

Advanced LLM Concepts for Developers

What Happens Beneath the Prompt (Optional)

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Abstract An optional deep dive into the mechanics behind AI-assisted development: how tokens, context windows, self-attention, and the training pipeline shape what a model can do for you; how embeddings, retrieval-augmented generation, and context engineering feed it the right information; how sampling and reasoning-effort controls change its output; and how to run review, refactoring, testing, documentation, and debugging passes with AI while staying aware of failure modes, licensing, privacy, and academic-integrity obligations.

Table of contents

1	Part 1: LLMs Under the Hood	3		
1.1	What a language model actually does	3		
1.2	Tokens: the unit of everything	3		
1.2.a	Why tokens matter (and why “about 4 characters” is an estimate, not a law)	3		
1.3	Context windows	4		
1.3.a	Long context is not free	4		
1.4	The Transformer and self-attention	4		
1.4.a	Why this matters for code	5		
1.5	How these models were made: the training pipeline	5		
1.5.a	Stage 1 — Pre-training	5		
1.5.b	Stage 2 — Supervised fine-tuning (SFT)	5		
1.5.c	Stage 3 — Preference alignment	5		
1.5.d	Stage 4 — Reasoning training (RL on verifiable rewards)	5		
1.5.e	Stage 5 — Deployed test-time control	6		
1.5.f	A note on coding assistants and “the model behind them”	6		
2	Part 2: Retrieval and Context — Feeding the Model the Right Information	7		
2.1	Embeddings: turning meaning into geometry	7		
2.1.a	Dimensionality is a developer choice	7		
2.2	RAG: Retrieval-Augmented Generation	7		
2.3	Context engineering: the 2026 frame	8		
3	Part 3: Controlling Generation	8		
3.1	Sampling controls: temperature and top-p	8		

3.2 Reasoning effort and verbosity	9
4 Part 4: AI in the Development	
Workflow	10
4.1 Code review: focused passes beat one giant prompt	10
4.2 Refactoring strategies	10
4.3 Testing with AI	11
4.4 Documentation with AI	12
4.5 Debugging with AI	12
5 Part 5: Limits, Failure Modes, and Responsibility	13
5.1 Limitations and failure modes	13
5.2 Ethics: licensing, academic integrity, bias	14
5.3 Privacy and data handling	14
5.4 Agents and oversight — the 2026 present	15
5.5 Habits that keep working	16
5.6 Key takeaways	16
5.7 Going further	17

i Optional advanced material — what this lesson is for

This lesson explains **how LLMs work and why they behave the way they do** — the machinery beneath the workflow you already practice.

Prerequisites: AI Quick Start (webdev_00_L_02_AIQuickStart.qmd) and its practice (webdev_00_P_AIQuickStart.qmd).

When to read: once you are comfortable with the generate → verify → integrate → commit loop and want the “why” behind its rules.

You already have the prompting toolkit. Chain-of-thought, few-shot, multi-turn chaining, role prompting, and system-vs-user prompts live in the L_02 *Advanced prompting* block — they are not repeated here. That block is collapsible in L_02 — open it, since this lesson cites that material repeatedly. This lesson goes **beneath** prompting: the model’s internals, its information supply chain, its generation controls, and the production-grade habits those mechanics justify.

Roadmap — five parts, in reading order, each building on the previous:

- **Part 1** — how the model turns text into finite-budget computation.
- **Part 2** — retrieval and context engineering: it doesn’t know your project.
- **Part 3** — controlling generation: sampling knobs and reasoning effort.
- **Part 4** — the workflow passes where this toolkit pays rent.
- **Part 5** — limits, ethics, privacy, agents, habits that outlast models.

1 Part 1: LLMs Under the Hood

1.1 What a language model actually does

At its core, an LLM is a **probabilistic sequence model** that predicts the next token:

```
P(token_n | token_1, token_2, ..., token_{n-1})
```

Everything — code generation, explanation, review — is this one operation repeated thousands of times.

Example in code generation:

```
// Input: "function add(a, b) { return  
a + "  
// Most likely next token: "b"  
// Less likely: ";", "1", "0"  
// Very unlikely: "elephant"
```

The model assigns probabilities to all possible next tokens and samples from this distribution. Two consequences follow immediately, and they explain most of what you will observe in practice:

- The model has **no internal verifier**. It is not checking your code; it is predicting plausible continuations. Correctness is *your* job — the verify step in the L₀₂ loop exists precisely because of this.
- Output quality tracks **pattern frequency in training data**. Common idioms in popular frameworks come out fluent; rare APIs, your private conventions, or last month's library changes come out plausible-but-wrong.

