], b: number[]) {
  let dot = 0, na = 0, nb = 0;
```

```
  for (let i = 0; i < a.length; i++) {
    dot += a[i] * b[i];
    na += a[i] ** 2;
    nb += b[i] ** 2;
  }
  return dot / Math.sqrt(na * nb);
}

cosine([1, 0], [2, 0]); // 1 — same meaning
cosine([1, 0], [0, 3]); // 0 — unrelated
```

> **Gotcha:** Vectors from different embedding models — or versions of one — are not comparable. Changing the model means re-embedding every stored document, not only new ones.

Retrieval-augmented generation

Step	Does
Chunk	Split documents into passages of a few hundred tokens
Embed and store	Vector per chunk, saved with its source
Retrieve	Embed the question, fetch the top-k chunks
Generate	Put the chunks in the prompt and ask

RAG answers questions about your data without training anything: you look up relevant passages per request and hand them to the model. Tell

it to use only those passages and to say when the answer is missing, then show the sources so users can check.

```
function buildPrompt(q: string, docs: string[]) {
  const ctx = docs
    .map((c, i) => `<doc id="${i}">${c}</doc>`)
    .join("\n");
  return [
    "Answer using only the documents below.",
    "If they do not contain the answer, say so.",
    ctx,
    `Question: ${q}`,
  ].join("\n\n");
}
```

> **Tip:** Most bad RAG answers are retrieval failures, not model failures. Log the retrieved chunks for each question and check them first; better chunking or hybrid keyword + vector search usually fixes more than a bigger model.

Running models in the browser

Option	Use for
Transformers.js	Hugging Face models via ONNX, WASM or WebGPU

Option	Use for
ONNX Runtime Web	Any ONNX model, lower level
WebGPU	GPU access that makes in-browser inference fast
Chrome built-in AI	`Summarizer`, `Translator` on an on-device model

Small models — classification, embeddings, speech-to-text — run well on the client: no per-request cost, works offline, and data never leaves the device. Transformers.js runs on CPU (WASM) by default; pass `device: "webgpu"` for the GPU and a quantized `dtype` such as `"q4"` to shrink the download. Chrome's built-in `Summarizer` API is stable since Chrome 138; call `Summarizer.availability()` first.

```
import { pipeline }
  from "@huggingface/transformers";

const classify = await pipeline(
  "sentiment-analysis",
  undefined, // default model for the task
  { device: "webgpu", dtype: "q4" },
);

await classify("I love this!");
// [{ label: "POSITIVE", score: 0.99… }]
```

> **Gotcha:** The first call downloads the model weights — often tens to hundreds of megabytes. Load after user intent, show progress, and treat a slow or failed first load as a normal state, not an error.

Security and evaluation

OWASP LLM risk	Defense
LLM01 Prompt injection	Limit what the model can reach and do
LLM02 Sensitive info disclosure	Keep secrets out of prompts
LLM05 Improper output handling	Escape or validate all output
LLM06 Excessive agency	Narrow tools; confirm risky actions
LLM10 Unbounded consumption	Rate-limit and cap tokens per user

Model output is untrusted input: rendering it as HTML is an XSS hole, and passing it to SQL or a shell is injection. For quality, keep an evaluation set — real inputs with the properties a good answer must have — and rerun it on every prompt or model change, because a fix for one case often breaks another.

```
declare const modelText: string;
const out = document.querySelector("#out")!;

out.textContent = modelText; // safe: plain text
// out.innerHTML = modelText; // XSS
```

> **Warning:** Any text the model reads — a web page, an email, a retrieved document — can carry instructions it may follow. A system prompt does not reliably stop this. Assume injection will succeed and limit what a hijacked model could do with its tools.

Further reading

- Claude API — Streaming messages
- Claude API — Tool use overview
- Transformers.js documentation
- Chrome — Built-in AI APIs
- OWASP Top 10 for LLM Applications 2025