1.2 Tokens: the unit of everything

Tokens are the **fundamental units** that LLMs process — the strings the model actually sees,

produced by a *tokenizer* that splits text into frequent pieces:

- **English text**: roughly 4 characters per token on average.
- **Code**: more variable — keywords, operators, and identifier fragments each tokenize differently; unfamiliar names split into many small pieces.
- **Special characters**: often their own tokens.

Examples:

```
"Hello, world!" → ["Hello", ",", " ",  
world", "!"] (4 tokens)  
"function calculateTotal()" →  
["function", " calculate", "Total",  
"(", ")"] (5 tokens)
```

1.2.a Why tokens matter (and why “about 4 characters” is an estimate, not a law)

Token counts determine what you can send, what the model can remember in one pass, and what you pay for. But the characters-per-token ratio **drifts across tokenizer generations**: Anthropic documents that its newer tokenizer (Claude Opus 4.7 and later, the generation shared by Claude 5.5, Fable, and Mythos) produces **roughly 30% more tokens for the same input text** than the earlier one.

More tokens per character means **lower character density** — about 2.5–3 characters per token on English text versus the familiar ~4-character rule (a derived estimate, not an official figure). So a 1M-token window on those models holds closer to **2.5–3 million characters than 4 million**: *less* raw text than the naive rule predicts. The ~4-chars/token heuristic stays a fine first estimate for most models, but the drift here runs in one direction — a token buys you **fewer characters** — so estimate, then verify.

! Important

Estimate, then verify. Use “~4 chars/token” to size a problem in your head — never to enforce a budget. Providers publish exact token-counting endpoints (for example the tokenizer APIs and usage endpoints in the OpenAI platform docs and the Anthropic platform docs); when token counts affect cost or a hard limit, measure them with the provider’s own tool.

1.3 Context windows

The **context window** is the total number of tokens (prompt + generated output) a model can attend to in one pass. It is the model’s working memory: anything outside the window simply does not exist for the answer being generated.

Historically, windows went 4K → 8K → 128K → 200K in a few years. **As of late 2026, roughly 1M-token windows are the frontier norm** across leading providers — typically with output capped far lower (around 128K tokens), and with input limits set slightly below a round million depending on the platform.

Two caveats make “big window” a more nuanced story than the raw number:

1.3.a Long context is not free

1. **It costs more.** Providers commonly price above-threshold input at a premium — for example, OpenAI’s GPT-6 Astra bills 2× the input rate and 1.5× the output rate once a request exceeds roughly 272K tokens. A large context window is not a license to paste your entire repository.
2. **Attention degrades with position and length.** The landmark study “Lost in the Middle: How Language Models Use Long Contexts” (Liu et al., 2023) showed that

models perform best on information at the *beginning and end* of a long context, and noticeably worse on information buried in the middle. This finding has aged well and still shapes how tooling assembles prompts.

Practical impact for you:

- Put the critical question or constraint **first or last** in a long prompt, not in the middle of pasted files.
- Curate context deliberately — 10 well-chosen files usually beat 100 dumped ones, even when the window could technically hold all of them.
- Watch the cost curve on very long requests; token count is a bill, not just a limit.

1.4 The Transformer and self-attention

LLMs use the **transformer architecture**, whose core mechanism is **self-attention**: for every token, the model computes how strongly it should “attend” to every other token in the context, at multiple pattern scales simultaneously. This is what lets a single prediction step consider the whole context at once, rather than only the preceding words.

Simplified example:

Input: "The cat sat on the mat"

Self-attention learns:

- "cat" relates strongly to "sat" (subject-verb)
- "sat" relates to "mat" (verb-location)
- "on" connects "sat" and "mat" (preposition relationship)

1.4.a Why this matters for code

Self-attention is exactly the capability that makes code — a highly cross-referential medium — tractable for these models:

- **Variable scope** across lines and blocks: every use of `reviews` can attend back to its declaration.
- **Function calls and arguments** tracked across files in context.
- **Consistency** in naming conventions within a generation.
- **Structural pattern recognition**: “this shape is a React hook”, “this is a builder”, “this try-block has no catch”.

It is also why the context window is the real constraint on code tasks: attention can only relate tokens that are *inside the window*. A symbol defined in a file you never sent cannot be attended to — the model will guess at it, which is one mechanical root cause of hallucinated APIs.

1.5 How these models were made: the training pipeline

Each stage answers a different question. Understanding the pipeline explains behaviors you will observe in daily use.

1.5.a Stage 1 — Pre-training

Question: how does language (and code) work?

The model learns from **massive corpora**: web pages, books, technical writing, and code repositories. The objective is still next-token prediction — but at trillions-of-tokens scale. **Result**: broad knowledge of language, syntax, and code patterns, and a raw completion ability, but no training in being *helpful*. A base model asked “write a login form” might continue with “write a signup form instead” — it completes text, it does not answer questions.

1.5.b Stage 2 — Supervised fine-tuning (SFT)

Question: how do you follow instructions?

The model is trained on **curated instruction–response pairs**: “Write a function that...”, “Explain how...”, high-quality code examples, carefully crafted Q&A. **Result**: the model learns the *shape* of a helpful assistant response.

1.5.c Stage 3 — Preference alignment

Question: which of several good answers is better?

Two families of techniques, both in wide use:

- **RLHF (reinforcement learning from human feedback)**: generate multiple outputs per prompt, have human raters rank them, train a reward model on the rankings, and push the LLM toward highly-ranked outputs.
- **DPO-family (direct preference optimization)**: train directly on the same preference pairs without a separate reward model — the original DPO paper is the reference point. Later variants (group-relative methods such as those described in OpenAI’s reinforcement fine-tuning guide) extend the same idea.

Result: outputs aligned with human preferences for helpfulness, harmlessness, and honesty — and a measurable tendency toward *sounding* confident, because confident phrasing tends to rank better. Keep that in mind every time the model answers a factual question.

1.5.d Stage 4 — Reasoning training (RL on verifiable rewards)

Question: can it think longer when it needs to?

Newer training runs add a stage where the model is reinforced on tasks with **objectively checkable answers** — code that passes tests,

math with a known result. The model learns to produce long internal chains of check, backtrack, and verify before committing to an answer. This is what the current generation of **reasoning models** is.

1.5.e Stage 5 — Deployed test-time control

Question: how much thinking should it spend on this request?

At deployment, providers expose the outcome of Stage 4 as a knob: a **reasoning-effort** or “thinking budget” level (minimal → high). The same model, given the same prompt, will answer differently depending on how much deliberation it is allowed. We return to this in Part 3, because it changes which settings actually matter for you.

i Model landscape snapshot — October 2026

These names age quickly; the mechanics above outlive them. Treat this as a dated photograph, not a map.

- **OpenAI** — the GPT-6 family: Astra (flagship, ~1.05M-token context, 128K output, long-context pricing premium above ~272K), GPT-6.1 Sol, and the lighter Luna. Models listed at <https://platform.openai.com/docs/models>.
- **Anthropic** — the Claude 5.5 line (Fable 5.1, Opus 5.5, Sonnet 5.5, Haiku 5.5): all at 1M-token context / 128K output, with thinking enabled by default (and not disableable on some of them). Overview at <https://platform.claude.com/docs>.
- **Google** — Gemini 3.x, ~1,048K-token input. See <https://ai.google.dev>.
- **Meta (open weights) — Llama 4** (Scout and Maverick) at <https://huggingface.co/meta-llama>. The earlier Code Llama line is officially marked *History* on the Hugging Face org — open code models have consolidated into general multimodal releases.

1.5.f A note on coding assistants and “the model behind them”

The old claim that “GitHub Copilot uses GPT-4-based Codex” is **outdated and no longer how the product works**. In 2026, Copilot is a **multi-model picker** across vendors — OpenAI, Anthropic, Google, Microsoft, Moonshot, xAI models appear in the selector, alongside an **Auto** option that routes your request for you (see the current list in the GitHub Copilot supported-models reference). Note also that “Codex” today names **OpenAI’s**

own coding-agent product line — not a model embedded inside Copilot.

The general lesson: assistant products and foundation models are different layers. A product’s model lineup changes underneath you, weekly; the principles in this lesson are about the model layer and stay valid.

2 Part 2: Retrieval and Context — Feeding the Model the Right Information

2.1 Embeddings: turning meaning into geometry

An **embedding** converts text or code into a vector — an array of numbers — such that *semantically similar* inputs land close together in that space:

```
"authentication" → [0.23, -0.45, 0.67, ...]
"login system"   → [0.21, -0.43, 0.69, ...] (nearby vector!)
"banana recipe"  → [0.89, 0.12, -0.34, ...] (far away)
```

This is what makes **semantic search** possible: a query for “error handling” can match code that says `try/catch`, “exception management”, or “graceful failure recovery” — even when the exact words never appear. Retrieval over your own codebase, documentation, or knowledge sources is built on this geometry.

2.1.a Dimensionality is a developer choice

Embedding vectors have a length — the number of components — and you get more say in that than most tutorials suggest. In OpenAI’s third-generation embedding family, `text-embedding-3-small` defaults to 1536 dimensions and `text-embedding-3-large` to 3072 (the older `ada-002` is legacy), and both

accept a `dimensions` parameter that shortens the vector **without retraining** — the model is trained so that the shorter vectors still capture most of the useful signal. Strikingly, `text-embedding-3-large` truncated to 256 dimensions still outperforms `ada-002` at its full 1536 on standard retrieval benchmarks (MTEB). More dimensions = better recall and more storage; fewer = cheaper and faster. See the OpenAI embeddings guide.

If you want embeddings that never leave your infrastructure, strong open models exist: **BGE-M3** (BAAI, MIT license) produces 1024-dimensional dense vectors over sequences up to 8192 tokens in many languages, and can additionally emit sparse and multi-vector (ColBERT-style) representations from the same model — which is what makes hybrid retrieval possible: combining dense and sparse signals (BGE-M3 model card).

2.2 RAG: Retrieval-Augmented Generation

Base-model weights are frozen at training time. No model natively knows:

- your private codebase or internal APIs,
- last week’s library release,
- your company’s documentation or database contents.

Retrieval-Augmented Generation (RAG) bridges the gap without retraining:

1. **Index** your documents/code as embeddings in a vector store.
2. **Retrieve** the chunks most similar to the user’s query.
3. **Augment** the prompt with those chunks.
4. **Generate** an answer conditioned on that context.

Example workflow:

```
User asks: "How do we handle authentication in our app?"
```

```
System:
```

```
1. Embeds the query; searches the repo index for "authentication"
2. Retrieves: auth.js, login.js, middleware/auth.js
3. Builds prompt: "Given this code: [retrieved files], answer: How do we handle authentication?"
4. LLM answers grounded in YOUR actual code
```

Where you meet RAG without noticing: repository search in **GitHub Copilot Chat**, “code-base question” features in IDE agents, documentation assistants, and frameworks such as LangChain or LlamaIndex — both fine to name as examples of a now-broad ecosystem rather than “the” way to build this.

Why retrieval quality, not model size, usually decides the answer quality: if the retrieve step misses the relevant file, even a perfect model receives a prompt without the needed facts and will improvise — which is experienced downstream as hallucination. When a RAG-backed assistant gives you a wrong answer about your code, the interesting debugging question is “*what chunks did it retrieve?*”, not “*what model is it?*”.

2.3 Context engineering: the 2026 frame

RAG as a four-step pipeline is the concept; **context engineering** is the discipline that grew around it. Anthropic’s guidance on effective context engineering for AI agents (September 2025) — echoed across the industry — treats the context window as a **finite attention budget** rather than free storage:

- **Recall degrades with length (“context rot”).** More tokens in the window doesn’t just cost more; measured performance on retrieving a given fact declines as context grows — consistent with the lost-in-the-middle finding.
- **Just-in-time over pre-loaded.** Rather than stuffing every possibly relevant file up front, effective tooling *retrieves as needed*: give the model search/read tools and let it pull the right chunk at the moment it is needed, keeping the working context small and relevant.
- **Compaction and structured note-taking.** Long sessions get summarized: older turns are compressed into notes or a handoff summary so the live window stays high-signal. (Every long chat tool that “forgets gracefully” is doing this.)
- **Retrieval budgets as stopping rules.** Agentic search needs an explicit budget — “stop after N retrieval steps / when evidence is sufficient” — otherwise the model loops, burning tokens on diminishing returns (this is reflected in provider guidance such as OpenAI’s agent tooling docs).

For you as a developer, the transferable rule: **curate context like a code review, not like a backup.** Every token you add should earn its place.

3 Part 3: Controlling Generation

3.1 Sampling controls: temperature and top-p

Recall Part 1: the model outputs a probability distribution over next tokens. **Sampling** decides how the actual token is picked from that distribution — and this is where the classic knobs live.

⚠ Warning

Scope first, settings second. Temperature and top-p are meaningful controls for **traditional (non-reasoning) sampling models**. Current reasoning-oriented models are steered primarily by different knobs (below). The older advice about keeping temperature low for code applies to classic models — do not assume it carries over, and do not assume these parameters are honored at all on a reasoning endpoint: OpenAI’s own chat API reference warns that “Parameter support can differ depending on the model... particularly for newer reasoning models”. Nothing official says temperature is *unsupported* on reasoning models either — the honest position is **not officially specified: don’t rely on it**.

Temperature rescales the distribution — how peaked versus flat it is:

```
Low (0.0–0.3): near-deterministic,
focused
  "Complete: function add(a, b) {" →
  "return a + b; }" (consistently)

Medium (0.5–0.7): balanced
  mostly the obvious completion,
  occasionally a variant

High (0.8–1.0): diverse, surprising
  "return a + b; }"
  "return [a, b].reduce((x, y) => x
  + y);"
  "if (typeof a !== 'number') throw
  new TypeError(...)"
```

Top-p (nucleus sampling) truncates instead of rescaling: rank tokens by probability, keep only the smallest set whose probabilities sum to p , and sample from that subset. `top_p = 0.9`

discards the long tail of unlikely tokens while allowing variety.

Best practice for classic models: the low-for-code, high-for-brainstorming recipe from L_02 (AI Quick Start) applies to these traditional sampling controls — we are not restating it here. What this lesson adds is a constraint: tune **one knob at a time** (temperature *or* top-p), never both aggressively, because they compound.

3.2 Reasoning effort and verbosity

For models with the Stage-4 training described earlier, the practical control surface shifts:

Control	What it changes	Typical developer use
Reasoning effort (minimal → high)	How much internal de-liberation the model spends before answering	Low effort for renaming, formatting, simple completions; high effort for concurrency bugs, tricky algorithms, architecture trade-offs
Verbosity	How long and explanatory the <i>visible</i> answer is	Terse diffs for routine edits; full explanation when learning or documenting

The mental model: **effort buys thinking; verbosity buys words**. They are independent axes, and both cost tokens — a high-effort request may bill substantially more than its

visible output suggests, because the deliberation is part of what you paid for.

A developer's heuristic: start at the provider's default, then move effort *up* when you see the symptoms of shallow thinking (missed edge cases, an incorrect async-ordering fix) and *down* when simple tasks are slow and over-explained. Which knob your tool exposes — and what it is even called — is product-specific; the two-axis logic is not.

4 Part 4: AI in the Development Workflow

The loop from L_02 — generate → verify → integrate → commit — is the unit of work. This part is about running that loop at **production quality**: the patterns that make AI useful beyond code generation, and the mechanics from Part 1 that explain why each pattern works.

4.1 Code review: focused passes beat one giant prompt

A single “review this code for everything that’s wrong” prompt underperforms a sequence of focused passes. Split the review into distinct passes, one concern each:

Pass 1 — Correctness. Logic errors, edge cases (null, empty, invalid input), off-by-one, async race conditions. Feed the function *and* its call-site contract.

Pass 2 — Security. The checklist from L_02 applies (injection, unsafe HTML sinks, auth gaps, secret exposure); a dedicated pass with *only* this lens finds what the everything-pass misses.

Pass 3 — Performance. This is where an AI pass is genuinely strong, because these issues are pattern-shaped:

Analyze performance of this code:
[paste code]

Identify:

- Big-O issues (nested loops over large collections, repeated scans)
- N+1 query patterns (a query inside a loop that should be one join/batch)
- Unnecessary re-renders and leaked listeners/timers in React
- Serial awaits that could run concurrently
- Database indexes the queries would need at production scale

Each pass returns a candidate findings list; **you** adjudicate it. The passes are cheap; a wrong merge is not.

Why splitting works mechanically: Part 1's lost-in-the-middle result plus Part 2's attention-budget frame — a vague instruction over long context distributes a scarce resource thin. One concern per pass concentrates the model's statistical prior on exactly the patterns you want surfaced — and keeps each findings list small enough that your own verify step stays real rather than ceremonial.

4.2 Refactoring strategies

The step-by-step prompt recipes live in L_02 (magic numbers → extract function → naming → error handling — use the incremental-refactor prompts from there). What belongs here is the engineering rationale that generalizes them:

- **A refactoring is behavior-preserving by definition.** The diff may restructure code but must not change what it does — so AI output is verifiable *only* against tests. No tests → no safe refactoring, AI-assisted or otherwise. Put the test net in place first.

- **Reviewability scales inversely with diff size.** A 200-line “modernize this file” rewrite cannot be meaningfully reviewed, which silently disables your verify step. One intent per diff keeps you able to say what should and should not have changed.
- **Behavior changes belong in separate commits from restructurings.** The classic “renames mixed with logic” review failure gets more expensive when an assistant produces the whole blob in one shot.

Pattern: apply one named transformation, get the diff, run tests, commit. Repeat. The AI accelerates each micro-step; you keep the audit trail.

4.3 Testing with AI

Models are strong test generators *when you supply the coverage dimensions* — what must be tested is a design decision; writing the Jest boilerplate is the mechanical part.

Generate tests for a product search function:

```
function    searchProducts(products,
query) {
    // Returns products matching query
    in name or description
}
```

Cover explicitly:

- Case-insensitive matching
- Partial matches
- Empty query
- Empty product list
- Special characters (regex metacharacters in the query!)
- Multi-word queries
- No matches found

Representative output:

```
describe('searchProducts', () => {
    const products = [
        { name: 'Wireless
Mouse', description: 'Ergonomic
2.4GHz mouse' },
        { name: 'Mechanical
Keyboard', description: 'Tenkeyless,
hot-swappable' }
    ];

    test('matches case-insensitively',
() => {
        expect(searchProducts(products,
'mouse')).toHaveLength(1);
    });

    test('empty query returns all
products', () => {
        expect(searchProducts(products,
'')).toEqual(products);
    });

    test('regex metacharacters are
treated literally', () => {
        expect(searchProducts(products,
'mouse
(ergonomic')).toHaveLength(0);
    });

    // ... one test per coverage
    dimension
});
```

Two habits that follow from “no internal verifier” (Part 1):

1. **The model cannot see your code fail.** It writes tests and implementation from the same statistical prior — so *you* must confirm the tests run, actually fail against wrong code, and don’t encode the same misunderstanding the implementation has.
2. **Prefer test-first ordering** for AI work: generate the coverage list and tests *before*

the implementation. Tests written after code tend to rationalize whatever the code happens to do.

4.4 Documentation with AI

A safe, high-value use, because the source of truth already exists — the model is *transcribing intent from code*, not asserting facts about the world. Verify the claims anyway: a `@param` the signature never uses is a documentation hallucination.

```
/**
 * Fetches product reviews from
 * multiple sources and aggregates
 * ratings.
 *
 * @param {string} productId - Unique
 * identifier for the product
 * @param {Array<string>} sources
 * - Source names, e.g. ['amazon',
 * 'bestbuy']
 * @returns {Promise<Object>}
 * Aggregated review data with average
 * ratings
 * @throws {Error} If productId is
 * invalid or sources is empty
 *
 * @example
 * const reviews = await
 * fetchReviews('prod-123', ['amazon',
 * 'walmart']);
 * console.log(reviews.averageRating); //
 * 4.2
 */
async function
fetchReviews(productId, sources) {
  // ...
}
```

The same pattern scales from doc comments to README sections to changelog drafts: give the model the artifact, ask for the prose, review for truth.

4.5 Debugging with AI

The turn-by-turn conversation pattern is in L_02 (describe → iterate → explain). What matters here is *why it works*: with your error, stack trace, and relevant code in context, the model ranks **plausible explanations by training frequency**. Common React mistakes are common in the data, so they surface first — usually what you want, but it also means the uncommon real cause in *your* code competes badly against the statistically typical bug. Treat outputs as ranked hypotheses: test them, don't accept them.

A full worked example — the classic non-re-render:

```
Prompt:
"My React component isn't re-
rendering when state changes.

function ReviewList() {
  const [reviews, setReviews] =
  useState([]);

  async function fetchReviews() {
    const data = await fetch('/api/
    reviews').then(r => r.json());
    reviews = data;    // component
    never updates
  }

  return <div>{reviews.map(r =>
  <ReviewCard key={r.id} review={r} /
  >)}</div>;
}
What's wrong?"
```

```
AI response (summarized):
"You assigned to the `reviews`
variable directly instead of calling
the
`setReviews` setter. React tracks
```

```
state through the setter; a plain
assignment changes a local variable
and schedules no re-render."
```

The fix, as a real diff:

```
async function fetchReviews() {
  const data = await fetch('/api/
reviews').then(r => r.json());
-   reviews = data;
+   setReviews(data);
}
```

The follow-up a good model adds: in a real component the fetch belongs in a `useEffect` (or a data-fetching library) with cleanup for component-unmounts-mid-fetch — and if a callback keeps *reading* stale `reviews`, that is the related stale-closure class of bug, fixed by functional updates (`setReviews(prev => ...)`) or correct dependency arrays. Notice what made this conversation work at all: the code *and* the observed behavior were in the window. Attention can only reason over what you put in context.

5 Part 5: Limits, Failure Modes, and Responsibility

5.1 Limitations and failure modes

The weights are frozen and there is no internal verifier. Four structural limits follow, and each maps to a failure mode you will actually meet:

1. **No live view of the world.** The model cannot browse, run your code, query your database, or check a package registry. (Products bolt on search/execution tools — that is the retrieval/agent machinery of Parts 2 and 5, not knowledge in the weights. When an assistant quotes a “current” version number, check *which layer* produced it.)

2. **Cannot execute or test your code.** Syntactically fluent, logically wrong output is a natural product of next-token prediction.
3. **No business context.** It doesn’t know your requirements, users, constraints, or risk tolerance — “best database” has an answer only relative to those.
4. **No guaranteed correctness, and no honest confidence signal.** Preference alignment (Part 1) rewarded assertive phrasing; the *tone* of an answer carries zero information about its reliability.

The three named failure modes, with their mechanics:

Hallucination — fluent fiction.

```
User:  "How do I use React's
useProductReviews hook?"
AI:    "Sure!
        import { useProductReviews }
        from 'react';
        const { reviews, loading }
        = useProductReviews(productId);"

Problem: no such built-in hook
exists.
```

Mechanism: `useProductReviews` is shaped exactly like a real hook; plausibility is the objective — truth is not in the loss function. *Defense:* verify every imported name against official docs before commit (the “invented APIs” checklist item from L_02, now with its explanation).

Outdated information.

Mechanism: weights stop at the training cutoff, and older idioms had more exposure than their replacements — class components before hooks, the deprecated API before its successor. *Defense:* check dates on what the model recommends, prefer official current docs, state your

pinned versions, and ask “is this current for this version?”.

Overconfidence.

Mechanism: hedged uncertainty ranks worse in preference data, so unsure answers still arrive in sure voices. *Defense:* ask for trade-offs and alternatives (“three options, when each fits”), which forces the model to expose the decision surface instead of a single verdict.

5.2 Ethics: licensing, academic integrity, bias

Copyright and licensing. Models are trained on copyrighted code, and generation can surface fragments of training material. Treat generated code like code from an unknown contributor: review it; watch for unusually distinctive blocks (long verbatim comments, odd naming, recognizable attribution — retrieval-flavored smells). Understand the license posture of your tool (enterprise tiers often add indemnification and filtering; consumer tiers vary). Keep proper licenses in your own projects, and never present AI-generated work as entirely unaided human work where that matters. A one-line disclosure in a README is normal professional practice:

```
This project uses AI-assisted
development tools; generated code has
been
reviewed, tested, and adapted to this
codebase's requirements.
```

Academic integrity. For students in this course:

Allowed ☑:

- AI for boilerplate and scaffolding.
- Explanations of concepts and error messages.

- Debugging assistance and hypothesis generation.
- AI output as a *study artifact* — then reproducing the logic yourself.

Not allowed ✕:

- Submitting AI-generated code you cannot explain line by line.
- Using AI in assessments where it is not permitted.
- Claiming AI-produced work as entirely your own unaided understanding.
- Letting AI bypass a learning objective the assignment exists to build.

Always: disclose usage where required, be ready to demonstrate understanding on request, use AI to *learn*, not to *avoid learning*. The L_02 test applies to this whole lesson: “*Can you explain what you shipped, without the model?*”

Bias. The model reproduces its training distribution, including its defaults: gendered assumptions in user-model fields, culturally-assumed UI defaults, accessibility oversights, English-centric examples. Bias auditing is a human responsibility no model can self-serve — for the same reason it cannot self-verify correctness. Review generated code for implicit assumptions, ask for inclusive-design checks explicitly, test with diverse inputs.

5.3 Privacy and data handling

Never include in prompts: API keys, passwords, tokens, production connection strings (nothing shaped like `sk_live_...` ever goes in a chat), customer data or PII, proprietary algorithms, unreleased or confidential business logic. The model layer has no “delete after use” you can see — the *product’s retention policy* is what protects you, and policies differ sharply.

⚠ Data-training and retention policies – snapshot, October 2026

High-volatility information: providers change these policies without notice. Re-check the linked pages before relying on any row — this table is a dated photograph, not a contract.

Where your prompt goes	Trains provider models?	Control	Source
Chat-GPT consumer (Free / Plus / Pro)	May train, by default	Toggle off “ Improve the model for everyone ” — trap: giving ☒ / ☒ feedback on a conversation can make <i>that</i> conversation trainable even after opt-ing out	OpenAI Help

Chat-GPT	De-fault:	Plan/	same
Standard	faults:	con-	Google
Do not assume	faults	ware	Gemini
training difference:	training	free vs paid	Herp
the control is the	control	the activ-	Privacy
ity setting, not the	plan	plan	Center
prize/	train-	ing for	
edu-	ing	2026-09-18	
OpenAI	Ther-	persv	

The habit that survives every policy change: **prompt as if the content will be stored, reviewed, and possibly used** — because for at least one of your tools, it will be. Use mock data, redacted identifiers, and abstracted business logic.

```

□ "Write a JWT auth middleware skeleton for Express"
✗ "Write auth using this secret: sk_live_abc123... and our customer table [pasted]"

```

5.4 Agents and oversight – the 2026 present

What older course material framed as a “future trend” is now shipped product, and the framing has to flip accordingly:

- **Coding agents are products:** OpenAI's Codex CLI / Codex agent line, Anthropic's Claude Code, and the GitHub Copilot coding agent all run multi-step, tool-using work in real repositories today.
- **MCP (Model Context Protocol) is a live, versioned open standard** — current spec revision **2026-07-28** at modelcontextprotocol.io — supported across Claude, ChatGPT, VS Code/Copilot, Cursor, and others. The retrieval and tool machinery of Part 2 is increasingly built this way: once, for many clients.

What has *not* changed is the vendor-neutral part — and it is this course’s stance too (see L 02’s “we defer the agents”):

- **Human approval for irreversible actions**
 - deploys, deletions, migrations, force-pushes are never auto-run.
- **Least-privilege tool scopes** — an agent gets read access until it has demonstrated it needs write access to a specific thing.

- **Review-before-merge** — agent-produced changes enter the codebase through the same diff review as any human PR.
- **Auditable logs** — every tool call should be replayable afterward. If you can't answer "what did it touch, and when?", don't run it.

An agent is context engineering + tool access + autonomy. Everything from Part 1 still applies — just now while nobody is steering turn by turn, which *raises* the verification bar rather than lowering it. In this course the position is unchanged from L_02: fundamentals in S0–S2 are built by hand; agents re-enter from S3 as an optional accelerator, with you still the gate.

5.5 Habits that keep working

The 80/20 rule. AI handles the 80% that is mechanical — boilerplate, structure, first-draft tests, doc comments, rename sweeps. You handle the 20% that carries responsibility — architecture, edge-case judgment, security review, business logic, and every final "yes" before commit.

1. You design the solution and its verification (the Concept step, per L_02)
2. AI generates structure and boilerplate
3. You review and modify — line by line
4. AI assists debugging (hypotheses, not verdicts)
5. You run the tests and the app
6. AI drafts documentation from the verified code
7. You commit — small, focused, checkpointed

Continuous learning — don't outsource the growth:

- **Read** every line the model writes before committing it.

- **Ask why** — then check the answer against a source, this lesson included.
- **Experiment** — deliberately break AI suggestions and watch what fails; that is how intuition is built.
- **Compare** multiple approaches; the model will happily generate three.
- **Research** unfamiliar patterns instead of accepting them because they ran.

Today's developer who learns *with* these tools becomes tomorrow's engineer who can supervise the agents everyone else merely runs.

5.6 Key takeaways

1. **One mechanism, many behaviors:** next-token prediction over a finite attended context — fluency, hallucination, context-sensitivity, and version-blindness all trace back to it.
2. **Tokens and context are a budget, not free storage:** estimate then verify counts; position matters (lost in the middle); length costs money.
3. **Training explains trust calibration:** alignment rewarded confident phrasing, so tone is never a reliability signal.
4. **Generation controls belong to a model generation:** reasoning-effort and temperature are different knobs; know which era's controls your model honours, and don't rely on undocumented ones.
5. **Products swap models weekly; the layers don't move:** the assistant you use is not the same thing as the model behind it, and a stated policy is not a promise — read the dated docs.
6. **Retrieval beats memory:** RAG and context engineering supply what frozen weights don't know; curate context like a review, not a backup.
7. **Workflow patterns are verification patterns:** focused passes, small diffs, test nets,

hypothesis-style debugging — each keeps *you* the gate.

8. **Limits are permanent:** no live world, no self-verification, no project knowledge, no decisions. Licensing, privacy, integrity, and bias duties stay human.

5.7 Going further

- **Back to the method:** AI Quick Start and its practice — the loop, the prompting toolkit, and the agentic framing this lesson assumed.
- **Next in the course:** continue to HTML & CSS Essentials, where the generate → verify → integrate → commit loop meets your first real front-end work.
- **Primary sources worth reading whole:** Lost in the Middle · DPO paper · Anthropic on context engineering · MCP specification · OpenAI embeddings guide